Doctoral Dissertation

Multiprocessor Real-Time Task Scheduling with Local Assignment Algorithm

Doan Duy

Supervisor: Associate Professor Kiyofumi Tanaka

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
[Information Science]
September, 2019

Abstract

With the bloom of smart devices and automatic systems, real-time embedded systems have been rapidly spreading into different aspects of daily life, science, and industry. Numerous real-time applications leads to introduction of efficient and powerful processing architectures such as multiprocessor platforms. The real-time embedded systems are nowadays becoming diverse and complicated in the sense that the system has to not only handle various types of tasks, but also manage multiple processing units. Real-time task scheduling in such systems is therefore challenging to researchers in spite of the fact that a number of scheduling algorithms have been introduced. One of the critical issues of multiprocessor real-time task scheduling is exploitation of the increased system capacity to improve system performance.

It is seen that the problem of real-time task scheduling in uniprocessor systems have been solved effectively with several optimal algorithms such as Earliest Deadline First (EDF) which achieves schedulability of 100% with low time complexity. The problem of multiprocessor real-time task scheduling is however faced with a trade off between the simplicity and optimality. Simple algorithms with low time complexity are introduced toward practical applications since they are commonly believed easier for developing, testing and implementing. Whereas, optimal algorithms are introduced with an attempt to utilize the entire capacity of the system. Unfortunately, simple scheduling approaches achieve low schedulability and optimal ones often cause high time complexity.

This dissertation is conducted in challenge of that scheduling trade-off. That is, scheduling varied workloads on multiprocessors at schedulability of up to 100% with a relatively low time complexity. To this end, algorithm, named *Local Assignment Algorithm (LAA)*, is introduced. In order to deal with static workloads such as periodic tasks, LAA exploits the notion of proportionate scheduling in the time-interval scheduling scheme. Calculations of LAA allow to guarantee full task assignments which reduce unnecessary idle times of processors on intervals for better utilization of the entire system capacity. In addition, LAA is associated with a selective method of arranging tasks to processors, which is effective to lessen task preemptions and migrations.

LAA is theoretically proved as an optimal scheduling algorithm with the schedulability of up to 100%. Software simulations are conducted to simulate the behaviors of LAA in comparison with several existing algorithms including Pfair, RUN, and semi-partition reservation. Assertive criteria consist of scheduler invocation, task migration, task preemption, and time complexity. Simulation results show that effectively LAA algorithm invokes approximately 50% of scheduler invocations fewer than the other candidates while still maintaining relatively lower time complexity. In addition, compared with the outstanding optimal algorithm RUN, LAA is comparable in terms of task migration and preemption.

Dynamic workloads such as aperiodic tasks cause difficulties to scheduling algorithms due to their unknown factors such as entering time, execution time. One of the difficulties is increase of runtime overhead. In fact, it is possible for aperiodic tasks to be scheduled in background of periodic ones with low scheduling cost in time. Nevertheless, this approach makes the aperiodic responsiveness increase. In this dissertation, combination of LAA and concept of servers, called $LAA+$, is introduced to deal with the hybrid task sets of periodic and aperiodic tasks. The aperiodic tasks are therefore scheduled on-line together with

periodic ones through service of servers. LAA+ is also extended with introduction of secondary scheduling events for better use of servers. The proposed scheduling approach then effectively improves the aperiodic responsiveness.

Software simulations are conducted to evaluate LAA+ in the context of mixture system of periodic and aperiodic tasks. The targeted criterion is switched to improving the aperiodic responsiveness while guaranteeing for periodic tasks to meet their deadlines. In evaluation, LAA+ continues maintaining the optimality of the original LAA. Simulation results show that LAA+ efficiently enhances the responsiveness of aperiodic tasks by approximately 30% compared with LAA and by about 10% compare with other scheduling candidates. Moreover, LAA+ still achieves lower runtime overheads and less number of scheduler invocations in comparison with the other candidates. Overall, achieving remarkably fewer scheduler invocations and relatively lower time complexity indicates that LAA and LAA+ have better exploitation of system capacity in a sense that the system time is preferred to spend for executing application tasks rather than the scheduling algorithm.

Finally, the Local Assignment Algorithm is implemented on a practical embedded system in order to confirm the applicability of the algorithm. The Zedboard FPGA Evaluation KIT is selected as the implementation environment. The system includes a built-in SoC Zynq7000 which supports a dual-core ARM Cortex-A9 processor. A multiprocessor real-time operating system (M-RTOS) in which LAA is utilized for scheduling application tasks is developed based on the ITRON RTOS kernel. It is found that the proposed algorithm is applicable to actual real-time embedded systems.

Keywords: real-time task scheduling, multiprocessor system, time complexity, aperiodic responsiveness, dynamic workload.

Acknowledgments

First of all, I would like to show my deep gratitude to my supervisor Associate Professor Kiyofumi Tanaka of Japan Advanced Institute of Science and Technology (JAIST) for his devoted guidance during my work. He appropriately gives me constant advice and kind encouragements, especially when I am faced with difficulties at work. It was a key decision that I enrolled at JAIST and then became a student in Tanaka laboratory from the master's program to the doctoral one. Looking back on my very first school days at JAIST, I have progressively grown with his suggestions and comments. Actually I had several times making mistakes in research; however, he still keeps belief in me and supports me to overcome difficulties. Without his support, I would not be able to finish my Ph.D degree successfully. Therefore, again I am extremely grateful to him.

I would also like to give many thanks to Professor Yasushi Inoguchi of JAIST. As my second-supervisor, he gave me different discussions and advices for several research situations. Thank to his advice, I can realize my shortcoming and enlarge my knowledge in the research field. I greatly appreciate Professor Mineo Kaneko of JAIST for his advice. His certain comments always help me to improve my work consistently. Furthermore, I would also like to give my gratefulness to Professor Yuto Lim of JAIST and Professor Yukikazu Nakamoto of University of Hyougo for their advice on my dissertation. Their comments are really contributive to the success of the dissertation.

Next, I am deeply thankful to Professor Shungo Kawanishi and Dr. Kotona Motoyama of JAIST for their sincerely sharing and discussion about emerging global issues and diversity. I acquired great knowledge from their courses, especially Diversity Studies and Global Leadership Training Seminar, which I have not found out at anywhere else.

I also appreciate Dr. Matthew N. Dailey, Dr. Mongkol Ekpanyapong, technical advisors and colleagues at Pineapple Vision System Co., Ltd. (PVS), Pathumthani, Thailand. They enthusiastically supported me so that I could finished my internship successfully. Experience at PVS is very helpful for me to complete my doctoral research.

Next, I am thankful to all members in Tanaka laboratory and Kaneko laboratory, JAIST officers and my friends at JAIST and over the world who have cheered me up and helped me to keep going on study and life. With them, I actually had a joyful life in Japan.

Finally, I would like to thank my family members. They alway stay with me, believe in me and give my endless love. Their belief and encouragement are my motivation to overcome difficulties.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Trend of multiprocessor embedded system

Needs for multiprocessor systems

Recent decades have seen the popularization of smart devices and automatic systems which are developed in order to support a wide range of applications in daily life, science, and industry. Real-time embedded systems therefore also play a very important role in the today's technology-driven world. Due to the increasing diversity and complication of targeted applications, traditional uniprocessor systems (systems with single processor) become unaffordable. More powerful and flexible embedded systems are needed to

Table 1.1: Representatives of multiprocessor system-on-chips

Name of MPSoC	**Year of announcement**	**# of processors**	**Application**
Lucent Daytona [1]	2000	4	Wireless base stations
C-5 Network Processor [2]	2001	16	Package processing in networks
Philips Viper Nexperia [3]	2001	2	Multimedia
STMicroelectronics Nomadik [4]	2003	3	Cell phones
Texas Instrument OMAP [5]	2004	2	Cell phones
ARM MPCore Family [6]	2005	4	Different applications
Intel IXP2855 [7]	2005	16	Cryptography functions
Cisco Silicon Packet Processor [8]	2005	192	High-end CRS-1 Router

handle such applications. As an incarnation of very large scale integration (VLSI) tech-

nology, multiprocessor system-on-chips (MPSoCs) emerged as a substantial solution for the demand. Table 1.1 shows a list of MPSoCs which have been introduced for different applications. The number of processors integrated on a chip is accordingly varied from two to over one hundred processors. Besides the significant increase of system capacity with many processing units, MPSoCs exhibit diverseness to support from general systems like cell phones to specific ones like network routers.

The dominance of multiprocessor embedded systems is easily seen because of the fact that MPSoCs are now made in reasonable cost and size with the development of manufacturing technologies. In addition, FPGA (field programmable gate array) technology is also a big support to the increase of MPSoCs. There are actually a number of MPSoC platforms built in FPGA which allows flexible configurations for various purposes. Xilinx and Intel are, for example, providing several MPSoC FPGA development boards such as Zedboard Zynq7000 Evaluation KIT [9], Intel®Stratix™10 Development KIT [10].

The emergence of MPSoC makes it attractive to researchers in the embedded field. Especially, the scheduler in such systems becomes important and challenging in a sense that it has to not only handle different task sets, but also manage multiple processors.

1.2 Challenges of multiprocessor scheduling

1.2.1 Exploitation of system capacity of multiprocessors

We ideally expect that upgrading from uniprocessor systems to n-processor systems is providing n-fold increment in computation capacity. This expectation is unfortunately impractical in fact. There are several reasons found. The first reason is that most actual computational problems cannot be effectively executed in parallel. Equation 1.1 exhibits analysis of Amdahl's law [11] to the parallel computation of a problem.

$$s = \frac{1}{(1-p) + \frac{p}{n}} \tag{1.1}$$

where s is the reachable speedup, p is the ratio of the parallel portion of the problems, and n is the number of processors available. Consider a system of four identical processors executed a problem 90% of which can be processed in parallel. Applying Amdahl's formula, the reachable speedup is as follows:

$$s = \frac{1}{(1-0.9) + \frac{0.9}{4}} \approx 3$$

Obviously, the obtained speedup is much lower than the expected one.

Similarly, in the situation of different problems (tasks), in order to achieve the full speedup of multiprocessors, tasks must be possible for parallel processing. However, even in case that parallelism of tasks is satisfied, the system capacity of multiprocessors is actually decreased due to system overhead caused by task scheduling and context switches. Dealing with the diversity of tasks and numerous processors, time complexity of scheduling algorithms is significantly increased. This make increase of runtime overhead, a part of system overhead. Task preemptions and task migrations are major context switches in multiprocessor systems. For better exploitation of the system capacity of multiprocessors, such sources of system overhead needs to be reduced.

1.2.2 Time complexity of multiprocessor scheduling

For the emergence of MPSoCs, the need of real-time operating systems (RTOSs) which provide multiprocessor support are increasingly drawing researchers and engineers. As an important part of RTOSs, the scheduler has been received serious research attentions as a result. Although a number of scheduling algorithms have been proposed, effective approaches for multiprocessor real-time task scheduling still remain challenging topic.

It is obvious that the problem of real-time task scheduling on uniprocessor systems has been solved successfully with a myriad of optimal scheduling algorithms such as Earliest Deadline First (EDF) [12], Least Laxity First (LLF) [13]. Such optimal uniprocessor algorithms are capable of scheduling task sets of utilizations of up to 100% with low time complexity. However, when considering the multiprocessor systems, researchers are faced with difficulty: increase of time complexity of scheduling algorithms. This leads to two concepts of scheduling: simplicity and optimality.

Algorithms on the concept of simplicity are targeted to schedule tasks with low time complexity. Such simple scheduling algorithms are commonly believed to be easier for developing, testing and deploying. Whereas optimal scheduling algorithms are expected to exploit the entire system capacity in order to improve the system performance; that is, they can schedule any task sets of utilization of up to 100%. However, a mutual trade-off appears between the two concepts. Simple solutions achieve low schedulability and conversely, optimal solutions often come up against high time complexity.

Scheduling algorithms with the simplicity

At the early period of concern, researchers strove to extend well-known uniprocessor scheduling algorithms to the context of multiprocessor. Typically, EDF and LLF were investigated in the paradigm of global scheduling. In the global scheduling, a common ready queue of the entire task sets is preserved for the scheduler. Tasks are likely to migrate among processors for execution during their existing period in the system. This approach is advantageous to come into the simplicity of such algorithms. Nevertheless, the optimality is untenable and the obtained schedulability of these approaches is considerably lower than the system capacity available [15]. Figure 1.1 shows an simple example where

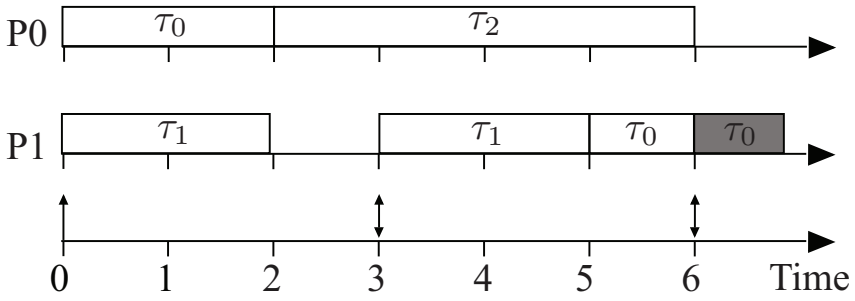

Figure 1.1: Example of EDF non-optimal on multiprocessors

the EDF scheduling is faired to schedule tasks on multiprocessors. It is supposed to have three periodic tasks scheduled on two processors: τ_0 and τ_1 request executions of two unit times (ticks) for every three ticks and τ_2 requests four-tick executions for every six ticks.

The total utilization of tasks is at 100%. If tasks have their phase time at 0, the EDF scheduling is failed at time 6 when τ_0 misses deadline for the second job.

After that, several scheduling algorithms including EDF-US[x] [16, 17], ED/LL, EDZL [18, 19], and EDCL [20] were successively proposed so as to improve the schedulability. Those solutions did not achieve the full schedulability yet.

Uniprocessor scheduling algorithms were alternatively applied to multiprocessor scheme in the manner categorized as partitioned scheduling. In partitioned schedulings, task sets are divided into concrete subsets which are affordable with unit processor. Processors in the system are each associated with an ready queue of the assigned tasks and the task scheduling on a processor is independent of each other. Well-known algorithms like EDF are independently applied on processors as a result. DC^2 [21] and partitioned-EDF [22] algorithms are appeared in this group.

A big advantage of the partitioned view is obviously that the problem of multiprocessor scheduling is reduced to the problem of uniprocessor one. In the other words, simplicity of uniprocessor optimal schedulings is prospectively gained. Partitioned schedulings, however, found themselves confronted by substantial disadvantages. First, partitioning task set into subsets is actually a bin-packing problem which is one of NP-hard problems. Non-optimal heuristic methods are often employed for task partition. Consequently, it is not always for task sets to be separated into subsets completely. Second, the fact that task migration is not allowed in partitioned scheduling causes negative impacts on the schedulability. Tasks are not accepted if their utilization exceeds the remaining capacity of individual processors although the remaining capacity of the whole system is still available. This often happens with heavy tasks which have high utilization. The achieved schedulability of partitioned schedulings is hence truly low [15, 23] and the system does not yield high performance.

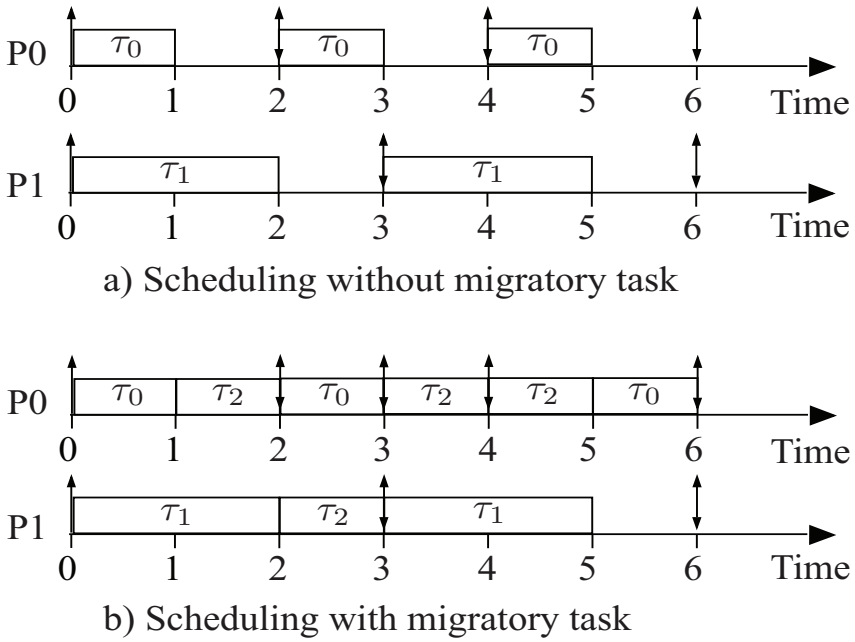

Figure 1.2: Impact of task migration on the schedulability of multiprocessor schedulings

Figure 1.2 shows the impact of task migration on the schedulability of the partitioned

schedulings. The system is supposed to have three periodic tasks τ_0, τ_1 and τ_2 scheduled on two processors. τ_0 requests unit executions for every two ticks and τ_1 and τ_2 request two-tick executions for every three ticks. The system utilization is approximately of 90%. The first two tasks are partitioned to processors so that τ_0 is executed on P_0 while τ_1 is allocated on P_1 as showed in Figure 1.2(a). It is then impossible to assign τ_2 to a processor since the cumulative utilization would exceed the processors' capacity.

Figure 1.2(b) shows a more preferable perspective where τ_2 is accepted and allowed to migrate between the two processors. The scheduling scheme is realized as semi-partitioned scheduling. A number of semi-partitioned solutions have been conducted to improve the schedulability of partitioned scheduling for practical applications. Semi-partition emerged as a balance between global scheduling and partitioned scheduling. Semi-partitioned approaches therefore allow migratory tasks besides the ones that are fixedly assigned to processors. Migratory tasks are moving among processors during their presentation to receive processor allocations. To this end, semi-partitioned algorithms are assisted with a splitting method which can split a task into subtasks so that the subtasks are appropriately assigned to processors. Works [24, 25, 26, 27, 28] are found in this scheduling class. Especially, semi-partitioned reservation (SPR)[29], which employs the $C = D$ scheme [30] as its splitting method, appears as a near-optimality solution. This approach can reach the schedulability of 99+% with relatively low time complexity. SPR has high potential to be applied to practical real-time systems.

Scheduling algorithms with the optimality

In the last decades, algorithms pursuing the optimality have been gained much attention to fully exploit the system capacity of multiprocessor. Well-known optimal multiprocessor scheduling algorithms are scrutinized in the paradigm of global scheduling. In early period of investigation, Dertouzos, et al. [31] took account of the problem of hard real-time task scheduling on multiprocessor systems through a scheduling game representation. In this work, a critical restriction of multiprocessor scheduling is disclosed; that is, without a priori knowledge about task sets, such as start time, an optimal multiprocessor scheduling algorithm is unachievable.

Periodic tasks with release times predictable are early gained attention of researchers. Pfair with the notion of proportionate scheduling [32] is very first solution achieved the optimality. Regarding tasks' utilization, Pfair guarantees a fairness of scheduling for periodic tasks at fine time scale; it means the scheduler is invoked to make schedules of tasks at every time tick. In spite of achieving the optimality, Pfair approach is faced with a difficulty of numerous scheduler invocations. In addition, since task executions are tentatively fragmentary by unit times, Pfair significantly causes task preemptions and migrations as sources of runtime overhead.

Another drawback of Pfair scheduling is that the system is not a work-conserving scheduler which is trying to maintain scheduled resources (processors for instance) busy if tasks are available in the system. In other words, since the amount of times that tasks receive in Pfair is restrictedly proportional to the tasks' utilization (weight), processors may be put in idle time unnecessarily. This is motivation of fast scheduling [33] and early-release fair scheduling [34] which improve Pfair to be a work-conserving scheduler.

Is guaranteeing the fairness of scheduling tasks at every unit time a must? The answer for this question was found in the following works: *Boundary Fair* (BF) [35], and

DP-WRAP [37]. These are alternative implementations of proportionate scheduling for interval (time period), which is considered as coarse time scale schedulings. The scheduler is consequently invoked at specific points in respect to time intervals rather than unit times. The intervals are decided by the periodic tasks' deadlines, which forms a so-called *deadline partitioning*. This approach has advantage of reducing the number of scheduling points. Task preemptions and migrations are also alleviated since tasks have chance executing seamlessly. Nevertheless, BF and DP-WRAP have their own limitations. First, amounts of time that the scheduler allocates to tasks on each interval are limited as a result of the proportionate scheduling. The allocation of BF is likely better with the unit extra amount than DP-WRAP. Processors' idle times may occur unnecessarily and the system capacity is therefore not used effectively. In addition, the context switching points in DP-WRAP scheduling are planned based on the multiplication of tasks' utilization by the interval length. This calculation often results in decimal values. As the system time is allocated to tasks by integer numbers of slots, complicated calculations are required to determine the actual context switching points at runtime so as to guarantee the schedulability.

In later years, *Largest Local Remaining Execution time First* (LLREF) [36] was proposed to the multiprocessor scheduling problem. LLREF is another coarse time scale scheduling. Intervals are decided by any two consecutive job releases, which forms *T-L Planes*. Like BF and DP-WRAP, LLREF shows effectiveness on reducing the number of scheduler invocations. However, task preemption and migration are still considerably high.

In 2011, RUN [38] was emerged as an outperforming optimal algorithm in terms of task preemption and migration. RUN introduces a novel scheme to cluster tasks into proper sub-systems which are served with an appropriate number of processors. Tasks in RUN scheduling are each accompanied with a (virtual) dual task the execution time of which is the complement of the real task's execution. Based on the duality, a structure called *reduction tree* is produced for a sub-systems in order to select task for execution at runtime. In the same family of schedulers as RUN, Quasi-Partitioned Scheduling (QPS) [39] were proposed. QPS offers a dynamic adaptation to the system workload. Namely, an efficient way was introduced to mutually switch between partitioned EDF scheduling and global-like scheduling when the workload changes.

Although effectively solving the scheduling problem of periodic tasks, these optimal algorithms are experienced with difficulty in scheduling dynamic workloads such as aperiodic tasks. This is clarified in Section 1.2.3.

1.2.3 Dealing with diverse workloads

Due to the diversity of applications, dynamic workloads more frequently occur in embedded systems. Aperiodic tasks which are entering the system irregularly without any information known a priori are typical dynamic candidate. While periodic tasks are sufficiently solved with a variety of solutions including optimal algorithms, aperiodic tasks are difficult to be deal with due to their unknown information. Reminding that according to work in [15] the optimality is impossible without known information of tasks. Conventional approach to deal with aperiodic tasks is therefore scheduling them in background. Background scheduling assigns aperiodic tasks processor times unused by periodic tasks. Aperiodic tasks are experienced long response times as a result. Figure 1.3 exhibits an

example of scheduling aperiodic task in background. The periodic task set is scheduled as the same as the example in Figure 1.2(b). An aperiodic task is supposed to enter the system at time 2. The aperiodic task must wait for execution until time 5 when processor P_1 is empty of periodic tasks' execution. It is then finished at time 6 with the response time of 4. Intentionally, we can see that the aperiodic task can be scheduled for execution at time 3, ahead of the second instance of τ_1, without violating scheduling constrains of periodic tasks.

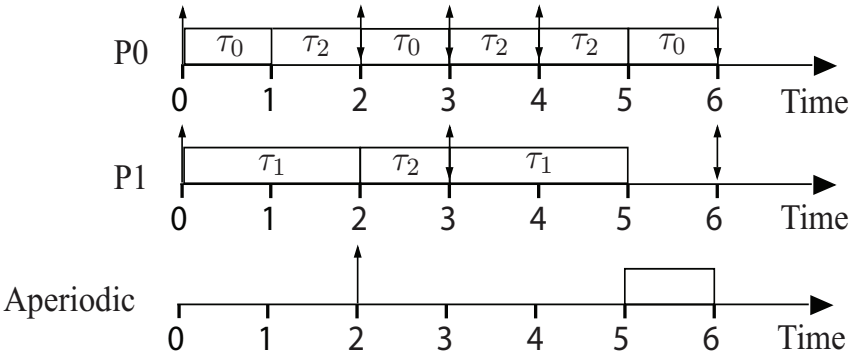

Figure 1.3: Scheduling aperiodic task in background

Another approach is that aperiodic tasks are scheduled on-line together with periodic ones. This square deal is in general believed to improve the aperiodic responsiveness. Considerable issue of the on-line scheduling is high runtime overhead. On one hand, partition-based schedulings employ complex heuristic algorithms for task grouping at the system initialization time. With the appearance of aperiodic tasks, the partition procedure needs to be repeated to assign the tasks to an appropriate processor. The runtime overhead significantly increases as a result. Moreover, it may be failed to assign aperiodic tasks to processors due to the non-optimality of the heuristic algorithms.

On the other hand, several optimal algorithms also process complex procedures off-line to reduce the runtime overhead. The procedure of building the reduction tree in RUN is for example. In order to schedule aperiodic tasks on-line, such complex procedure needs to be rerun to include the aperiodic tasks at their released times. This perspective is actually worse in terms of runtime overhead.

In order to prevent the repeat of complex procedure, concept of servers is introduced to dedicatedly serve aperiodic tasks at runtime. This scheduling scheme was found in work of Srinivasan et al. [49] where servers are integrated into Pfair scheduling scheme for scheduling mixed task sets. Although improving the aperiodic responsiveness, the problems of enormous scheduler invocations, task preemptions and task migrations still remain obstacles of the work.

The review on multiprocessor real-time task scheduling shows that several critical issues still exist needing researchers' attention despite the fact that a number of solutions have been proposed. Existing problems include the balance between simplicity and optimality for effective exploitation of the system capacity, dealing with the variation of workloads, and increase of runtime overhead. Such existing problems motivate study of the dissertation.

1.3 Contribution of dissertation

This work is conducted to propose an effective scheduling algorithm for the problem of multiprocessor real-time task scheduling. The proposed algorithm is aimed at challenging the trade-off between the simplicity and optimality. Is an optimal scheduling algorithm with relatively low time complexity achievable? This question is the motivating philosophy of the research. Accordingly, an optimal algorithm is the one can schedule any task sets at up to 100% of the system capacity. The optimality then reflexes the maximum schedulability of the algorithm. It is believed that the higher the schedulability is, the better the system exploitation is. Low time complexity means the system will spend less processor time for scheduling, which effectively reduces the runtime overhead. On the other hand, characteristic of low time complexity makes it possible for the algorithm to be implemented in practical systems with limited hardware supports like embedded systems.

The research is therefore contributing to improving the system performance and toward practical implementation.

1.3.1 Research objectives

The research pursues the following objectives:

- Achieving the optimality. This target is to guarantee the schedulability of the proposed algorithm. Generally, the higher the schedulability is guaranteed, the better the system capacity is exploited. A full schedulability is therefore targeted (in theory) so that the proposed algorithm can schedule any task sets of the utilization of up to 100%.

- Being capable of scheduling hybrid task sets. Task sets are targeted to mixture of periodic and aperiodic tasks. The proposed algorithm therefore meets the diversity of applications. Additionally, the objective makes the research applicable to practical embedded systems where hybrid task sets more frequently occur.

- Reducing runtime overhead. The proposed algorithm accordingly achieves the optimality with relatively low time complexity and less number of scheduler invocations. The accumulative runtime overhead is then reduced so that the system times can be spent better for task executions.

- Maintaining comparable number of task preemptions and migrations. These criteria also contribute to the system overhead. The proposed algorithm therefore maintains the criteria comparable with the existing algorithms.

- The proposed solution is applicable to practical embedded systems.

1.3.2 Research process

Toward the goals specified above, the research process is designed as follows.

1. We first propose an effective scheduling algorithm called Local Assignment Algorithm (LAA) for the problem of periodic tasks on multiprocessors. LAA applies the

notion of proportionate scheduling and deadline partitioning to pursue the optimality. That is, applying the deadline partition, time is divided into intervals (time slices) based on job releases. The executions of jobs of tasks are then split into portions which are executed separately on different intervals. How long each of the portions is is calculated using the fairness of proportionate scheduling. The portions of jobs are then assigned to processors for execution during the interval. The assignment of jobs, including the amount of execution and the processor allocation, is local on the involved interval; it means a job can be moved to another processor with different amount of execution on another interval. This locality makes it named "local assignment." A full assignment scheme is introduced in LAA to fully utilize the system capacity. Theoretically, LAA is proved to achieve the schedulability of up to 100%. The effectiveness of the proposed algorithm is exhibited with extremely fewer scheduler invocations and relatively low time complexity in comparison with the other scheduling candidates. Whereas, the results of task preemption and migration satisfy the goal specified above.

2. We then extend LAA to schedule hybrid task sets of periodic and aperiodic tasks. Concept of servers are integrated into the system to dedicatedly serve aperiodic tasks. The extended algorithm is called LAA+. Modeled like periodic task, servers can be scheduled on-line together with periodic tasks in LAA+. LAA+ then guarantees the optimality from the original algorithm. Several techniques such as secondary scheduler invocation are applied in order to enhance the aperiodic responsiveness. As a result, aperiodic tasks' response times are shortened efficiently without significantly increasing the time complexity.

3. In addition, a multiprocessor real-time operating system (M-RTOS) in which LAA is exerted to schedule application tasks is developed for practical applications. The M-RTOS is an extended version of ITRON real-time operating system [50] for multiprocessor systems. The system is implemented on Zedboard FPGA Evaluation KIT which includes a Zynq-7000 MPSoC of a dual-core ARM Cortex-A9 processor. This shows the applicability of the proposed algorithm.

1.4 Outline of dissertation

The remaining of this paper is organized as follows. Chapter 2 is specification of the targeted system with tasks' characteristics and related criteria of multiprocessor real-time task scheduling. Chapter 3 is introduction of related algorithms and techniques that are applied to solve the problem of periodic schedule. Chapter 4 shows details of Local Assignment Algorithm. Chapter 5 describes a preliminary of the concept of server in order to deal with aperiodic tasks. Chapter 6 shows details of the extended Local Assignment Algorithm for hybrid task sets. Implementation of multiprocessor real-time operating system with Local Assignment Algorithm is presented in Chapter 7. Finally in Chapter 8 is a summary of this dissertation and discussions about future work.

Chapter 2

System model

2.1 Multiprocessor architecture

In this study, we aim at the system of symmetric multiprocessors (SMP). Figure 2.1 illustrates model of SMP in hardware and software architectures. The system includes m identical processors each of which has an unit capacity. Processors are integrated with private cache memory (level 1 cache) for instructions and data. All processors are fully connected to a single, shared memory and input and output devices (I/O) via system bus with bus arbiter. In addition, the system is managed by a single operating system called multiprocessor real-time operating system (M-RTOS). Application tasks are common among processors; that is, an application task can be executed on any processor and likely to migrate among processors during its presentation period. Processors are thus considered as shared resources that needs scheduling in the context of the dissertation.

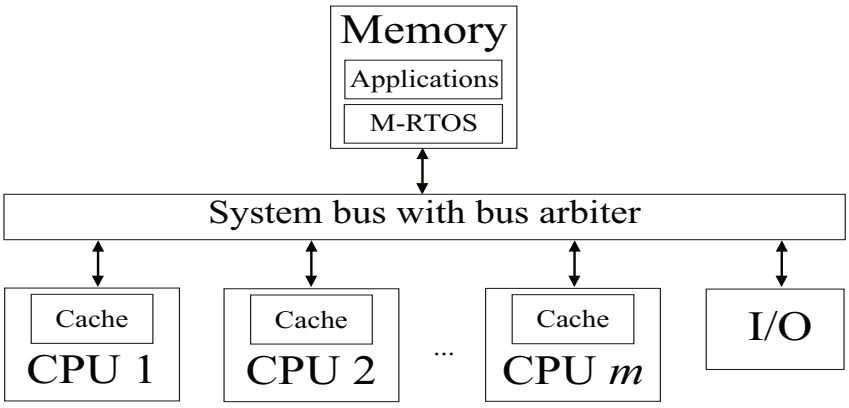

Figure 2.1: Model of symmetric multiprocessors

2.2 Task characteristics

The task system is targeted to mixtures of periodic and aperiodic tasks. The specification of tasks are as follows.

2.2.1 Periodic tasks

Periodic tasks are the ones that regularly release jobs as their instant works. A periodic task τ_i is typically specified by two attributes: a fixed rate μ_i and a period T_i; in notation $\tau_i = (\mu_i, T_i)$. μ_i represents the processor utilization of the task which refers to the fraction of processor time spent to finish a job of the task on a single processor. The processor utilizations are therefore less than or equal to 1; namely, $0 < \mu_i \leq 1$. T_i represents the time span between two consecutive job releases of the task and therefore $T_i \in Z^+$. Accordingly, the worst-case execution time (WCET) of task τ_i, the longest execution time among the task's jobs, is defined as:

$$C_i = \mu_i * T_i \tag{2.1}$$

We denote n_p $(0 < n_p)$ as the number of periodic tasks presenting in the system. Furthermore, we assume that periodic tasks have their phase time (the first time releasing job) at 0.

In real-time execution, a periodic task is defined as an infinite sequence of jobs. Periodic tasks are hence predictable in terms of release time and deadline. That is, the kth job of task τ_i would be released at time $r_{i,k}$ and receive an absolute deadline $d_{i,k}$ as follows:

$$r_{i,k} = k * T_i \tag{2.2}$$

and

$$d_{i,k} = (k+1) * T_i \tag{2.3}$$

where $k \in N$.

This definition indicates that the absolute deadline of a job is the same as the release time of the next job. As a result, there are no two or more jobs of a task simultaneously presenting in the system. When released, a job of the task requests an actual execution time $c_{i,k} \leq C_i$ which must be accommodated by the deadline $d_{i,k}$. Periodic tasks are thus considered as hard real-time tasks.

In this study, the system is restricted to feasible systems where the total task utilization does not exceed the system capacity. That is,

$$U_p = \sum_{i=0}^{n_p-1} \mu_i \leq m \tag{2.4}$$

Task sets satisfying Equation 2.4 are considered as feasible sets for scheduling. Situations of system overload where the total task utilization exceeds the system capacity are out of the scope of this study and tentatively taken into account in future.

2.2.2 Aperiodic tasks

Aperiodic tasks are the ones that enter the system irregularly and then exit after finishing their execution. Generally, there are two types of aperiodic tasks: soft real-time aperiodic task and hard real-time aperiodic task. While no deadline constraints are required for the soft ones to finish their execution, the hard ones are somehow associated with absolute deadlines when they are released. In this study, The soft real-time aperiodic tasks are targeted.

Different from the context of periodic tasks where guaranteeing to meet their deadlines is the most important requirement, aperiodic tasks are expected to finish as soon as possible. As a result, the scheduling attempt is to improve the responsiveness of aperiodic tasks while guaranteeing the deadline constrains for periodic ones.

Finally, tasks are fully preemptive and independent of each other. Namely, a task can be preempted unlimitedly by one another higher prioritized task and then resume their execution during its presentation period. For the ease of understanding, Table 2.1 summarizes notations related to the system specification in this study.

Table 2.1: Notations of system specification

Notation	**Description**
m	Number of processors
n_p	Number of periodic tasks
τ_i	A periodic task
μ_i	Utilization rate of periodic task
T_i	Period of periodic task
C_i	Worst-case execution time of periodic task
$r_{i,k}$	Release time of the kth job of task τ_i
$d_{i,k}$	Absolute deadline of the kth job of task τ_i
$c_{i,k}$	Actual execution time of the kth job of task τ_i
U_p	Total utilization of periodic tasks

2.3 Criteria of multiprocessor real-time task scheduling

For the ease of understanding and evaluating, in this section basic criteria of multiprocessor real-time task scheduling are defined. Definitions include optimality, time complexity, runtime overhead, scheduling invocation, task preemption, task migration, and response time. Targets of each criterion in real-time task scheduling are also discussed.

2.3.1 Optimality

In this dissertation, a scheduling algorithm is said to be optimal if it guarantees the schedulability of 100% of the system capacity. The schedulability is typically considered as the total utilization of tasks that can be scheduled without any deadline miss. Schedule of tasks without deadline miss is call a valid schedule. Therefore, an optimal algorithm

always finds out a valid schedule for any task sets with the total utilization of up to 100% of the system capacity. Normally, the optimality of scheduling algorithms is proved in theory where costs (overheads) of scheduling and context switches are excluded.

2.3.2 Time complexity and runtime overhead

Time complexity (computation complexity) is considered as the timing cost for executing scheduling algorithms at scheduling points. Execution of scheduling algorithm significantly contributes to runtime overhead. In evaluation, runtime overhead is generally calculated by counting the number of cycles during which the instructions in the algorithm are executed.

In this study, two types of runtime overhead are taken into account: maximum runtime overhead per tick and accumulative runtime overhead.

- *Maximum runtime overhead per tick*

  This runtime overhead indicates the worst-case computation of the scheduling algorithm over all scheduling points. Consider tick t as a scheduling point. Let $O_e(t)$ be the runtime overhead caused by execution of scheduling algorithm and $O_r(t)$ be the additional runtime overhead of recording system information at tick t. Then the total runtime overhead at tick t (denoted as $O(t)$) is calculated as:

  $$O(t) = O_r(t) + O_e(t) \tag{2.5}$$

  The maximum runtime overhead per tick ($O_{max/tick}$) is then calculated as:

  $$O_{max/tick} = max(O(t)) \tag{2.6}$$

  The maximum runtime overhead per tick can be used to assess the time complexity of an scheduling algorithm.

- *Accumulative runtime overhead*

  The accumulative runtime overhead is the sum of runtime overheads over all scheduler invocations. This parameter has meaning to evaluate the performance of the system. That is, less accumulative runtime overhead means that more system capacity is utilized for application tasks' execution. Let K denote the total number of scheduler invocations, the accumulative runtime overhead (denoted as O_{tot}) is calculated as:

  $$O_{tot} = \sum_{t=1}^{K} O(t) \tag{2.7}$$

2.3.3 Scheduler invocation

In real-time embedded system, scheduling invocation (or scheduling point) is defined as the time when the scheduler is invoked to make scheduling decisions. In multiprocessor systems, a scheduling decision basically includes two processes: (1) selecting tasks for execution; (2) assigning the selected tasks to processors.

The former process decides necessary context switches among tasks to guarantee scheduling requirements such as deadline meet. When the scheduler needs to be invoked

is therefore importantly concerned in RTOSs to assure the system safety. How often the scheduler needs to be invoked is another critical issue since it is related to increase of runtime overhead (the accumulative runtime overhead). A scheduling algorithm which can guarantee the system safety with less scheduler invocation is actually more preferable. The latter process is to distribute the selected tasks to processors for execution. Effective assignment methods can support to improve scheduling criteria including task preemption and migration which are described in the following sections.

Moreover, typically the scheduler is invoked at integral times to design the schedule of tasks. Tasks are then received amount of times in integer for their execution.

2.3.4 Task preemption

In real-time embedded system, task preemption is defined as the event when a job of task which has not finished its execution yet is preempted by one another job of the other tasks. The preempted job is then resumed its execution according to the scheduling results.

As mentioned above, tasks are all supposed to be preemptive. How many times a job is preempted is unrestricted. Nevertheless, since task preemptions cause context switches which increase the system overhead, reducing task preemptions is important in high performance systems.

2.3.5 Task migration

Task migration is a new criterion rising in the context of multiprocessor systems. An event when a job of task migrates from a processor to one another processor is counted for task migration. When a task migration happens, a process called *cache coherence* is invoked to keep data required for the job's execution consistent between the original processor and the destination one. This causes system overhead of task migration.

In symmetric multiprocessor systems, we assume that the system overhead caused by one task migration is constant. Namely, timing cost for a job of task to migrate from processor 0 to processor 1 is the same as the cost when it moves from processor 0 to processor $m - 1$.

Like task preemption, reducing task migration is important in high performance systems.

2.3.6 Response time

Response time is defined as the total amount of times it takes for the system to response to a task's request. In other words, response time is the sum of the time the task spends waiting for execution and the task's actual execution time. Formally, the response time of a task is calculated as:

$$R(\tau) = f_\tau - r_\tau \tag{2.8}$$

where r_τ and f_τ are the release time and the finishing time of the (job of) task, respectively.

For periodic tasks, the responsiveness is a minor issue which is often ignored in scheduling consideration. Whereas, this is the top requirement for aperiodic tasks; the response times of aperiodic tasks are expected to be as short as possible. Therefore, it is prevalent in the context of mixture system that attempt is spent on enhancing the aperiodic responsiveness while maintaining for periodic tasks to meet their deadlines.

Chapter 3

Periodic Task Scheduling

Periodic tasks are considered as the most important objective of the task scheduling problem. Ensuring the schedule of periodic tasks becomes a must for any scheduling algorithm and draws many attentions from researchers. As fundamentals of the research of the dissertation, typical approaches of periodic task scheduling on multiprocessors are introduced in this chapter.

The chapter will cover the following approaches:

- Fluid scheduling;
- Proportionate scheduling on time quanta; and
- Proportionate scheduling on time interval.

3.1 Fluid scheduling

The problem of periodic schedule was first taken into account by Liu, et al. in 1969 [51]. In this work, n periodic tasks are scheduled to use m shared resources. A periodic task τ is characterized by weight $\mu = C/T$ $(0 < \mu \leq 1)$. A periodic schedule is then considered to allocate to task τ exactly C units of time in each interval $[k * T, (k + 1) * T)$ for all $k \in N$. C and T are accordingly known as task's (worst-case) execution time and period, respectively. Periodic schedule is the basic concept to guarantee the resource allocation for every job (instant execution) of the periodic tasks.

There were a number of works conducted to implement the periodic schedule in multiprocessor systems. The ideal offer was found in the model of fluid scheduling [52]. The fluid scheduling assigns resources to tasks at constant rates over time. That is, at any time $t \geq 0$, a periodic task τ_i has been executed for exactly $\mu_i * t$ units of time where μ_i is the task utilization (weight). Figure 3.1 shows the curve of fluid scheduling model. The diagonal path indicates the constant rate of resource that is allocated to a job of task which is released at time r with execution time c and received absolute deadline $r + d$.

However, since the scheduler in practical systems is invoked at integral time and allocates tasks integral amounts of time, exactly guaranteeing the fluid scheduling is impossible. Figure 3.2 exhibits difference between fluid scheduling and practical scheduling. The red broken path is the actual rate of resource allocation. The diagonal parts implies the scheduled periods of the task and the horizontal parts expresses the un-scheduled ones.

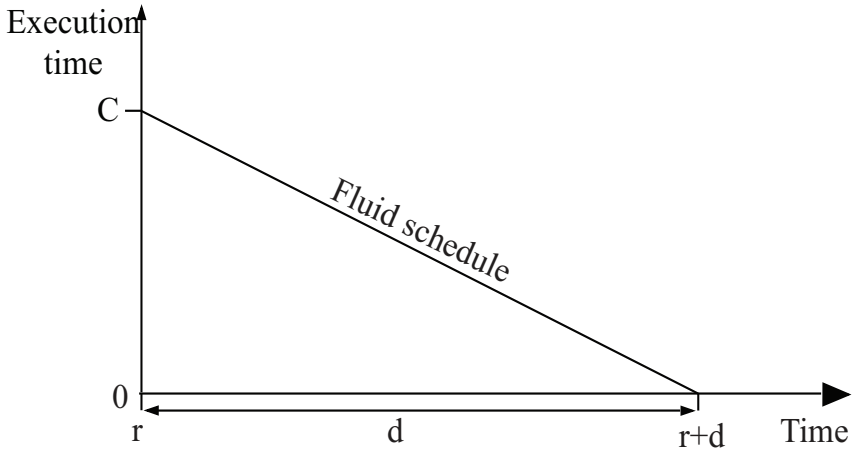

Figure 3.1: Concept of fluid schedule

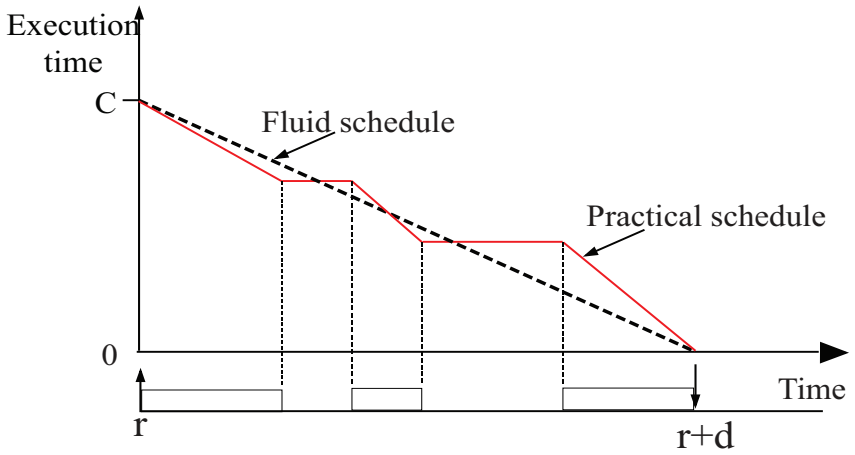

Figure 3.2: Difference between fluid schedule and practical schedule

Since scheduling periodic tasks at constant rates is impractical, alternative approaches with dynamic rates are concerned.

3.2 Proportionate scheduling on time-quanta

Proportionate scheduling was early introduced in Pfair algorithm [32]. Pfair is considered as stronger fairness constraints compared with the original notion of periodic schedule and as a relaxation of the fluid scheduling. Given a problem of n periodic tasks scheduled on m identical processors where each task τ_i has a rational weight μ_i $(0 < \mu_i \leq 1)$ for all $i \in N$ and $\sum_{i=0}^{n} \mu_i \leq m$. Pfair maintains a proportionate progress so that each task is scheduled resources in proportion to its weight at every unit time (tick). Specifically, at time t, a task τ_i with its weight of μ_i must have been scheduled either $\lfloor \mu_i * t \rfloor$ or $\lceil \mu_i * t \rceil$ times.

Figure 3.3 shows an example of Pfair scheduling in difference from the original periodic schedule. It is supposed that a periodic task τ has its weight $\mu = 1/3$ and period $T = 6$.

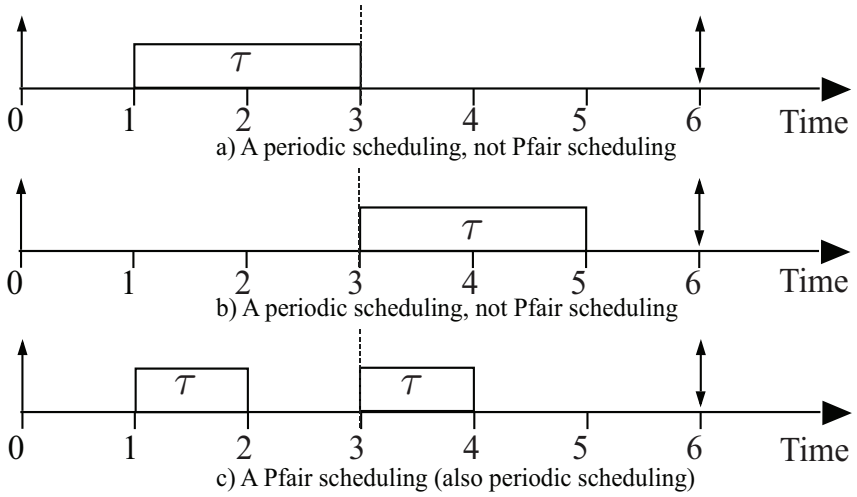

Figure 3.3: Pfair scheduling vs original periodic scheduling

The task therefore requests execution of two unit times ($= 1/3 * 6$) for every six ticks. In respect of the definition of periodic schedule, the task can be allocated two unit times at any ticks during the presentation period. However, under the constraints of Pfair scheduling, at time 3, the task must have been received exactly 1 unit time. Therefore, while all three cases displayed are periodic schedule, the schedule in Figure 3.3(c) is Pfair scheduling only.

Obviously, the scheduler utilized Pfair algorithm needs to be invoked at every system tick to guarantee the proportionate progress, which also makes Pfair as time-quanta scheduling. In fact, a task is separated into subtasks of unit executions in Pfair scheduling. Each subtask is therefore only valid to be executed within a certain period (also known as "window"). Figure 3.4 illustrates an example of periodic task scheduling with execution windows.

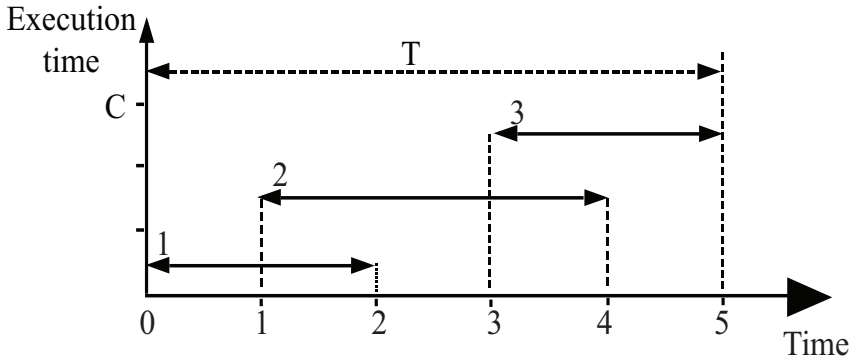

Figure 3.4: Pfair scheduling with "windows" of unit executions

The illustration includes a periodic task with weight of 3/5 and period of 5; the task's (worst-case) execution time is equal to 3 units of time. The task is separated into three subtasks each of which has unit execution. The execution windows of subtasks are shown by arrows with numbers above indicated subtask indexes. Accordingly, the first subtask

would be executed within the first two slots, the second one would be executed from time 1 to time 4, and the last one is valid for execution from time 3 to time 5. The start and finish times of windows can be derived from Pfair calculation using Equation 3.1 and Equation 3.2.

$$t_s(w_i^k) = \lfloor \frac{k-1}{\mu_i} \rfloor \tag{3.1}$$

$$t_f(w_i^k) = t_s(w_1^k) + \lceil \frac{k}{\mu_i} - 1 \rceil \tag{3.2}$$

where $t_s(w_i^k)$ and $t_f(w_i^k)$ are start time and finish time of the window of the kth ($1 \leq k \leq C_i$) subtask of job of task τ_i.

The difference of Pfair scheduling from the fluid scheduling is formally captured by the concept of *lag*. The *lag* of task τ_i at time t with respect to schedule S, denoted as $lag(S, \tau_i, t)$, is defined as:

$$lag(S, \tau_i, t) = \mu_i * t - S(\tau_i, t) \tag{3.3}$$

where $S(\tau_i, t)$ is the total time slots that task τ_i has been received during the period $[0, t)$. *lag* is considered as an allocation error associated with each task. Pfair scheduling is validated in a manner that allocation errors of tasks are guaranteed to be less than one over all times. That is,

$$\forall \tau_i, t :: -1 < lag(S, \tau_i, t) < 1 \tag{3.4}$$

Together with the concept of *lag*, additional definitions including characteristic string and characteristic substring are introduced in Pfair procedure. Accordingly, the characteristic string of task τ_i, denoted as $\alpha(\tau_i)$, is an infinite string of symbols $\{-, 0, +\}$ each of which at t, denoted as $\alpha_t(\tau_i)$, is defined as:

$$\alpha_t(\tau_i) = sign(\mu_i * (t+1) - \lfloor \mu_i * t \rfloor - 1) \tag{3.5}$$

The characteristic substring of task τ_i at time t is defined as the finite string:

$$\alpha(\tau_i, t) = \alpha_{t+1}(\tau_i)\alpha_{t+2}(\tau_i)...\alpha_{t'}(\tau_i) \tag{3.6}$$

where $t' > t$ is the minimum successive time so that $\alpha_{t'}(\tau_i) = 0$.

At every time t, Pfair procedure is executed to select tasks for execution based on the tasks' state classification described in Table 3.1. The selection consists of two steps: (1) Schedule all urgent tasks; (2) Allocate the remaining resources to the highest-priority contending tasks. The priority of the contending tasks are decided using a so-called *total order* $\succeq$ which is delineated as follows: $\tau_i \succeq \tau_j$ if and only if $\alpha(\tau_i, t) \geq \alpha(\tau_j, t)$ where the comparison between characteristic substrings $\alpha(\tau_i, t)$ and $\alpha(\tau_j, t)$ is resolved lexicographically with $- < 0 < +$.

Although achieving the optimality, Pfair is faced with several drawbacks. First, Pfair causes a large number of scheduler invocations. Second, tasks' executions appear to be fragmentary, which potentially increases task preemption and migration. Third, due to the heavy processes of task state classification and tie breaking of contending tasks, Pfair issues major runtime overhead repeatedly at every tick. Therefore, the accumulative runtime overhead is considerably high in Pfair scheduling. These drawbacks make Pfair difficult to be applied to practical scheduling situations.

Table 3.1: State classification of task τ in Pfair scheduling

State	Conditions
Ahead	$lag(S, \tau, t) < 0$
Behind	$lag(S, \tau, t) > 0$
Punctual	τ is neither ahead or behind
Tnegru	τ is ahead and $\alpha_t(\tau) \neq +$
Urgent	τ is behind and $\alpha_t(\tau) \neq -$
Contending	τ is neither tnegru or urgent

3.3 Proportionate scheduling on time interval

Pfair is a typical periodic scheduling algorithm for multiple shared resources, which guarantees the fairness of scheduling for tasks at any unit time. Due to the excessive scheduler invocations issued in this scheduling scheme, it is risen a question: How exactly does the fluid scheduling need to be approximated in practical use? The answer to this question is found in scheduling approaches of boundary fairness, largest local remaining execution time first, and DP-WRAP. The fairness of scheduling is indeed guaranteed for time intervals.

3.3.1 Boundary Fairness Scheduling

Boundary Fairness (BF) was introduced to relax the notion of fairness in Pfair. That is, it is unnecessary to assure the fairness for tasks at any system time. Instead, the fairness is guaranteed for tasks on time intervals. In BF scheduling, intervals are decided based on the set of system deadlines (task deadlines): two consecutive deadlines are forming an interval. Then, on each interval, tasks are receiving an appropriate amount of times as part of their execution time. In other words, a task is separated into subtasks corresponding to intervals. Subtasks of different tasks on an interval therefore have the common deadline at the end of the interval, which is known as deadline partitioning. Since intervals are mostly longer than a system tick, deadline partitioning scheme tends to reduce the number of scheduling points in comparison with Pfair scheduling. In addition, subtasks in BF scheduling can be allocated executions longer than unit time, which has potential to alleviate task preemptions and migrations.

The execution allocation on each interval can be briefly described as follows. Considered an interval $I = [t_1, t_2)$ with its length $L_I = t_2 - t_1$ where $t_1, t_2 \in N$ and $0 \leq t_1 < t_2$. BF algorithm is then processed in four steps:

1. Allocating mandatory units
   A task τ_i must be allocated mandatory units $m_i(t_1, t_2)$ which is calculated as:

$$m_i(t_1, t_2) = max(0, \lfloor RW_i^{t_1} + L_I * \mu_i \rfloor) \tag{3.7}$$

   where $RW_i^{t_1}$ is the remaining work of τ_i at time t_1. $RW_i^{t_1}$ is also known as the allocation error of τ_i respect to the fairness required at time t_1. $RW_i^{t_1}$ is considered

to be 0 at time $t_1 = 0$. $m_i(t_1, t_2)$ represents the integer part of the allocation during interval I that τ_i must be received in order to maintain the fairness at time t_2. The decimal part of the allocation is captured by a factor called pending work, denoted as $PW_i^{t_2}$. $PW_i^{t_2}$ is then obtained as:

$$PW_i^{t_2} = RW_i^{t_1} + L_I * \mu_i - m_i(t_1, t_2) \tag{3.8}$$

2. Allocating optional unit, if any
   The remaining time slots within interval I is distributed to tasks as the optional units. Namely, tasks are selected to receive one extra unit of time. Task τ_i is eligible for extra allocation if $PW_i^{t_2} > 0$ and $m_i(t_1, t_2) < L_I$.

3. Updating remaining works
   The remaining work of task τ_i at time t_2 is updated as:

$$RW_i^{t_2} = PW_i^{t_2} - o_i^{t_2} \tag{3.9}$$

   where $o_i^{t_2}$ is the optional amount of τ_i. The remaining work at t_2 is hence used for the future scheduling.

4. Generating schedule on the interval
   Finally, based on the allocated amounts on interval I, the schedule of tasks is generated. The task assignment and execution order on processors are decided using idea of McNaughton's algorithm [53].

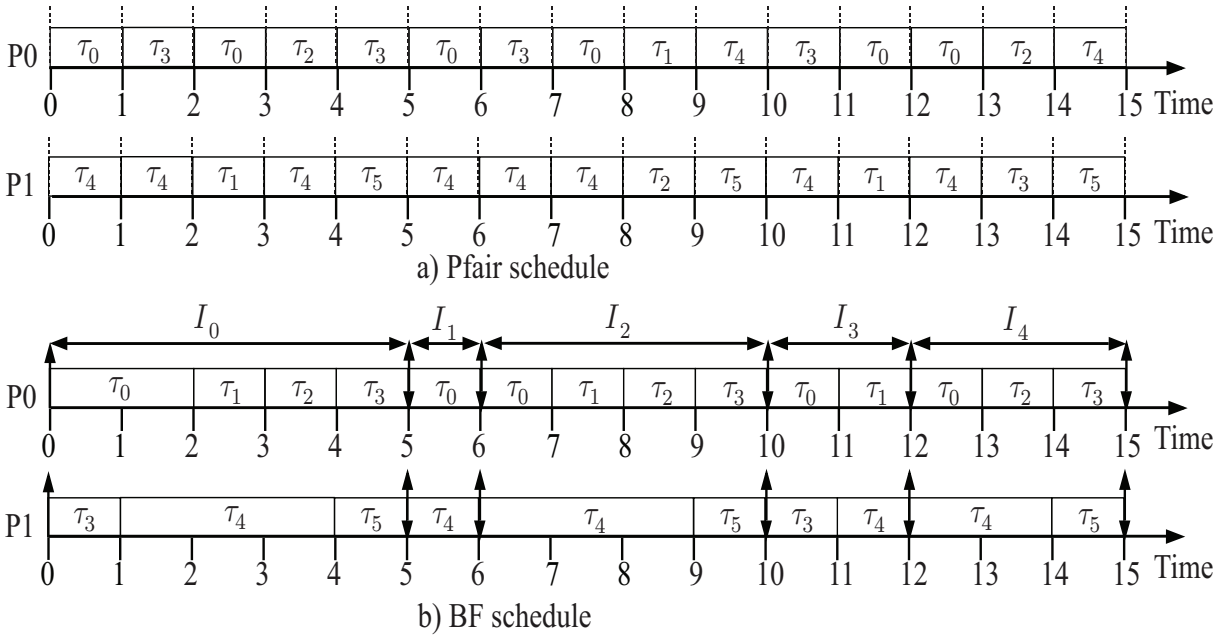

Figure 3.5: Example of BF scheduling vs Pfair scheduling

Figure 3.5 shows an example of BF scheduling for periodic tasks in parallel with Pfair scheduling. The example invokes a task set of six periodic tasks: $\tau_0 = (2/5, 5)$, $\tau_1 = (1/5, 15)$, $\tau_2 = (1/3, 6)$, $\tau_3 = (1/3, 6)$, $\tau_4 = (2/5, 15)$, and $\tau_5 = (1/5, 15)$. In Figure

3.5(a), dashed lines at every tick indicate scheduling points of Pfair scheduling. As a result, it is required 15 times of scheduler invocations to schedule tasks during the first 15 slots. Figure 3.5(b) shows the schedule for BF scheduling. Upward arrows indicate the time at which job of task is released and double-headed arrows indicate the time at which job release and job deadline concurrently occur. Based on the job release at time 0 and system deadlines at 5, 6, 10, 12 and 15, five intervals are identified for BF during the first 15 slots. BF therefore causes only five scheduler invocations to schedule tasks. Obviously, BF is better than Pfair in terms of scheduling invocation. In addition, the numbers of preemptions and migrations are also fewer in BF schedule since tasks have more chance to receive seamless execution during an interval; for example τ_4 is executed seamlessly for three time slots in the first interval.

However, BF has a disadvantage of unnecessary idle times of processors. This is caused by calculation of allocated resources of tasks on the interval based on mandatory execution and optional unit execution. In other words, adding only the unit execution is not enough to exploit the entire system capacity on the interval. For example, consider two periodic tasks $\tau_0 = (2/3, 3)$ and $\tau_1 = (2/3, 6)$ scheduled on two processors. The first interval therefore is identified for $[0, 3)$. The calculation of mandatory units (using Equation 3.7) gives tasks two time slots each. Since using Equation 3.8 the pending works for two tasks are zero, no optional unit is added to tasks. The allocated amount of τ_1 is therefore two time slots on the interval although actually it is possible to assign three time slots to τ_1. One processor is put in idle time unnecessarily at slot 2 as a result. Occurrence of unnecessary idle times reduces the performance of BF scheduling.

3.3.2 Largest Local Remaining Execution Time First

Largest Local Remaining Execution Time First (LLREF) is another optimal algorithm introduced to the problem of periodic scheduling on multiple shared resources. LLREF is also explored the fluid scheduling and fairness notion to pursue the optimality. In LLREF, an effective abstraction called Time and Local Execution Time Domain Plane (T-L plane) is utilized to model task execution behaviors. The scheduling algorithm is then formed based on such modeled task behaviors. Similar to BF scheduling, LLREF is targeted to guarantee the fairness of scheduling for time intervals during which T-L planes are established. The concept of interval is defined in LLREF as time periods between any two consecutive job releases. The definition of time intervals is therefore convertible with that of BF scheduling where system deadlines are used. This is because both algorithms are targeted to periodic tasks; except the first job release, deadline of a job of task is also the release time of the next job.

Figure 3.6 illustrates the establishment of T-L planes. It is supposed to have n periodic tasks τ_0, τ_1, ..., τ_{n-1} existing in the systems. C_0, C_1, ..., C_{n-1} are tasks' (worst-case) execution times, respectively. Double-headed arrows marks time points that are release times (and also deadlines) of jobs of the tasks. Let us consider such time points t_1, t_2, and t_3 and identified intervals $[t_1, t_2)$ and $[t_2, t_3)$. Dashed diagonal lines present the fluid schedule of tasks on their corresponding periods. Based on the task fluid schedules and interval's ends, a right isosceles triangle can be found for each task on the interval. As a result, n such triangles are found on each interval corresponding to n tasks. A T-L plane of an interval is then constructed by overlapping these n triangles; $T - L^k$ and $T - L^{k+1}$ are for example shown in the figure for the two intervals $[t_1, t_2)$ and $[t_2, t_3)$. The

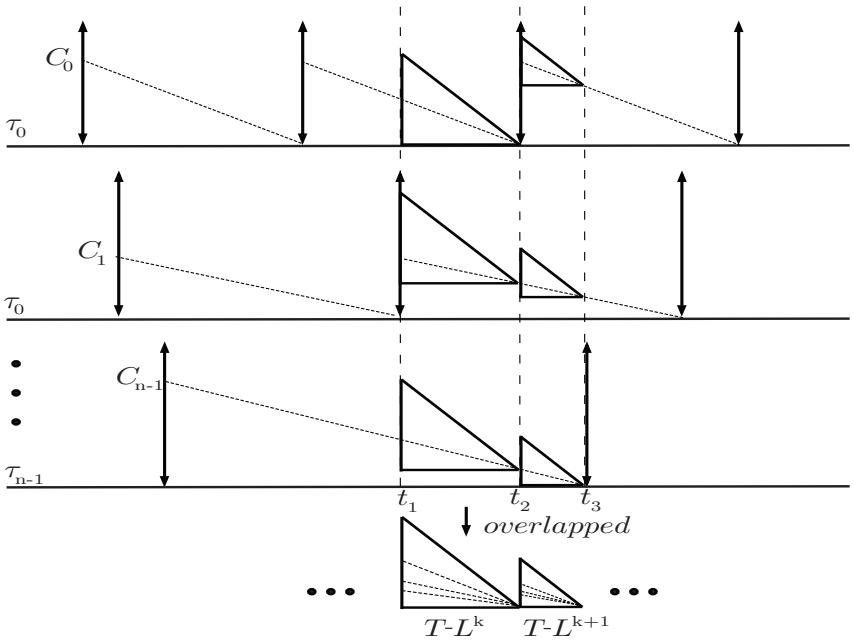

Figure 3.6: Establishment of T-L plane in LLREF

bottom side of T-L planes represents the time while the left vertical side indicates a part of execution of the tasks which is called the task's local remaining execution time and appears to be consumed during the interval. The schedules of tasks during intervals are decided by modeled task behaviors on T-L planes.

Figure 3.7 shows the model of task behaviors on a T-L plane for interval $[t_c, t_f)$. In the T-L plane, tasks are modeled as tokens the starting location of which on the vertical side is equal to their local remaining execution time (l_0, l_1, ...,l_{n-1}). The dashed diagonal paths again are the ideal moving paths of tasks following the fluid schedule. However, the moving of tasks practically deviates from these paths. The model of moving behaviors of task is described as follows:

- When task is selected for execution, the corresponding token is moved diagonally down like τ_0 moving.
- When task is not selected for execution, the corresponding token is moved horizontally like τ_{n-1} moving.
- When a token hits the bottom side, the represented task has no local remaining execution time on the interval. The task should not be scheduled any more and subjects to move horizontally to the end of the interval (t_f).
- When a token hits the ceiling of local laxity, the represented task has no local laxity on the interval. The task must be scheduled seamlessly for the remaining time of the interval.

The target of scheduling on T-L planes is to guarantee for all tasks to arrive at t_f with no local remaining execution time. Respect to tasks' moving behaviors, context switches incur in necessity at events of bottom hit and ceiling hit to guarantee tasks' executions.

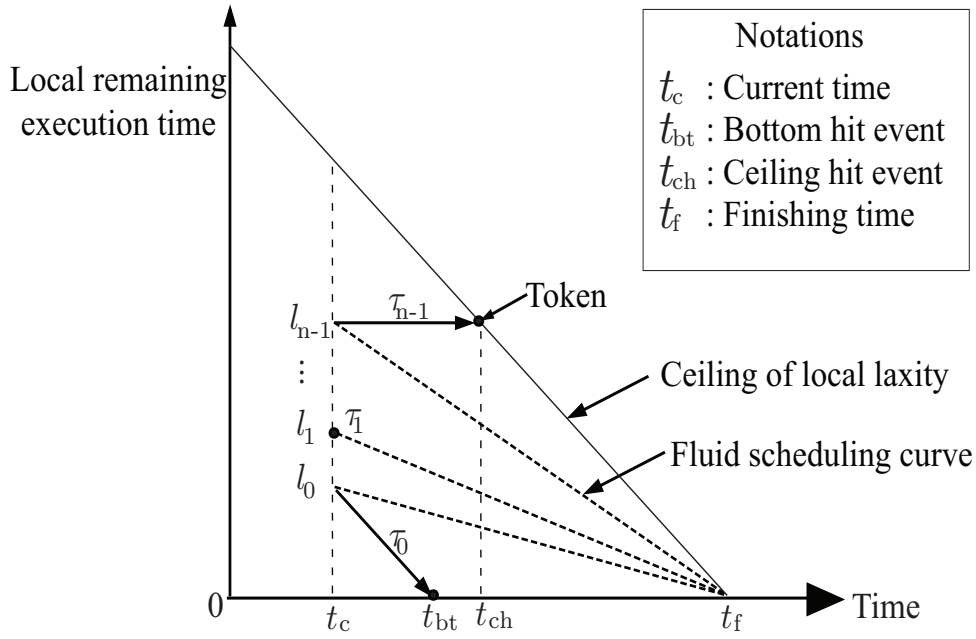

Figure 3.7: Scheduling within an T-L Plane

Therefore, bottom hits and ceiling hits are considered as scheduling points during the T-L planes besides the evident scheduling point at the starting times of the T-L planes. LLREF algorithm for scheduling n periodic tasks on m identical shared resources can be summarized as follows. At every scheduling points, the scheduler will do:

1. Sorting tasks' tokens in the descending order of local remaining execution times.
2. Selecting m tasks of the highest local remaining execution times.
3. Return m selected tasks and assign to processors for execution.

It is obviously that LLREF scheduling is worst than BF scheduling in terms of number of scheduling points. Specifically, the number of intervals formed in the two algorithms is essentially the same. However, due to the scheduler invocations additionally required at bottom hits and ceiling hits, the number of scheduler invocation of LLREF scheduling always larger than that of BF scheduling. In addition, LLREF is also suffered with the increase of task preemption and migration.

3.3.3 DP-WRAP

In the background of real-time task scheduling on multiple identical shared resources, many scheduling algorithms such as BF, LLREF were introduced based on the fluid schedule and deadline partitioning. DP-WRAP algorithm was emerged to show a unifying theory among algorithms.

DP-WRAP introduces DP-FAIR scheduling policies to pursue the optimality. Especially, DP-WRAP also shows an extension of DP-FAIR to schedule sporadic task sets with arbitrary deadlines. The algorithm is described with two basic definitions: time slice and task's density. Time slice is defined as the time period between two consecutive system

deadlines. Specifically, the jth time slice, denoted as σ_j, is the time period $[t_{j-1}, t_j)$ where $j \in N$ and t_{j-1} and t_j are two consecutive deadlines of tasks. The time slice is then have length of $L_j = t_j - t_{j-1}$. This concept of time slice is therefore similar to the concept of time interval in BF and LLREF. Task's density , denoted as δ_i, is defined in Equation 3.10 where C_i, T_i, and D_i are the (worst-case) execution time, period and relative deadline of the task, respectively.

$$\delta_i = \frac{C_i}{min(T_i, D_i)} \tag{3.10}$$

In consideration of periodic tasks period and relative deadline of which are the same, δ_i is known as tasks' utilization.

Then, DP-WRAP algorithm can be simply described as follows. In time slice σ_j, each task τ_i is allocated a segment of length δ_i; there are totally n segments created corresponding to n tasks. Such segments are then concatenated into one line the length of which is not longer than m since $\sum_{i=0}^{n} \delta_i \leq m$ for feasible sets. Figure 3.8 is an illustration of the line of six tasks with densities of 0.6, 0.6, 0.4, 0.8, and 0.2, scheduled on a system of three processors. Note that tasks' segments can be arranged in any order in the line. The schedules of tasks on processors are obtained by dividing the line into m chunks of unit length and assign chunks to the corresponding processors. The order of tasks on a chunk indicates the execution order of tasks on the respective processor. Tasks which are split to two processors (τ_1 and τ_3 in this example) are the migratory tasks during the time slice. The actual execution time on processors and the context switching times are found by multiplying each segment's length by length L_j of the time slice. The scheme of assigning tasks to processors by chunks is actually inspired of the McNaughton's algorithm [53].

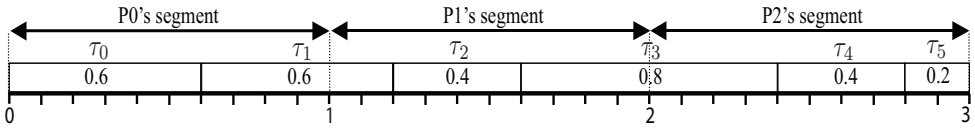

Figure 3.8: DP-WRAP scheduling

DP-WRAP research completely generalizes the theory of guaranteeing fairness for periodic schedules on time intervals. Especially, this is also applicable to problem of scheduling sporadic tasks. However, since multiplying task density by time slice's length often results in decimal values, determining context switches at runtime is very complicated.

3.4 Reduction to Uniprocessor

Recently, RUN (Reduction to UNiprocessor) [38] has been realized as the outperforming optimal algorithm for multiprocessor scheduling in terms of task preemptions and migrations. Therefore, RUN becomes the appropriate competitive candidate when one introduces a new scheduling algorithm. In the rest of this section, we briefly review this scheduling algorithm as a comparison candidate for the evaluation of this dissertation.

The key principle of RUN is an off-line transformation of the multiprocessor system to an uniprocessor one which is used to schedule tasks at runtime. The transformation is conducted based on two operations: packing operation and dual operation. In order to

present the two operations, RUN introduces concept of server which is a virtual task representative for a set of tasks or servers. A server has utilization equal to the total utilization of its client tasks (servers) and deadlines as union of its client tasks' (servers')deadline. (Readers can refer [38] for the detailed definition of server.) Packing operation accordingly is a partition of a task (server) set into subsets so that the accumulative utilization of every single subset is less than or equal to 1 and the summation utilization of any two subsets are greater than 1. As a result, the number of elements in the output server set of the packing operation is less than then number of elements in the input task (server) set. Figure 3.9 shows an example of the transformation of RUN. At the bottom of the figure is the set of four tasks with the utilizations of 0.6, 0.6, 0.6, and 0.2, respectively. As shown in the figure, the first packing operation reduces the task set to a set of three servers S_0, S_1, and S_2. S_0 and S_1 have the same utilization as their original task and S_2 has utilization of 0.8 as summation of its two original tasks.

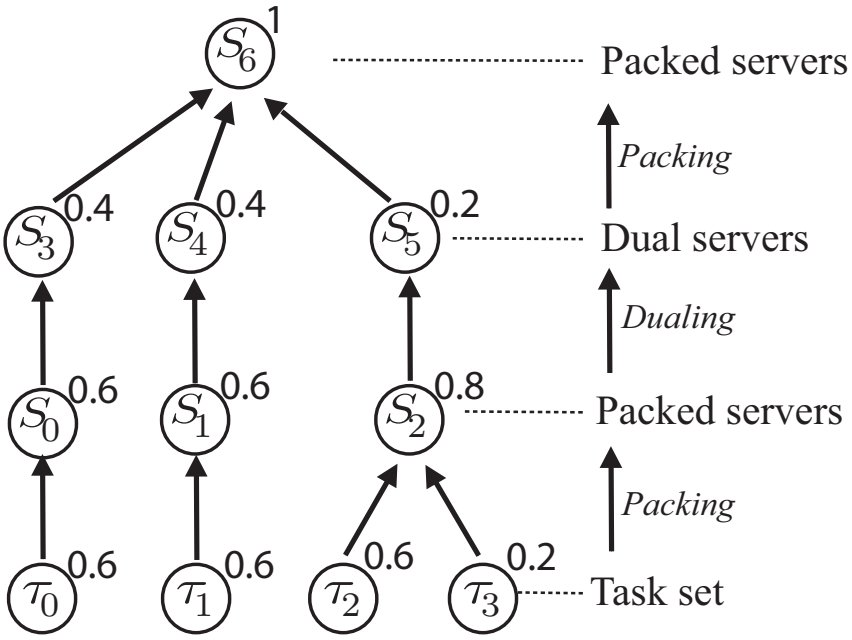

Figure 3.9: RUN: reduction tree

On the other hand, dual operation is a generation of a dual server which has the same period with the original server and complementary utilization of the original server. In other words, the dual server of a server $S = (\mu_s, T_s)$ is $S^* = (1 - \mu_s, T)$. For example, the dual operation applied to servers S_0, S_1, and S_2 in Figure 3.9 generates three servers S_3, S_4, and S_5 with the utilizations of 0.4, 0.4, and 0.2.

By alternatively conducting the packing operation and dual operation, the problem of multiple processors is reduced to one processor [38]. Each packing operation corresponds to a reduction level. This process of transformation generates a so-called reduction tree in RUN. The root node of a reduction tree is always one server with utilization of 1. The child nodes are either servers (packed or dual servers) or tasks (at the bottom level). Branches in a reduction tree therefore indicate packing and dual operations. Note that for a problem of tasks, RUN transformations can generate many reduction trees each of them is corresponding to scheduling of a subset of entire task set on a group of processors.

A reduction tree of a subset of utilization of U_Γ will invoke $\lceil U_\Gamma \rceil$ processors.

Using the reduction tree, the schedule of tasks at runtime is decided by tracing servers from the root to the bottom leaves (the original tasks). The selection of tasks are provided with the following two basic rules:

- If a node of packed server is selected, then among its child nodes the one which has the earliest deadline and still have execution remaining is selected.
- If a node of dual server is selected, then its child node is unselected.

Since the server at the root has the utilization of 1, it is always selected at scheduling time. The tracing process starts at the root of each reduction tree and ends when the bottom leaves (the original tasks) are reached. The final tasks are selected for execution on the invoked processors.

Besides effectiveness on task preemption and migration, RUN efficiently reduces run-time overhead with off-line process of the reduction transformation. However, when the number of tasks and processors increase, the number of servers dramatically increases and many reduction trees are required for scheduling. The on-line tracing on all of reduction trees and updating status of servers (remaining execution times, deadlines) potentially cause considerable overhead.

Chapter 4

Local Assignment Algorithm for Periodic Task Scheduling

4.1 Introduction of Local Assignment Algorithm

In Chapter 3, we have reviewed several scheduling algorithms which achieve the optimality by approximating the fluid schedule. These approaches have their own advantages and disadvantages:

- Pfair early proposes the notion of proportionate scheduling to guarantee the fairness for periodic tasks. However, time-quanta scheduling makes Pfair ineffective due to increase of scheduler invocation and runtime overhead.
- Although remarkably reducing the number of scheduler invocations and task preemption with time-interval scheduling, BF has a drawback in exploitation of system capacity due to unneeded occurrence of idle times of processors.
- Compared with BF, LLREF requires scheduler invocation more frequently. Besides, increase of task preemption and migration is also worth considering.
- DP-WRAP completes the unifying theory and suggests a simple model for proportionate scheduling on time intervals. Nevertheless, this simple model requires complicated calculation to determine context switches at runtime.

In this chapter, an effective approach, named Local Assignment Algorithm (LAA), is proposed for the problem of periodic schedule on multiprocessor systems. The proposed algorithm is motivated by taking over advantages and overcoming disadvantages of the existing algorithms mentioned above. In detail, LAA is solving the following issues:

- Employing the notion of proportionate scheduling on time interval (similar to BF and DP-WRAP schedulings) in order to pursue the optimality with less scheduler invocations.
- Utilizing the system capacity effectively. It is expected that the system capacity is fully utilized on every interval. To this end, full assignment scheme is introduced. Approach of full assignment ultimately reduces unnecessary idle times of processors.

- Distributing tasks to processors in a better manner to improve task preemption and migration. McNaughton's algorithm and consecutive assignment model of DP-WRAP are exploited with inclusion of task selection process. In the proposed assignment scheme, tasks are assigned to processors selectively using schedule history and the remaining capacity of processors.

The remaining of this chapter is structured in six sections. Section 4.2 describes definitions which are basic mathematical calculations of LAA. Section 4.3 is details of the procedure of LAA. For a complete understanding of the proposed algorithm, an example of scheduling with LAA is given in Section 4.4. The proof of optimality comes in Section 4.5. Section 4.6 is evaluation of the proposed algorithm in comparison with the existing algorithms. Finally in Section 4.7 is summary of the effectiveness of LAA as well as discussion on its limitation.

4.2 Definition of LAA

Before elaborating the procedure of LAA, it is important to make basic definitions involved in the algorithm clear. Definitions include time interval, notion of proportionate scheduling, local requested execution time, fully-assigned system, and scheduling plan.

4.2.1 Time interval

Since LAA is proposed in the class of time-interval scheduling, definition of time interval therefore needs to be clarified first.

The time interval of LAA is defined as follows:

Definition 4.2.1. *Time interval (or shortly interval) is the time period between any two consecutive job releases.*

Specifically, let t_i and t_j denote two consecutive job releases where $i, j \in N$ and $0 \leq t_i < t_j$. t_i and t_j are forming an interval $I = [t_i, t_j)$ so that t_i is included in and t_j is excluded from the interval. The length of interval, denoted as $|I|$, is then given by $|I| = t_j - t_i$.

Figure 4.1 depicts intervals determined by release times of five periodic tasks. Tasks are supposed to have period of 3, 9, 12, 4, and 6, respectively. In this figure, upward arrow at time 0 indicates time of job releases and double-headed arrows indicate time of concurrently job release and deadline. Materially for this task set, jobs are released at times 0, 3, 4, 6, 8, 9, and 12 during the first twelve time slots. Therefore, six intervals are formed as I_0, I_1, I_2, I_3, I_4, and I_5.

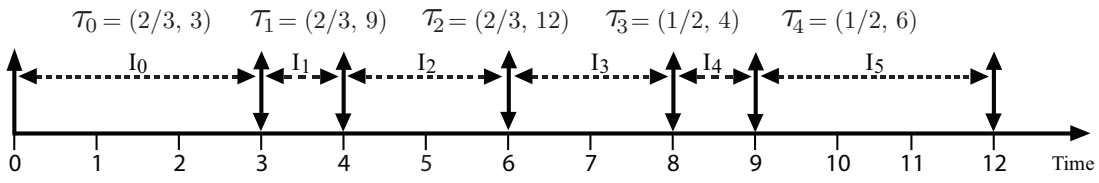

Figure 4.1: Identification of intervals

Each interval is further associated with a factor called **system capacity**. The system capacity is the total time slots over all processors that are allocated to tasks for execution during the interval. Accordingly, an interval I with the length of $|I|$ would have a capacity equal to $m|I|$ where m is the number of processors available.

4.2.2 Proportionate scheduling

The notion of fairness of Pfair is employed to pursue the optimality of LAA. Reminding that the proportionate scheduling of Pfair guarantees that at every time t, a task τ_i must have been scheduled to receive either $\lfloor \mu_i * t \rfloor$ or $\lceil \mu_i * t \rceil$ resources where μ_i is task utilization (weight).

The proportionate scheduling is re-defined in LAA as follows:

Definition 4.2.2. *Consider t as the end time of an interval. A task τ_i must receive at least $\lfloor \mu_i * t \rfloor$ resources by time t.*

The difference is that the flooring term $\lfloor \mu_i * t \rfloor$ only is required in LAA algorithm. This simplification is to reduce the time complexity of the algorithm. Let $S_i(t)$ denote the number of time slots (resources) that τ_i is entitled to receive by time t based on Definition 4.2.2. $S_i(t)$ is calculated as follows:

$$S_i(t) = \lfloor \mu_i * t \rfloor \tag{4.1}$$

Additionally, we can obtain the sum of the total resources requested by all existing tasks up to t, denoted as $S(t)$, using Equation 4.2.

$$S(t) = \sum_{i=0}^{n_p - 1} S_i(t) \tag{4.2}$$

Let $A_i(t)$ denote the actual resources that τ_i has been already assigned by time t (or during period from time 0 to time t). This value is easily recorded by the operating system. The total already assigned resources of all tasks up to t, denoted as $A(t)$, can be obtained using Equation 4.3.

$$A(t) = \sum_{i=0}^{n-1} A_i(t) \tag{4.3}$$

4.2.3 Local requested execution time

Definition 4.2.3 shows a definition of local requested execution time in LAA.

Definition 4.2.3. *Local requested execution time (LRET) is the amount of times for each task to be executed during an interval.*

Since a job of task can be involved in several intervals, LRET is a part of the task's execution. The term "local" here means within one interval; that is LRET of a task is a part of execution that is accommodated within one interval.

The calculation of LRET of task τ_i on interval $I = [t_1, t_2)$, denoted as $E_i(t_1, t_2)$, is shown in Figure 4.2. As shown, $E_i(t_1, t_2)$ consists of two parts: mandatory execution

$M_i(t_1, t_2)$ and extra execution $P_i(t_1, t_2)$. These parts are calculated using Equation 4.4 and Equation 4.5.

$$M_i(t_1, t_2) = \lfloor \mu_i * t_2 \rfloor - A_i(t_1) \tag{4.4}$$

$$P_i(t_1, t_2) = min(JRE_i, slack, L_I - M_i) \tag{4.5}$$

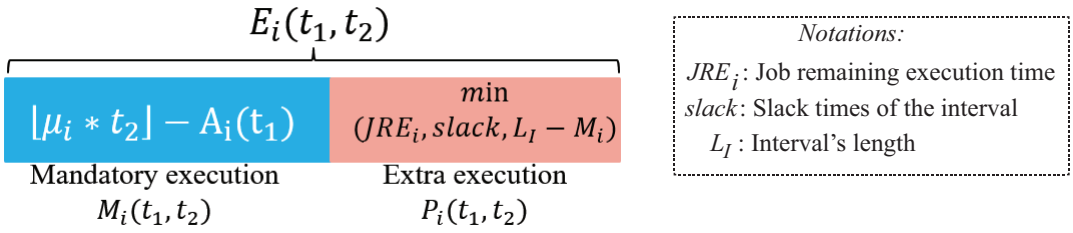

Figure 4.2: Local requested execution time of LAA

Then, the total requested execution time of tasks on interval I, denoted as $E(t_1, t_2)$, is calculated in sum as:

$$E(t_1, t_2) = \sum_{i=0}^{n_p - 1} E_i(t_1, t_2) \tag{4.6}$$

4.2.4 Fully-assigned system

Definition 4.2.3 shows a definition of fully-assigned system in LAA.

Definition 4.2.4. *A multiprocessor system is called a fully-assigned system by time t if no slack time (empty slot) occurs over processors up to t.*

The definition implies that if the system is fully assigned up to time t, all of time slots are occupied by task execution up to t. As a result, the total already assigned resources of tasks up to t (or $A(t)$ as denoted above) on fully-assigned system of m processors can be obtained using Equation 4.7.

$$A(t) = \sum_{i=0}^{n_p - 1} A_i(t_1) = m * t \tag{4.7}$$

4.2.5 Scheduling plan

Definition 4.2.5. *Scheduling plans are arrangement of task executions on processors during intervals. A scheduling plan is therefore solving the following two things:*

- *The amount of times allocated to each task on each processor and*
- *The execution order of tasks with allocated amounts on each processor.*

Scheduling plans are used as instruction to select tasks for execution and determine context switches at runtime. The scheduler is then invoked at the start time of every interval to make scheduling plans. The start times of intervals are considered as **scheduling events** in LAA. Notably, since periodic tasks are targeted for scheduling, the end time of the current interval is the start time of the next one and therefore also a scheduling event in LAA.

4.3 Procedure of LAA

Local Assignment Algorithm (LAA) is introduced to make scheduling plans for scheduling tasks on intervals. In respect of the definition of scheduling plan, execution of the algorithm needs to provide the following two tasks:

1. Calculating local requested execution times as the amount of times allocated to tasks on the interval and

2. Assigning tasks to processors to make the execution order of tasks on processors.

In addition, toward the goal of achieving the full schedulability of 100%, LAA is designed to guarantee the system fully-assigned on every interval. **Algorithm 1** shows the pseudo procedure of LAA. This procedure is executed by scheduler at the beginning of each interval in oder to make the scheduling plan. As shown in the procedure, besides the task set (indexed by i), the execution of LAA requires two input data: t_s as the start time and t_e as the end time of the involved interval. These information can be obtained easily since they are release times of jobs of periodic tasks and can be solved statistically. The length of the involved interval, L_I, is by definition equal to $t_e - t_s$. Then, the procedure of LAA consists of three phases: the first two phases are to calculate LRETs for tasks and the third phase is to assign tasks to processors. The targeted scheduling plan for the involved interval is therefore obtained at the end of the third phase. Specifically, in the first phase, *LRET Estimation*, the mandatory part of LRETs for each tasks is calculated using combination of Equation 4.1 and Equation 4.4 where t_1 and t_2 are correspondingly replaced with t_s and t_e.

Since the flooring term $\lfloor \mu_i * t_e \rfloor$ is applied, the total amount of LRETs of tasks from this step is less than or equal to the system capacity on the interval. In other words, the fully-assigned system has not been guaranteed with these mandatory amounts and an adjustment for LRETs is required. This is the work designated to the second phase, *LRET Adjustment*. The adjustment process is distributing the remaining slack times of the interval to tasks in harmony with the tasks' remaining execution time and interval length. To this end, a variable *slack* is computed by subtracting the total mandatory amount, calculated in the first step, from the system capacity of the interval ($m * L_I$). Then, tasks are each judged to receive an extra execution equal to the minimum one among their current job's remaining execution time (JRE_i), the remaining slack (*slack*), and the possible increment ($L_I - E_i(t_s, t_e)$). The term $L_I - E_i(t_s, t_e)$ indicates that the local requested execution time of a task on the interval is not larger than the interval length. In other words, the local requested execution time of any task is completely accommodated within the interval. When task τ_i is received an extra amount, JRE_i and *slack* are subtracted by the amount.

The final phase, *Task Assignment*, is to assign tasks with their calculated LRET to processors appropriately. We introduce a consecutive assignment scheme for this purpose. Basically, the assignment idea is that the schedule of tasks on actual processors is extracted from assignment of tasks on an virtual uniprocessor which is established by concatenating actual processors together.

As described in Algorithm 1, process of consecutive assignment requires two input data: $E(t_1, t_2)$ as the set of LRETs of tasks (calculated in the previous two phases) and $Record[][]$ as the record of schedule history of tasks on processors at every slot. In detail,

an access to $Record[1][5]$ returns the identifier of the task that was executed at slot 5 on processor 1. The output of the process is therefore the targeted scheduling plan for the involved interval. The process is then described in four steps. For ease of understanding, the process is elaborated with illustration in Figure 4.3. In this illustration, the problem is supposed to make scheduling plan of five tasks on three processors for interval $I = [t_1, t_2)$. LRETs of tasks are $E(t_1, t_2) = \{2, 3, 3, 3, 4\}$ and slot $t_1 - 1$ on the three processors are scheduled for τ_1, τ_3, and τ_4, respectively.

Algorithm 1: Local Assignment Algorithm

$\tau = \{\tau_i, 0 \leq i < n_p\}$ *: task set indexed by* i
t_s, t_e *: the start and end times of the involved interval*
$L_I \leftarrow t_e - t_s$

1. LRET Estimation:
   For each task τ_i:
   $E_i(t_s, t_e) \leftarrow \lfloor \mu_i * t_e \rfloor - A_i(t_s)$
2. LRET Adjustment:
   $slack \leftarrow m * L_I - \sum_{i=0}^{n-1} E_i(t_s, t_e)$
   $i \leftarrow 0$
   while $slack > 0$ **do**
   {
   $min \leftarrow min(JRE_i, slack, L_I - E_i(t_s, t_e))$
   $E_i(t_s, t_e) \leftarrow E_i(t_s, t_e) + min$
   $slack \leftarrow slack - min$
   $JRE_i \leftarrow JRE_i - min$
   $i \leftarrow i + 1$
   }
3. Task Assignment: consecutive assignment process
   Inputs: $E(t_1, t_2) = \{E_i(t_s, t_e), 0 \leq i \leq n_p - 1\}$, $Record[][]$
   Outputs: Schedule plan SP
   Procedure:
   (1) Establish a virtual uniprocessor UP
   (2) Assign tasks just executed at slot $t_s - 1$ *to the same processor's segment:*
   For each segment p *of* UP*:*
   $p \leftarrow E_i(t_s, t_e) : 0 \leq p < m$ & $Record[p][t_s - 1] = i$
   (3) Assign the other tasks to the emptiest space on UP
   (4) Convert order of tasks on UP *to schedule plan* SP

At the first step, virtual uniprocessor is established by joining m actual processors altogether in series. Each actual processor is therefore corresponding to a segment of the virtual uniprocessor. The capacity of the virtual uniprocessor is equal to the total capacity of actual processors; that is $m * L_I$.

At the second step, tasks are selectively assigned to processors' segments on the virtual uniprocessor based on the schedule history. Concretely, by indexing slots from 0, the schedule at slot $t_1 - 1$, the slot just before the scheduling time, is used for this task selection. Then, the task executed at slot $t_1 - 1$ will be assigned to the segment of the same processor if its LRET is larger than zero. Accordingly, in this illustration, τ_1, τ_3 and τ_4 are assigned first to the three processors' segments, respectively. Tasks are allocated amounts of time equal to their LRETs.

At the third step, the remaining tasks are seamlessly placed at the emptiest space of the virtual uniprocessor. This action may cause the just-assigned tasks to move along the processor's segments to give a fit space for the coming task. Obviously, the capacity of the virtual uniprocessor is sufficient to satisfy all tasks' LRETs. In this illustration, τ_0 is placed after τ_1 on P0's segment and τ_2 is then placed after τ_3 ans belongs to two segments. τ_4 is the moving one to give fit space for τ_2.

At the final step, the assignment of tasks on the virtual uniprocessor is converted to schedule of tasks on virtual processors in correspond to processors' segments. Specifically, keeping the order of task executions on the first segment makes the scheduling plan on the first actual processor. The second and third segments are respectively mapped to the second and third processors in the same manner. The expected scheduling plan is obtained.

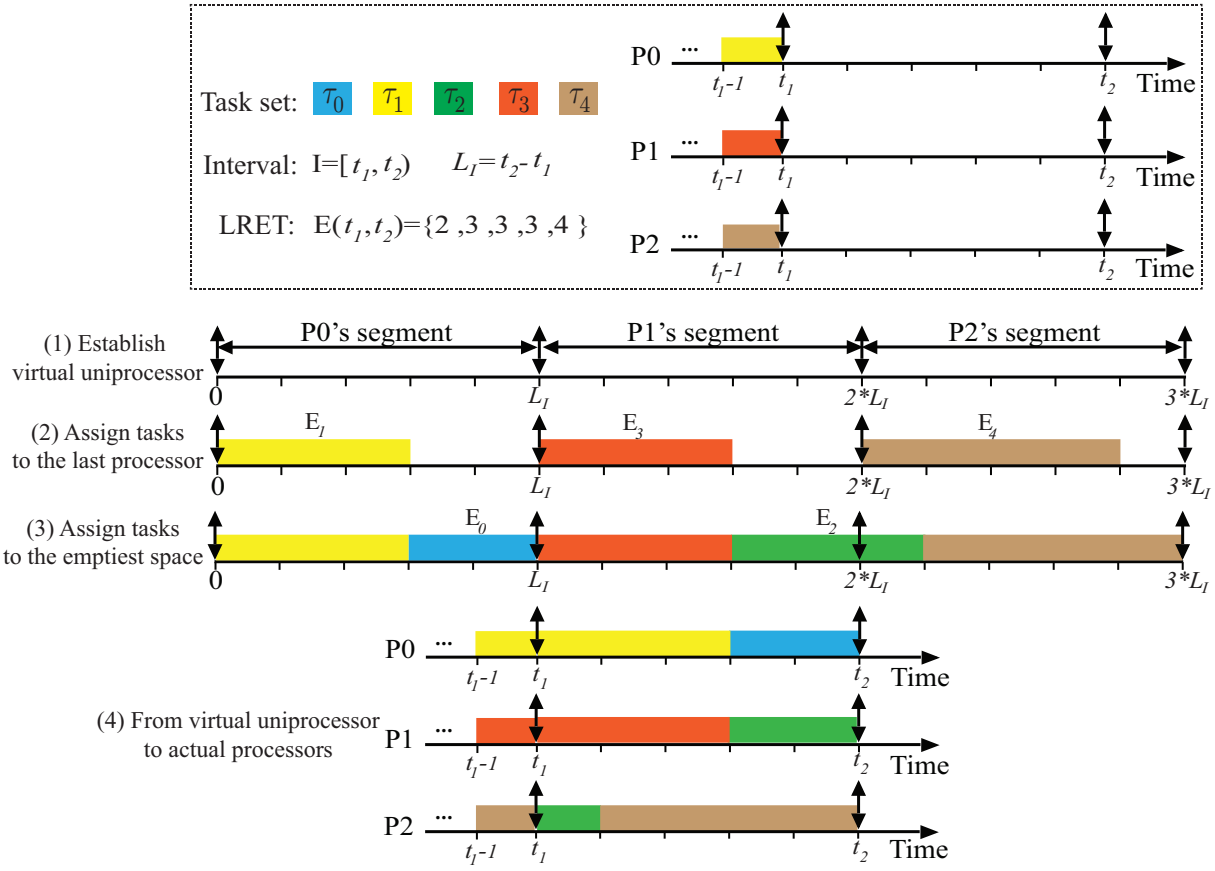

Figure 4.3: Process of consecutive assignment

It is notable that the consecutive assignment strategy guarantees two important scheduling principles:

1. There exists no concurrent execution of a task on more than one processor at the same time.

2. All LRETs of tasks are accommodated by the end of the involved interval;

The calculations of LAA guarantee that LRETs of tasks on an interval are less than or equal to the interval's length. Observing the allocation scheme of the consecutive

assignment (Figure 4.3), there exist two allocation situations happening for LRET of a task: (1) allocated on one processor's segment and (2) belong to two adjacent processors' segments. In the first situation, after the schedule converting process, the task is executed on one processor during the interval. In the second situation, the LRET of the task is separated into two sub-executions which are executed on two processors. A sub-execution is therefore executed at the start of one processor and the other sub-execution is executed at the end of the other processor. Since the LRET of a task on an interval is not larger than the interval's length, the two sub-executions are not overlapping each other. In other words, no concurrent execution of a task occurs on more than one processor at the same time.

The latter principle is easily assured with the cumulative capacity of the virtual uniprocessor. That is, the capacity of the virtual uniprocessor is equal to the total capacity of actual processors on the interval. Since for feasible systems the total LRETs of tasks are less than or equal to the system capacity on the interval, the cumulative capacity of the virtual uniprocessor is large enough to accommodate LRETs of tasks.

4.4 Example of scheduling with LAA

In Figure 4.4 we show a simple example of task scheduling with LAA. The example is to deeper clarify steps of making scheduling plans for serial intervals. The problem takes into account of five periodic tasks with identical utilization of 0.6 and periods of 5, 10, 15, 10, 5, respectively. Tasks are scheduled to execute on three processors. Supposed that all tasks are first releasing jobs at time 0. Jobs will be continuously released at time 5, 10, 15, ..., and so on. Based on the job releases, two intervals I_0 and I_1 are forming during the first ten slots. The scheduler is invoked at time 0 and time 5 to make scheduling plans for the two intervals.

Applying Algorithm 1 for I_0, in the first phase of LRET Estimation, the mandatory parts of LRETs are calculated for tasks on interval I_0 (with $t_s = 0$ and $t_e = 5$), which gives three slots for each task. In the second phase of LRET Adjustment, since the total amount of mandatory parts fulfills the system capacity of I_1 (15 slots in total), tasks receive no extra execution. The LRETs of tasks on I_0 are eventually equal to 3, as express by E_0. E_0 is used for allocation of tasks to processors in the last phase of Task Assignment.

In the last phase, a virtual uniprocessor is established for I_0 with the capacity of 15 (as capacity of three processors over five slots) as the result of the first step of the consecutive assignment process. Since at time 0, no schedule history is recorded, the second step is passed with no task assigned on the virtual uniprocessor. At the third step, assigning tasks to the emptiest space leads tasks in order of τ_0, τ_1, τ_2, τ_3, and τ_4 on the virtual uniprocessor alike task indexes. Each task is allocated three slots on the virtual uniprocessor. At the last step of the assignment process, order of tasks on the virtual uniprocessor is converted to the scheduling plans on actual processors. In detail, τ_0 is allocated three slots on $P0$ followed by two slots of τ_1. τ_1 is allocated one first slot of $P1$ followed by three slots of τ_2 and one slot of τ_3. Finally, $P2$ will start with two slots of τ_3 followed by three slots of τ_4. The whole scheduling plan for I_0 is found in the figure.

Similarly, the scheduler will be invoked at time 5 to make scheduling plan for I_1. Applying Algorithm 1, LRETs for tasks on I_1 are calculated, which eventually results to give three slots for each task as shown in E_1. The consecutive assignment process is then

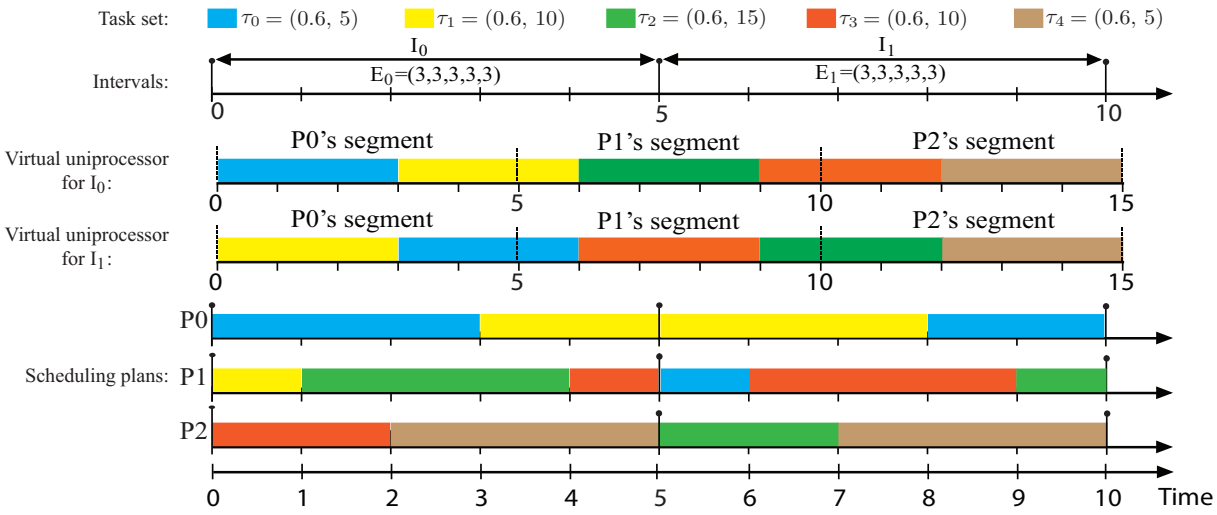

Figure 4.4: Example of making scheduling plans for intervals

conducted for I_1 with the schedule history recorded for slot 4 (the slot just before time 5). According to the scheduling plan of I_0, τ_1, τ_3, and τ_4 were executed on $P0$, $P1$, and $P2$, respectively, at slot 4. The consecutive assignment process for I_1 is conducted as follows. As the first step, the virtual uniprocessor is established. Then, based on the schedule history at slot 4, τ_1, τ_3, and τ_4 are selected to be assigned to the head of three processor's segments as the tasks last executed on these processors. The task selection continues to select τ_0 and allocate this task at the space after τ_1. τ_0 will take the two remaining two slots of the first segment and one slot at the head of the second segment. As a result, τ_3 has to move along the virtual uniprocessor to give fit space for τ_0. Repeating the task selection process, τ_2 is selected and assigned to empty space after τ_3. In this case, τ_4 is the moved one to give fit spaces for τ_2. The order of tasks on the virtual uniprocessor of I_1 and the final scheduling plan for I_1 are shown in the figure.

4.5 Schedulability guarantee of LAA

This section gives the proof of achieving the optimality for scheduling system as a whole. For the optimality (as defined in Chapter 2), we prove that LAA is capable of scheduling any feasible task set at the total task utilization up to 100% of the system capacity so that all task deadlines are met. Without loss of generality, we assume that the total task utilization imposed upon the targeted system is 100%. That is,

$$\sum_{i=0}^{n_p-1} \mu_i = m \tag{4.8}$$

In case the total utilization is less than m, dummy tasks are inserted to fulfill the system without any additional cost. Conceptually, a dummy task has its maximum utilization equal to one and its execution time is arbitrarily large. More than one dummy tasks may needed to fill the remaining system capacity unused by periodic tasks. All inserted dummy tasks therefore occupy the rest capacity of $m - \sum_{i=0}^{n_p-1} \mu_i$. During execution of dummy tasks, the assigned processor do nothing (or put into idle time for saving energy).

Now, we sequentially show that LAA provides the following result:

1. LAA maintains fully-assigned systems for every interval.
2. Maintaining fully-assigned systems for every interval can guarantee the schedulability of the system.

The first result comes straightforwardly by exploring Algorithm 1 described in Section 4.3. At the second phase, *LRET Adjustment*, all of slack times within the interval are distributed to tasks (including dummy ones) as their extra execution (Section 4.2.3). The system capacity of the interval is overspread with tasks' execution as a result. This indicates that fully-assigned system is achieved on the interval.

For the second result, the proof is supported with two lemmas and then a theorem as follows.

Lemma 4.5.1. *Given a feasible set of n_p periodic tasks scheduled on m identical processors and an interval I. If the total amount of local requested execution times of tasks on I does not exceed the system capacity of I, LAA scheduling can then finish all of the local requested execution times by the end of I.*

Proof. The system capacity on I is equal to $m * |I|$. At the third phase, *Task Assignment*, of Algorithm 1, this capacity is set for the capacity of the virtual uniprocessor established in the consecutive assignment. Since the total amount of local requested execution times is not greater than $m * |I|$, the capacity of the virtual uniprocessor is capable of accommodating that amount. In other words, the task assignment on the virtual uniprocessor is achievable. Then, scheduling plans obtained from the conversion step of the consecutive assignment process confirm the correctness of the lemma. □

Lemma 4.5.2. *Given a feasible set of n_p periodic tasks scheduled on m identical processors and two consecutive scheduling events t_1 and t_2 ($t_1, t_2 \in N$ and $0 \leq t_1 < t_2$). If making the system fully assigned can guarantee the schedulability (no deadline miss occurring) up to time t_1, then the system is also schedulable with the same scheduling manner up to time t_2.*

Proof. Firstly, the scheduling events t_1 and t_2 form an interval I with its length $|I| = t_2 - t_1$. Then, the total amount of local requested execution time of tasks on I can be derived from Equation 4.6 as follows:

$$E(t_1, t_2) = \sum_{i=1}^{n_p} M_i(t_1, t_2) + \sum_{i=1}^{n_p} P_i(t_1, t_2) \tag{4.9}$$

Applying Equation 4.4, the total mandatory execution of tasks on I can be obtained as:

$$\sum_{i=1}^{n_p} M_i(t_1, t_2) = \sum_{i=1}^{n_p} \lfloor \mu_i * t_2 \rfloor - \sum_{i=1}^{n_p} A_i(t_1) \tag{4.10}$$

Since the system is fully assigned up to t_1, the total already assigned resources of tasks is exactly equal to $m * t_1$. Then, by applying Equation 4.7 we have:

$$\sum_{i=1}^{n_p} M_i(t_1, t_2) = \sum_{i=1}^{n_p} \lfloor \mu_i * t_2 \rfloor - m * t_1 \tag{4.11}$$

Next, Because Equation 2.4 holds for the feasible task sets, the following result is obtained:

$$\sum_{i=1}^{n_p} M_i(t_1, t_2) \leq m * |I| \tag{4.12}$$

In addition, the distribution of slack times at the second phase of Algorithm 1 leads the following result:

$$\sum_{i=1}^{n_p} P_i(t_1, t_2) = m * |I| - \sum_{i=1}^{n_p} M_i(t_1, t_2) \tag{4.13}$$

Equation 4.12 and Equation 4.13 in combination imply that:

$$E(t_1, t_2) = m * |I| \tag{4.14}$$

Now, according to Lemma 4.5.1, we can claim that all local requested execution times of tasks can be accommodated by the end of interval I, or by t_2 and the fully-assigned system is achieved on I. In other words, the system can be successfully scheduled without any deadline misses up to time t_2. □

In the rest of this section, we are showing that, LAA can guarantee the schedulability of up to 100% for the system as a whole. It concisely states the following theory.

Theorem 4.5.3. *Consider a feasible set of n_p periodic tasks scheduled on m identical processors. Each task is characterized by a utilization $\mu \leq 1$ and an integer period $T > 0$. By maintaining fully-assigned systems for every time intervals, LAA is applicable to schedule any task set of utilization of up to 100% without deadline miss.*

Proof. Let $D = \{0, t_1, t_2, ...\}$ be a set of scheduling events (job releases) where $t_1, t_2, ... \in N$ and $0 < t_1 < t_2 < ...$ For the first interval $I_0 = [0, t_1)$, the total amount of local requested execution times of tasks is calculated as follows:

$$E(0, t_1) = \sum_{i=1}^{n_p} M_i(0, t_1) + \sum_{i=1}^{n_p} P_i(0, t_1) \tag{4.15}$$

Because the already-assigned resources of tasks at time 0 is equal to 0, the total mandatory execution of tasks on I_0 is:

$$\sum_{i=1}^{n_p} M_i(0, t_1) = \sum_{i=1}^{n_p} \lfloor \mu_i * t_1 \rfloor \tag{4.16}$$

Since Equation 2.4 holds, we have:

$$\sum_{i=1}^{n_p} M_i(0, t_1) \leq m * t_1 \tag{4.17}$$

Then, the total extra execution of tasks on I_0 is obtained as follows:

$$\sum_{i=1}^{n_p} P_i(0, t_1) = m * t_1 - \sum_{i=1}^{n_p} M_i(0, t_1) \tag{4.18}$$

Finally, we can easily reason out that $E(0, t_1) = m * t_1$. Accordingly, Lemma 4.5.1 indicates that the amount of local requested execution times can be successfully accommodated with a fully-assigned system on interval I_0. The system is therefore schedulable with LAA algorithm up to t_1. Then, applying Lemma 4.5.2 also indicates that a fully-assigned system on the next interval $I_1 = [t_1, t_2)$ can guarantee the schedulability up to t_2. This is repeated on successive intervals and therefore the schedulability is achieved for the whole system. □

4.6 Evaluation of LAA

4.6.1 Simulation environment

Evaluation criteria

The proposed algorithm is evaluated by software simulations. The targeted criteria include scheduler invocation, task migration, task preemption, and time complexity. The criteria follow definitions in Chapter 2. In simulation, scheduling overhead and overheads caused by task preemption and migration are ideal (zero). Instead, the number of scheduler invocation, task preemption and task migration are objects of assessment. The time complexity of algorithms is assessed through runtime overhead issued at system ticks and the maximum runtime overhead per tick is considered. Runtime overhead is obtained by adding up the number of cycles spent for each instruction execution in the algorithms. This calculation includes arithmetic, logical, and control operations in the algorithms. The instruction set of the Cortex-A9 processor [57, 58] is employed as reference for executing cycle estimation. Table 4.1 shows the simulated number of cycles spent for execution of different operations. The observation period is 100,000 ticks.

Table 4.1: Number of cycles of the targeted operations in simulation

Operation	# of cycles	Description
IADD	1	Integer addition
IMUL	2	Integer multiplication
FADD	4	Floating-point addition
FMUL	5	Floating-point multiplication
FDIV	15	Floating-point division
COMP	1	comparison
ASSIGN	1	Assignment
CEIL	1	Ceiling
FLOOR	1	Flooring

Task set

In simulation system, task sets are generated corresponding to the simulated number of processors available. Accordingly, the simulation system is set for cases of 4, 8, 16 and 32 processors. Periodic task sets involved in each case of processors have varied utilization levels from 75% to 100% with increment divided by 5%. The generation of a periodic task set follows the following steps:

1. Determine the number of processors.

2. Calculate the total targeted task utilization U_p ($U_p = \sum_{i=0}^{n_p} \mu_i$) for each case of utilization level. For example, in case of 4 processors, $U_p = 3$ if the utilization level is equal to 75% and $U_p = 3.6$ if the utilization level is equal to 95%.

3. Distribute U_p to tasks as their utilization. This step decides the tasks' utilizations and the number of tasks in a task set corresponding to U_p. Tasks' utilizations are generated using exponential distribution. Tasks' utilizations are restricted to values in $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$ (one decimal place). The restriction is applied for ease of implementing comparative algorithms. Furthermore, the number of tasks in a set follows the constraint: $m + 1 \leq n_p < 100$, where m is the number of processors. The lower bound of the constraint assures task migrations happening and the upper bound reflexes the limited number of tasks in practical embedded systems. Figure 4.5 shows a simple pseudo-code for this utilization distribution. In this code, *utildis* is an exponential distribution function with default rate 1. *generator* is a random engine to generate a value in the specified range.

```
std :: exponential_distribution <double> utildis(1);
int np;
int i;
while(n < m & n>100)
{
    np = 0;
    i = 0;
    while(Up > 0)
    {
        μ[i]=utildis(generator);
        Up = Up−μ[i];
        i ++;
        np ++;
    }
}
```

Figure 4.5: Simple mechanism of utilization distribution

4. Generate WCET and task period. For each task, WCET is a random value in range $[1, 20]$. The task period is task WCET divided task utilization. Limiting value of WCET is to shorten task period and then increase the number of job releases during the observation period.

5. Export generated data to files for simulation.

For each case of utilization level in each case of processor numbers, ten different task sets are simulated and the result is reported with the average value.

Comparison candidates

The proposed algorithm is evaluated in comparison with the existing candidates: Semi-partitioned reservation (SPR) [31], Pfair [32], and RUN [38]. In order to simulate the behaviors of RUN algorithm, *Best-Fit Bin Packing* approach is applied for the packing procedure. Similarly, *Worst-Fit Bin Packing* approach is selected for the partition procedure in SPR. The two packing methods are used for their best yields in the corresponding algorithm. Furthermore, SPR is excluded at the evaluation cases of periodic utilization of 100% since the algorithm occasionally failed in partition of task sets.

4.6.2 Simulation results of LAA

Scheduler invocation

Figure 4.6 shows the total number of scheduler invocation occurring during the observation period for four examined algorithms.

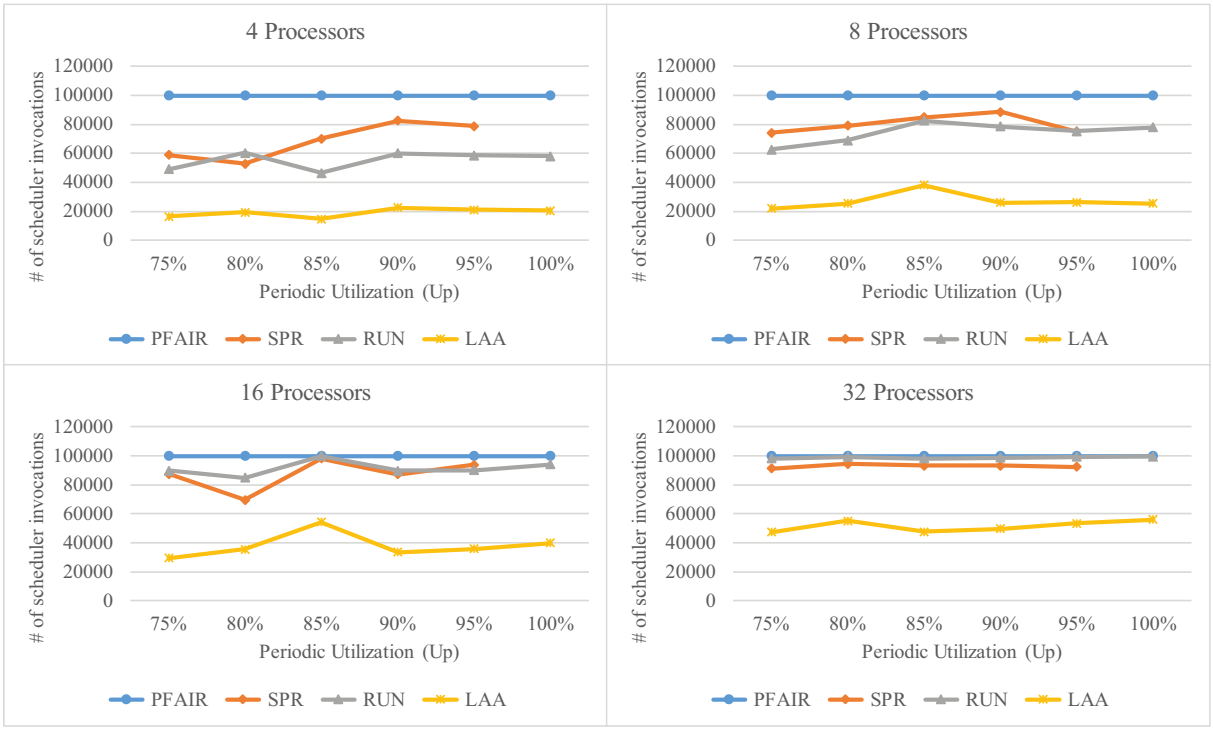

Figure 4.6: Scheduler invocation of LAA

As a fine time-scale scheduling, Pfair is the worst candidate in this criterion with the results of 100,000 scheduling points over all simulation cases. Obviously, SPR and RUN experience a similar trend that the number of scheduler invocation increases together with the number of tasks that causes the increase of job releases and job completions.

Specifically, RUN tends to cause fewer scheduler invocations than SPR does in cases of 4 and 8 processors; scheduler invocation in RUN is more frequent than in SPR in the other cases where the number of tasks and then jobs is larger. This is because RUN abstraction significantly increases dual servers in such cases. Overall, the results indicate that the proposed algorithm, LAA, outperforms the others in this criterion. Using only job releases as scheduling points, LAA reduces scheduler invocations by over 50% in almost all simulation cases. This is an important achievement of LAA.

Time complexity

Figure 4.7 displays the results of the maximum runtime overhead per tick which is used to assess the time complexity of algorithms. In all our simulation cases, PFAIR experiences significantly high runtime overheads than the other candidates. This is caused by complex procedure of task classification of PFAIR at every instant time. Therefore, the results of runtime overhead of PFAIR is not plotted here. For further assessment, results in Table 4.2 are helpful.

Overall, belonging to class of semi-partitioned schedulings, SPR shows advantage in reducing runtime overhead for all task sets. Compared to the optimal algorithm RUN, LAA shows better results in most of cases, especially on the scenarios of high numbers of processors. In cases of 4 and 8 processors, no one of LAA and RUN overwhelms the other and the gaps between their results are not much different. On the other hand, when the number of processors increases, the improvement of LAA becomes apparent with the clear improvement gap. In the case of 16 processors, LAA issues approximately 20% of runtime overhead less than RUN. The improvement gap increases to about 40% in the case of 32 processors.

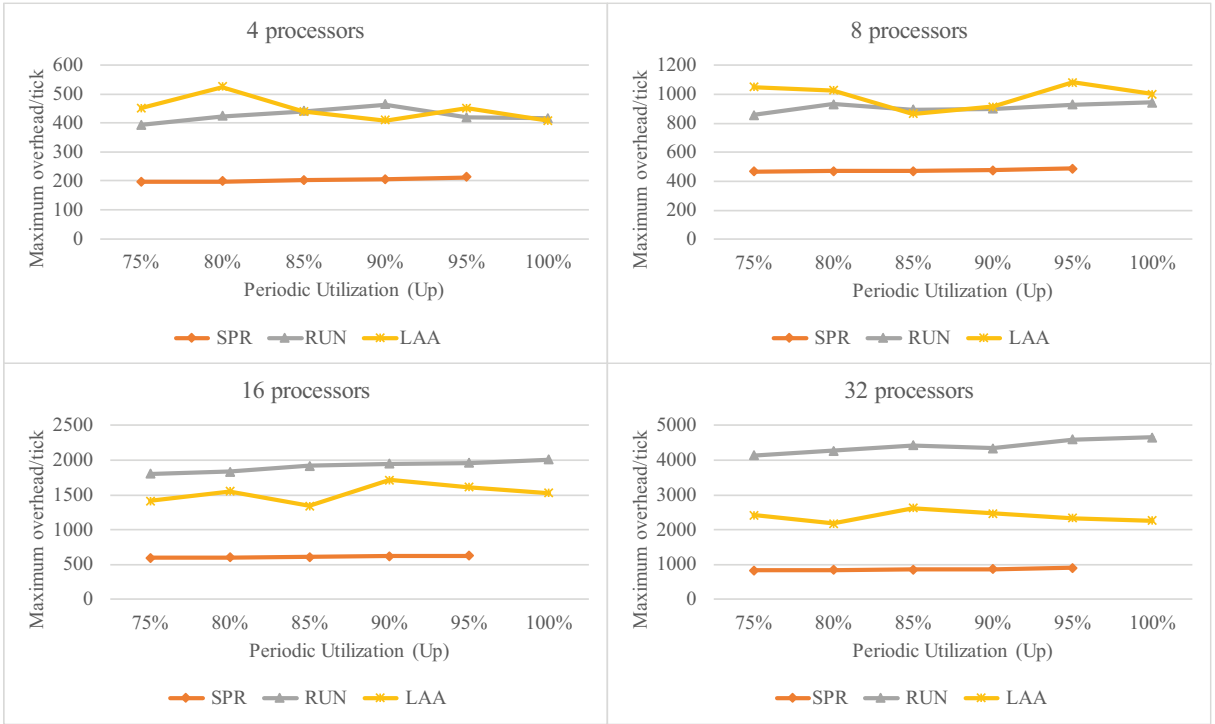

Figure 4.7: Runtime overhead of LAA

For deeper assessment of time complexity of algorithms, Table 4.2 shows the average number of operations invoked in algorithms' execution at $U_p = 95\%$ on cases of 4 and 16 processors in Figure 4.7. High runtime overheads of PFAIR are confirmed by a large number of operation compared with the other candidates. Especially, PFAIR executes many floating multiplications in comparing characteristic substring [32]; which dominantly causes runtime overhead.

Table 4.2 also indicates that although the total runtime overhead of LAA may less than that of RUN, RUN has an advantage of no execution of floating-point numbers. Whereas, LAA requires processor architectures with floating-point unit.

Table 4.2: Number of operations in execution of LAA

	4 processors			
	PFAIR	SPR	RUN	LAA
Overhead	**2932.3**	**212.1**	**418**	**449.9**
IADD	682.6	4	19.1	94.7
IMUL	0	0	0	1
FMUL	274.2	0	0	5.9
COMP	474.3	53	216.3	206.6
ASSIGN	250.2	155.1	182.6	111.2
FLOOR	154.2	0	0	5.9
	16 processors			
	PFAIR	SPR	RUN	LAA
Overhead	**16954.4**	**626.5**	**1959.9**	**1612.1**
IADD	4456.8	110	79.3	400
IMUL	0	0	0	1
FMUL	1315.6	0	0	27.1
COMP	3883	199.9	1035.2	649.6
ASSIGN	391	316.6	845.4	397.9
FLOOR	1645.6	0	0	27.1

In addition, respect to the definition in Chapter 2, accumulative overhead is potentially improved with LAA. Combining characteristics of fewer scheduler invocations and low runtime overhead, LAA substantially reduces accumulative overhead as the sum of runtime overheads incurring at every scheduler invocation. This effectively improves the throughput of the system in a manner that the system can spend more time executing tasks, rather than processing the scheduling algorithm.

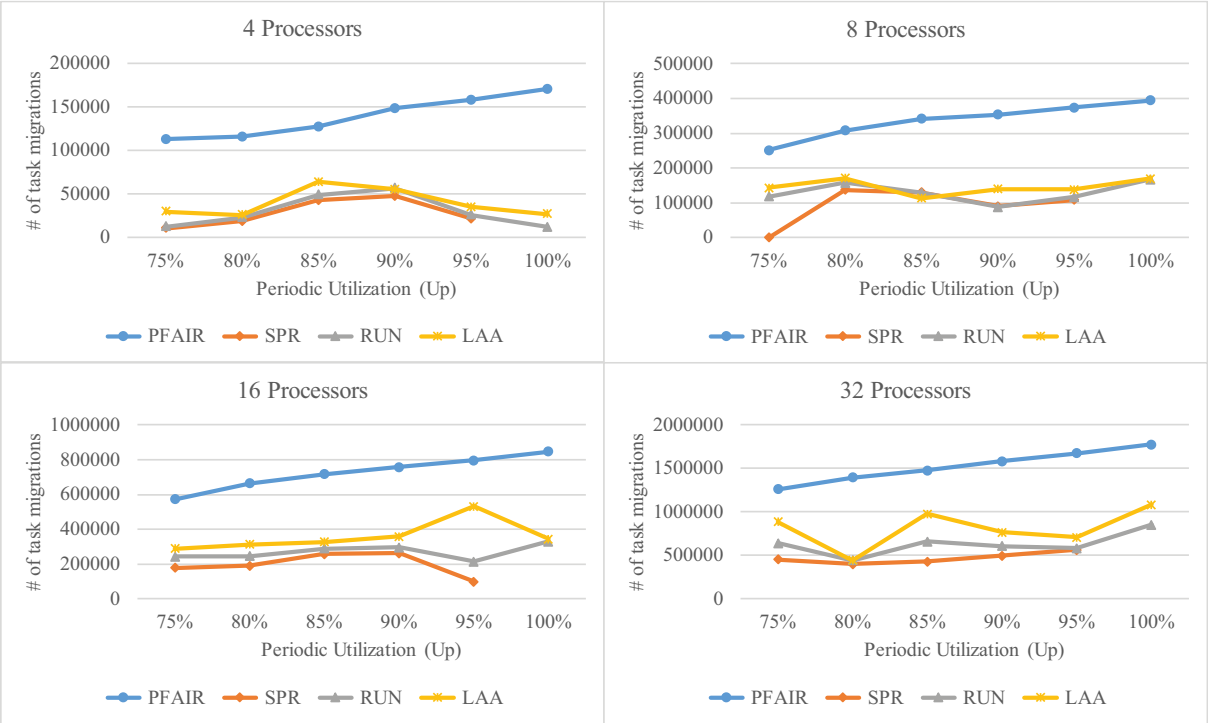

Figure 4.8: Task migration of LAA

Task migration and preemption

The total numbers of task migration and task preemption are displayed in Figure 4.8 and Figure 4.9, respectively. Firstly, Pfair causes the largest number of migrations and preemptions due to the impact of time-quanta schedules. This is because tasks' executions are fragmentary by time unit. Fragmentation makes jobs of tasks migrated and preempted frequently during their presentation.

As an advantage of partition scheduling, SPR is better than RUN and LAA in terms of task migration. Conversely, it experiences higher number of task preemptions than RUN and LAA do, especially at heavy workloads of 95%. In other words, limiting tasks' moving among processors negatively increases the number of preemptions.

Compared with RUN scheduling, LAA is worse in terms of task migration, except cases of 90% on 4 processors, 85% on 8 processors, 80% on 32 processors. However, LAA is a superior candidate when it comes to exhibiting lower numbers of task preemption than RUN does. Overall, it can be seen that the gaps between migration and preemption results of LAA and RUN are not large.

Furthermore, it is notable that results of task migration and preemption also indicate a trend that increase of task migrations potentially has potential to decrease task preemptions. In other words, moving a task to alternative processors means giving more chance to the task to be executed seamlessly, which can reduce the task preemption. For example, let us observe results of migration and preemption of LAA and RUN on the case of 32 processors. For task migration, LAA has better result than RUN on the case of 80%; but increases task migration on the other cases. Whereas, for task preemption, 80% is only the case where LAA is worst than RUN; LAA issues less task preemptions

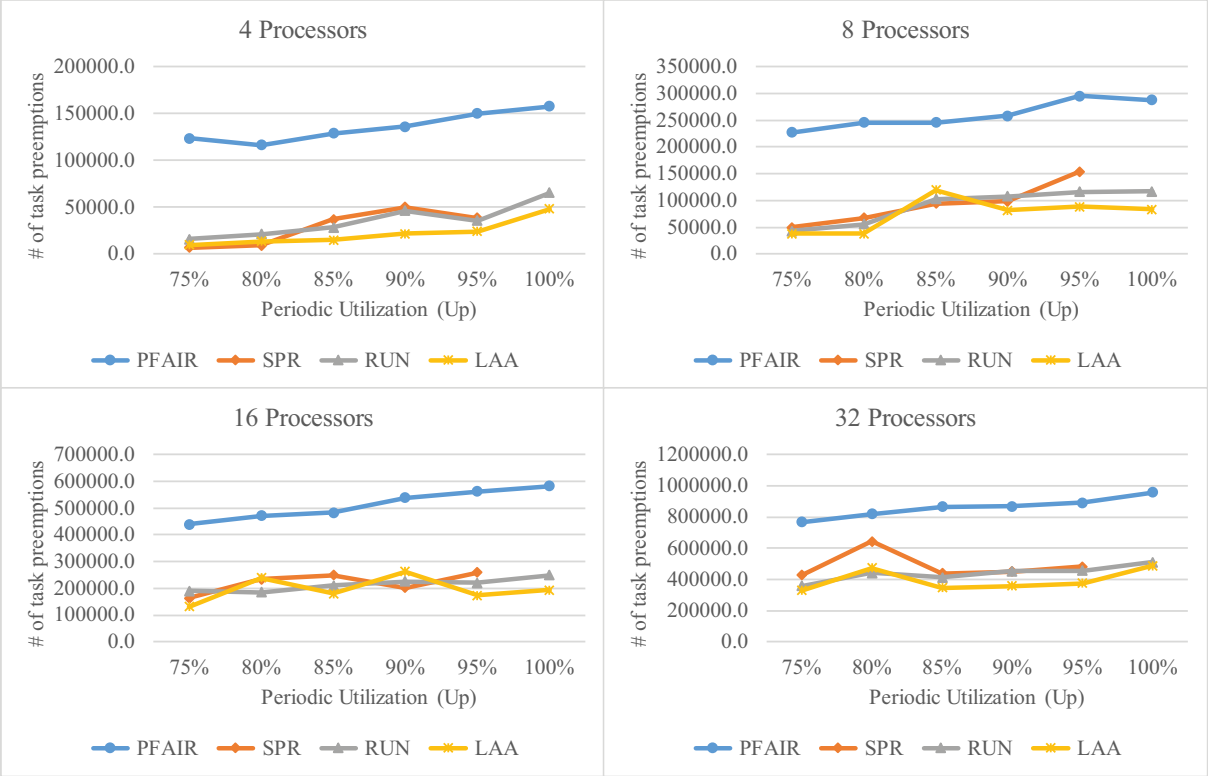

Figure 4.9: Task preemption of LAA

than RUN on the other cases.

This can be considered as trade-off between task migration and task preemption in multiprocessor scheduling.

4.7 Conclusion: Effectiveness and Limitation of LAA

4.7.1 Effectiveness of LAA

First, LAA clearly shows effectiveness in reducing accumulative runtime overhead. This advantage is obtained from the combination of fewer scheduler invocations and low time complexity. The system performance therefore is improved in a sense that the system can afford to execute more tasks in practical.

Especially in embedded systems where the number of tasks is smaller, LAA may become more effective to reduce accumulative runtime overhead. First, it is plausible that smaller numbers of tasks potentially issue fewer number of job releases. The number of scheduler invocations is then decreased. Second, observing Algorithm 1, calculations of LRETs at steps 1 and 2 are done for each task and tasks are individually allocated on the virtual uniprocessor at step 3. Therefore, the process of scheduling can be faster when the number of tasks is decreased. This means the runtime overhead (scheduling overhead) is lower for smaller numbers of tasks.

Second, as showed in Algorithm 1, local requested execution times of tasks are resulted in integral values. It means the actual context switches can be determined easier at

runtime. This is a plus of LAA compared with DP-WRAP where complicated calculations are required to decide the context switches at runtime.

Third, full assignment scheme allows LAA to better exploit the system capacity for high performance. This is better than the distribution scheme of BF scheduling. Furthermore, the adjustment for full assignment actually does not required any priority of tasks; namely, tasks can be selected arbitrarily to increase their local requested execution time at the second phase in Algorithm 1. It is effective to eliminate the additional runtime overhead for task sorting.

Finally, thank to relatively low time complexity, LAA can utilize a more complex process of consecutive assignment than the simple model in DP-WRAP (also McNaughton's wrap algorithm). That is, instead of arranging tasks to processors simply based on task indexes in DP-WRAP, tasks are selectively distributed to processors in LAA to reduce task preemption and migration. Therefore, LAA can guarantee task preemption and migration comparable with the other candidates. Furthermore, migration occurs at most once for a task during an interval based on the consecutive assignment scheme. This can be considered as an effectiveness in terms of reducing task migrations. However, since the migratory task is restrictedly moved to the adjacent processor in this assignment scheme, it can cause unexpected preemptions on the tasks that have already assigned. In fact, a more complex procedure can be introduced to select a more appropriate processor for the migratory task to move, which is in trade-off with the increase of runtime overhead.

4.7.2 Limitation of LAA

The first limitation of LAA is that it is not a work conserving system. In fact, scheduling plans on intervals are made using WCET of tasks. The schedule of tasks at runtime within an interval is completely instructed by the corresponding scheduling plan. However, the actual execution of jobs of tasks is normally less than its WCET. Therefore, a (job of) task can spend shorter execution than its assigned amount of LRET on the interval. In this case, due to the preserved resources of scheduling plan, the processor is continuously assigned for the task until its LRET is exhausted. The processor becomes idle unnecessarily. SPR and RUN are better to deal with this situation when the other tasks can utilized the processor. Fortunately, this disadvantage of LAA happens only within the involved interval. The processor is utilized in the next interval since the already finished tasks are not included in calculation of scheduling plan for the next interval.

Another limitation is that the current version of LAA is incapable of scheduling sporadic tasks. However, since LAA is based on notion of proportionate scheduling similarly to DP-WRAP, it is considerable to apply DP-WRAP's approach for sporadic tasks. This is worth considering as future work.

Chapter 5

Aperiodic Task Scheduling

In periodic task scheduling, guaranteeing deadline constraints for periodic tasks is the top requirement. Deadlines of periodic tasks are fixedly decided and any deadline miss can make the system unsafe. Differently, aperiodic tasks are not strict on deadline requirements. There exists soft real-time aperiodic tasks executed without deadlines. Another type of aperiodic tasks is hard real-time, which is assigned an absolute deadline at its release. The absolute deadlines of hard real-time aperiodic tasks are calculated flexibly based on the system situation (unused system capacity for example). Instead, improving response time is most important for aperiodic tasks. Appearance of aperiodic tasks leads to increase of time complexity of scheduling algorithms. Scheduling aperiodic tasks with concept of servers has emerged as a promising approach. In this section, preliminary studies including enhanced virtual release advancing for aperiodic tasks are presented.

5.1 Aperiodic task scheduling with concept of servers

Looking back to the problem of real-time task scheduling on uniprocessor systems, concepts of servers is very successful to deal with aperiodic tasks. There are several optimal solutions introduced to schedule mixture context of periodic and aperiodic tasks on uniprocessors. Total Bandwidth Server [40], Dynamic Priority Exchange [41], and Constant Bandwidth Server [42] are representative scheduling schemes.

Among these approaches, TBS shows remarkable achievements on responsiveness with relatively low implementation complexity. In TBS context, the periodic tasks' deadlines are explicitly designed to be equal to their period ending while the aperiodic tasks are assigned absolute deadlines using Equation 5.1. In this equation, k means the kth aperiodic task, d_k is the absolute deadline of the kth task, r_k is release (arrival) time of the kth task, d_{k-1} is the absolute deadline of the $(k-1)$th (previous) task, C_k is worst-case execution time of the kth task, and U_s is the bandwidth of the server (the remaining system capacity unused by the periodic tasks).

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s} \tag{5.1}$$

EDF [12] is then applied to schedule periodic and aperiodic tasks altogether. For the aim of further improving response time for aperiodic tasks, virtual release advancing (VRA) technique was introduced [43]. The techniques based-on TBS context tried to assign the target aperiodic tasks earlier deadlines in a sense that earlier deadlines lead

to earlier scheduling for aperiodic tasks under EDF. Materially, earlier deadlines can be obtained by introducing virtual release times earlier than task's actual release times in the deadline calculation. To this end, VRA tries to virtually and retroactively move release times backward to the past while maintaining the past schedule. In VRA, absolute deadline of aperiodic tasks are calculated using Equation 5.2 in which vr_k, virtual release time obtained by the virtual advancing, is replacing r_k in Equation 5.1 of the original TBS.

$$d_k = max(\mathbf{vr_k}, d_{k-1}) + \frac{C_k}{U_s} \tag{5.2}$$

The scheduling example in Figure 5.1 shows the effectiveness of VRA in improving responsiveness of aperiodic tasks. The example encompasses two periodic tasks $\tau_0 = (1/3, 3)$ and $\tau_1 = (1/2, 4)$. The processor utilization by the periodic tasks is $U_p = 1/3 + 1/2 = 5/6$ and then the bandwidth of the server is $U_s = 1 - 5/6 = 1/6$. An aperiodic task is supposed to enter the system at $t = 6$ with its execution time of 1. According to the original TBS, the aperiodic task receives an absolute deadline at $t = 12$. Scheduling with this deadline leads the aperiodic task to finish at $t = 8$ with response time of 2.

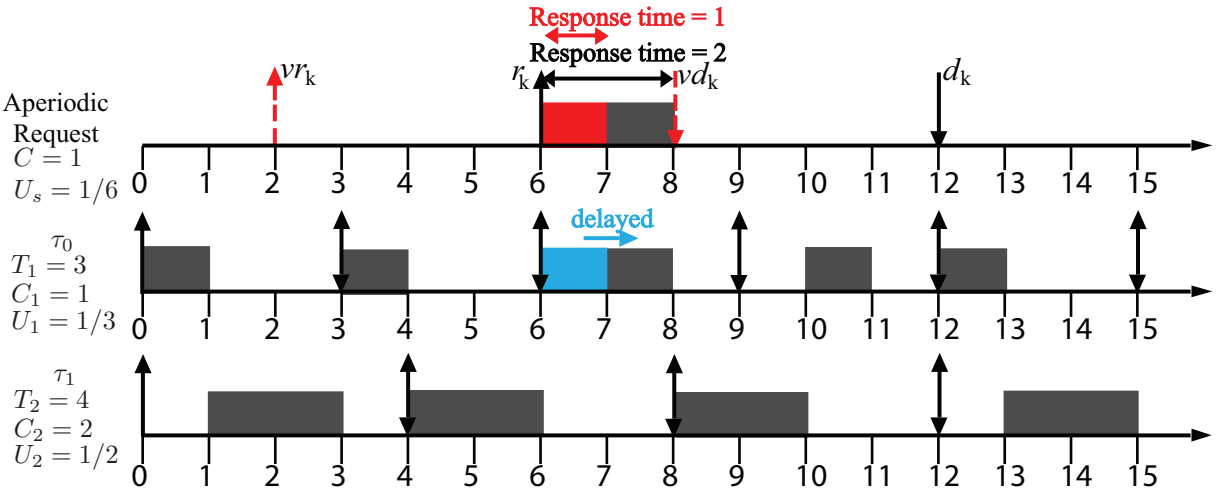

Figure 5.1: Example of virtual release advancing

In this case, by applying the virtual release advancing technique, an earlier virtual release time for the aperiodic task is set up at $t = 2$ as indicated by a red, upward dashed arrow. This new release time leads the absolute deadline in VRA at $t = 8$ as depicted by a red, downward dashed arrow, four ticks earlier than the original deadline in TBS. This new deadline is earlier than that of the third instance of τ_0 (at $t = 9$). Under the EDF algorithm, the target aperiodic task is hence immediately executed ahead of the periodic instance and finishes sooner at $t = 7$ with the response time of 1.

In spite of significantly improving the response time of aperiodic tasks, VRA has a problem with its high time complexity. The runtime overhead of VRA increases due to the repeat of release time advancing backward to the pass by single slot. The problem is motivation of the technique called Enhanced Virtual Release Advancing. Study on Enhanced Virtual Release Advancing is considered as one of preliminary researches to deal with aperiodic tasks in this dissertation. Section 5.2 presents the technique in detail.

5.2 Enhanced Virtual Release Advancing Algorithm for Aperiodic Servers

5.2.1 Limitations causing runtime overhead in VRA

In fact, virtual release times of tasks (vr_k) in VRA are obtained by supposedly moving the actual release times (r_k) backward to the past. A loop is implemented for this purpose in respect of three limit factors: previous deadline, last empty slot, and previously-used maximum deadline [43]. As introduced above, the VRA technique checks limit factors slot by slot from the target task's arrival time (r_k) backward to the past. The loop is then checking limits slot by slot and stops when one of the limit factors is reached. This advancing method by single slot may increase runtime overhead as follows.

- Firstly, the algorithm checks two factors, previous deadline and last empty slot, in every iteration. However, there are actually only one previous deadline and one last empty slot and they remain constant during the algorithm's execution.
- Secondly, consecutive slots spent by the same instance of a task are checked one by one. It is obviously unnecessary since such slots have the same associated deadline.
- Thirdly, the limit how long past the release time is advanced is not defined.

By observing these sources of runtime overhead, EVRA algorithm described in Section 5.2.3 is introduced to overcome these inefficiencies.

5.2.2 Enhancement of the EVRA algorithm

Different from the original VRA where the advancing is done for tasks' release times, EVRA is deadline-based advancing. That is, the absolute deadlines of tasks are virtually moving backward to the past. To this end, definitions of boundary deadline, check-bounding slot, and representative slot are emerged in EVRA. (The details of these definitions can be found in [54].) The approach of deadline-base advancing has significant advantages to reduce the sources of runtime overhead as follows:

- First, with boundary deadlines, limits of previous task's deadline and last empty slot. As a result, these limits are checked once only. This helps to reduce the influence of repeated checking.
- Check-bounding slot is introduced to solve the limit how long past the deadline advancing is performed.
- EVRA advances tasks' deadlines instance by instance using representative slots of instances. This is effective in reducing loop count and memory costs. In the original VRA, the associated deadlines of the used slots all have to be recorded for advancing by single slot. In EVRA, on the other hand, only the first slots of instances need recording. Actually, the number of released instances is in most cases smaller the number of past slots. Therefore, advancing over the representative slots is obviously more efficient than over every used slots.

- In addition, EVRA is supported with a hardware accelerator which allows tasks' absolute deadlines to be advanced faster. This shows effectiveness in further reducing runtime overhead of EVRA.

5.2.3 Proposed algorithm of EVRA

Algorithm 2 shows the pseudo code of the EVRA algorithm. Notations used in the algorithm are described in Table 5.1. The purpose of the algorithm is to obtain a virtual deadline, vd_k, for the targeted aperiodic task by advancing the actual release time, r_k, backward to the past. The advancing is done under conditions of limiting deadlines (vd'_k, vd''_k, and vd'''_k) calculated corresponding to the previous deadline, last empty slot, and last starting time of τ_{max} and the maximum deadline of the previous used slots (max_dl). Accordingly, the expected virtual deadline vd_k cannot be earlier than vd'_k, vd''_k, vd'''_k, or max_dl.

Table 5.1: Notations of EVRA algorithm

Notation	**Description**
vd_k	The expected virtual deadline for the targeted aperiodic task
r_k	The actual release time of the target aperiodic task
vr_k	The virtual release time of the target aperiodic task
C_k	The worse-case execution time of the target task
d_{k-1}	The deadline of the $(k-1)$th (previous) aperiodic task
U_s	The bandwidth of TBS server
U_s	The bandwidth of TBS server
$last\_empty$	The slot number of the last empty slot (which is assumed to be -1 for no empty slot)
ls_{max}	The start time of the last instance of τ_{max} (the task that has the maximum period)
vd'_k, vd''_k, vd'''_k	Limiting deadlines calculated corresponding to the previous deadline, last empty slot, and last starting time of τ_{max}
$dl[]$	Record of past slots' associated deadlines
$S[]$	The start times of the released instances
max_dl	Holding the maximum associated deadline of traced $S[]$ elements during the advancing process
Z	The number of released instances (up to the current time of executing)

For ease of understanding, the algorithm's execution is explained in accompany with an example depicted in Figure 5.2. In this example, EVRA is applied for the target aperiodic task in the scheduling situation in Figure 5.1 above. Here, when the scheduler is invoked

at time $t = 6$ as the target task is released, the algorithm is executed. According to this situation, at the beginning of the algorithm's execution (the first step), vd_k is initialized using the actual release time r_k. vd_k then results in value of 12.

Algorithm 2: The EVRA algorithm

```
1: /*Definition*/
2:     vd_k = r_k + C_k/U_s
3: /*Checking limit of k-1-th deadline*/
4:     vr'_k = d_{k-1}
5:     vd'_k = vr'_k + C_k/U_s
6:     if vd_k ≤ vd'_k then
7:         vd_k = vd'_k
8:         Goto End
9:     endif
10: /*Checking limit of last empty slot*/
11:     vr''_k = last_empty + 1
12:     vd''_k = vr''_k + C_k/U_s
13: /*Checking limit of previously-used slots*/
14:     vd'''_k = dl[ls_max]
15:     bound = max(vd'_k, vd''_k, vd'''_k)
16:     i = Z − 1
17:     max_dl = 0
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 ≤ max_dl then
24:             break
25:         endif
26:         vd_k = vr_k + C_k/U_s
27:         if vd_k ≤ max_dl then
28:             vd_k = max_dl
29:             break
30:         else
31:             i = i − 1
32:         endif
33:     endwhile
34:     Label: End
```

Then three limiting deadlines (vd'_k, vd''_k, vd'''_k) are calculated at steps 2, 3, and 4, respectively. The first limiting deadlines (vd'_k) is calculated corresponding to the deadline of the previous aperiodic tasks using the TSB formula. Because the previous deadline (d_{k-1}) is assumed to be 0 for no previous aperiodic task, vd'_k then results in value of 6. The expected virtual deadline is checked with this limiting deadline so that if the expected virtual deadline is earlier than or equal to vd'_k, the it is set to vd'_k, and the algorithm finishes. Similarly, the second limiting deadline (vd''_k) is calculated corresponding to the last empty slot. For this example, vd''_k gets value of 6 due to no empty slot in this situation.

The third limiting deadline (vd_k''') is then calculated corresponding to the ls_{max}th slot's associated deadline. In this case, vd_k''' is equal to 8 since $ls_{max} = 4$ and $dl[ls_{max}] = 8$. As the maximal of the three limiting deadlines, a variable *bound* presents the check-bounding deadline of the algorithm, which is of 8 for this example.

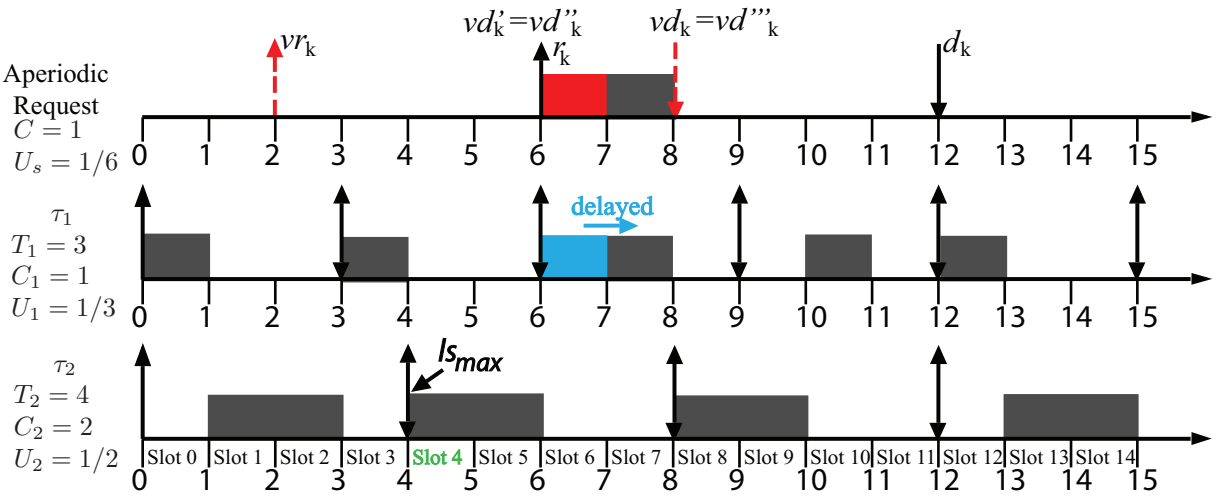

Figure 5.2: Example of deadline advancing in EVRA

Next, a loop is implemented to advance vd_k backward to the past in comparison with the limit of previously-used maximum deadline for each of the traced $S[]$ elements. The value of vd_k obtained after the loop execution satisfies conditions of *bound* and max_dl and therefore it is the expected deadline.

Materially in this example, temporary variables for the loop execution are initiated as: $i = 3$ for 4 released instances ($Z = 4$) and $max\_dl = 0$. Since condition $vd_k > bound$ is satisfied for $12 > 8$, the advancing loop starts to assign $vr_k = S[3] = 4$ (the start time of the third released instance). Then, the if-state updates max_dl to be 8 ($dl[4] = 8$). Since $vd_k > max\_dl$ ($12 > 8$), the algorithm's execution continues to update vd_k to be 10 at line 26. The updated value of vd_k remains greater than max_dl and the if-state at line 27 is not satisfied. Consequently, the $else$-state at line 31 is executed to decrease variable i by 1 and the advancing loop repeats.

For the second repeat of the advancing loop, with the similar procedure, considered variables result in $i = 2$, $vr_k = S[2] = 3$, $max\_dl = 8$, and $vd_k = 9$. With these values, the advancing loop continuously repeat for the third time of $i = 1$. At this time of iteration, $vr_k = S[1] = 1$, $max\_dl = 8$, and $vd_k = 7$. The condition of if-state at line 23 is now passed, then vd_k is set to be equal to max_dl (or 8) and the advancing loop stops. After three times of deadline advancing, the algorithm finishes with the tentative virtual deadline of 8.

Note that if the original VRA is applied for this example, the virtual release time is moved backward to the past from $t = 6$ until $t = 2$. Due to moving slot by slot, the advancing takes four times of repeat for vk_k to reach value of 2, which leads to the virtual deadline of 8. This example hence shows the effectiveness of EVRA with reduction in the number of loop count compared with the original VRA.

5.2.4 Hardware accelerator for EVRA

The aim of hardware accelerator is to further reduce the runtime overhead spent to execute the algorithm. Figure 5.3 shows the block diagram of the accelerator hardware that is integrated into the hardware system to support deadline calculation of EVRA. Connections between the accelerator and CPU are shown as well. In this system, the CPU is targeted for an ARM Cortex-A9 processor. Connections are implemented using general input/output ports (GPIO).

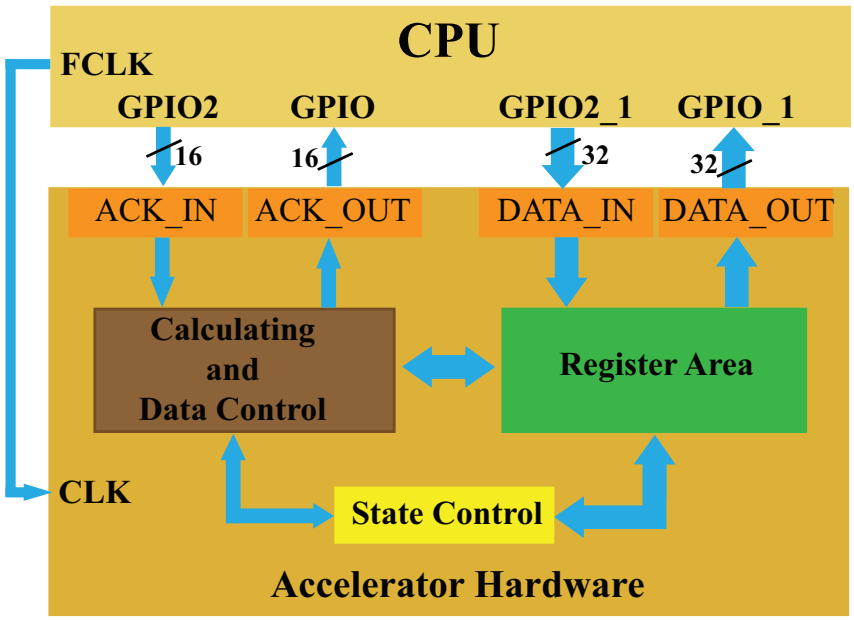

Figure 5.3: Block diagram of the accelerator hardware of EVRA

As shown in the figure, the hardware accelerator consists of four main parts: I/O registers, Calculating and Data Control unit, Register Area, and State Control unit. The I/O registers are deployed to connect directly to CPU's GPIO ports for mutual communication. The purpose of I/O registers is to temporarily store information of the communication as follows:

- ACK_IN: storing an ACK request which has been sent by the CPU.
- ACK_OUT: storing an ACK response which the hardware needs to send to the CPU.
- DATA_IN: storing input data which has been sent by the CPU.
- DATA_OUT: storing output data which the hardware needs to send to the CPU.

These registers are accessed specifically by the operating system through their fixedly assigned address in the physical address map.

A main part of the accelerator is the *Calculating and Data Control* unit, which performs deadline calculations and is controlled by a *State Control* unit. *Calculating and Data Control* unit is designed to profit from parallel calculations to reduce runtime overhead compared with software processing. *State Control* unit is a state machine synchronize by the $FCLK$ clock the processor.

The last component of the hardware accelerator is *Register Area* which plays a role of an internal memory. This area is separated into two segments for private data and shared data. The former segment is composed of registers where private data of the accelerator are stored. Therefore, only the accelerator can access this segment. On the other hand, the latter segment consists of indexed registers. This segment is to store data shared between the accelerator and the CPU. Therefore, both the accelerator and the CPU can access the global segment. The shared registers are indexed with register IDs which is described in the following section of communication of the CPU and accelerator.

Communication between CPU and accelerator

The communications between the CPU and accelerator applies a request-response protocol in which 16-bit request commands and 32-bit data buses are used. As shown in the block diagram in Figure 5.3, two 16-bit ports, named GPIO and GPIO2, are used for the CPU to send requests to and receive responses from the accelerator. Similarly, two 32-bit ports GPIO_1 and GPIO2_1 are utilized for mutual data transmissions. A typical communication procedure to request the accelerator is as follows:

1. CPU sends a necessary data to the data port (GPIO2_1);
2. CPU sends a corresponding request command to the request port (GPIO2);
3. CPU keeps waiting for a response acknowledging of valid data from the accelerator at port GPIO;
4. CPU loads needed data at port GPIO_1.

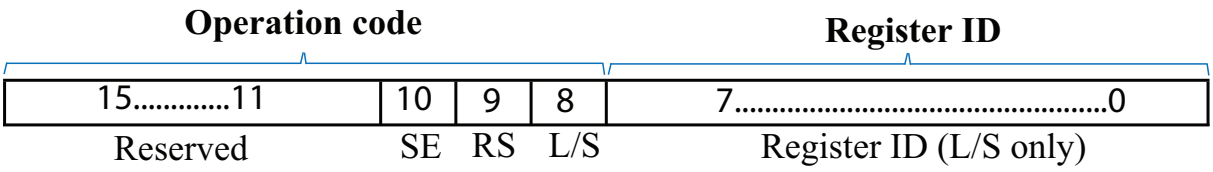

Figure 5.4: Structure of a request command

The structure if a request command is depicted in Figure 5.4. A request command accordingly consists of 16-bit information. The least significant 8 bits form a register ID used for accessing the shared registers of the *Register Area* of the accelerator. The other 8 bits are considered as an operation code (op-code) to determine exactly which operation is requested. The op-code is composed of:

- The 8th bit (L/S) specifies whether it is loading or storing operation. The bit is set to 1 for loading and clear to 0 for storing.
- The 9th bit (RS) is for reset operation. This activates with value of 1.
- The 10th bit (SE) is for enabling operation. When the bit is set to 1, the accelerator is enabled for deadline calculation. When one of reset or enabling operations is active, the least significant 8 bits are regarded as "don't care".

- The remaining bits (bits from 11 to 15) are preserved for future use.

In respect of the structure of request commands, the accelerator is designed with four operation: storing, loading, reset, and enabling.

In the scope of this dissertation, EVRA is briefly introduced with enhanced points, proposed algorithm, and overview of the integrated hardware accelerator. For more detailed analysis and evaluation of EVRA and the accelerator, readers concerning can be satisfied in [54].

5.3 Scheduling aperiodic tasks on multiprocessors

It is fact that a vast number of algorithms have been proposed for the problem of multiprocessor real-time task scheduling. They are in majority focused on predictable workloads like periodic tasks which have foreseeable release times. Dynamic workloads including aperiodic tasks, on the other hand, cause difficulties for scheduling due to their uncertain parameters such as entering time. Effective approaches for hybrid systems of periodic and aperiodic tasks are hence still in consideration.

Review in Section 5.1 shows that concept of servers is very popular approach used in the context of uniprocessor systems to deal with aperiodic tasks. CBS server [42] and TBS server [40] are two of the famous algorithms which optimally schedule hybrid task sets. There are also a number of techniques introduced to improve such scheduling algorithms. As preliminary researches to deal with aperiodic tasks, authors of this study have also conducted several researches related to scheduling aperiodic tasks with server concepts [54, 55, 56], especially achievements with EVRA algorithm.

However, unfortunately, applying such server approaches to the context of multiprocessors is found ineffective. M-CBS [47] and M-TBS [48] are the two representatives of exploiting CBS and TBS on multiprocessor systems. The obtained schedulability of these approaches is significantly lower than the system capacity.

There exists another way to apply the concept of servers on multiprocessors. That is modeling servers like periodic tasks server's utilization and period. This approach is to exploit the optimal solutions having been proposed for scheduling periodic tasks. Such servers are found in implementation of Srinivasan's research [49] which is targeted for Pfair scheduling. By this way, servers are scheduled together with periodic tasks and their schedule is obviously preserved for aperiodic tasks' execution. This shows an potential way to deal with dynamic workloads on multiprocessor systems.

Since it seems to be ineffective to extend EVRA to deal with aperiodic tasks on multiprocessors, the approach of modeled servers like periodic tasks is investigated in this dissertation. Servers can be integrated into LAA to schedule hybrid task sets on multiprocessor systems. This is the research of Chapter 6.

Chapter 6

Enhanced Local Assignment Algorithm for Scheduling Hybrid Task Sets

6.1 Introduction to Enhanced Local Assignment Algorithm - LAA+

As seen during the last decades, real-time embedded systems have been increased quickly in diversity and complexity. Homogeneous task sets including single type of tasks become unfashionable in practical real-time applications. Hybrid task sets which are combinations of different types of tasks frequently occur in today's applications. Mixture task sets of periodic and aperiodic tasks are the most popular and attractive combination to researchers in the embedded field. One of the challenging problems due to mixture task sets is task scheduling. Scheduling algorithms of this context essentially need to provide the following characteristics:

- Guaranteeing the schedule of periodic tasks so that they meet all their deadlines;
- Improving the responsiveness of aperiodic tasks. Certainly, algorithms which can produce shorter response times for aperiodic tasks are more preferable.

The occurrence of aperiodic tasks rises difficulties to scheduling algorithms. The release times of aperiodic tasks are unpredictable; namely, aperiodic tasks can enter the system at any time. The requested execution times of aperiodic tasks at every instant presentation are nondeterministic also. An effective scheduling algorithm for mixture task sets are motivation of the research presented in this chapter.

As seen in Chapter 4, Local Assignment Algorithm (LAA) is an effective solution to the problem of periodic schedule on multiprocessor systems. LAA is identified as an optimal algorithm and does not rely on any off-line process. In addition, the algorithm effectively schedules periodic tasks with fewer scheduler invocation and relatively low time complexity compared with the existing algorithms. Efficient exploitation of system capacity is another advantage of LAA scheduling. Therefore, LAA is selected to deal with aperiodic tasks in our research.

We introduce an enhanced Local Assignment Algorithm for the scheduling context of mixture task sets. The original LAA is enhanced with the integration of servers for

aperiodic tasks' schedule. The enhanced algorithm for mixture task sets is called **LAA+**, which mean LAA plus servers. LAA+ is therefore targeted to the following goals:

- Guaranteeing the schedule of periodic tasks;
- Reducing the response times of aperiodic tasks;
- Not significantly increase the time complexity.

The rest of this chapter is structured in six sections. Section 6.2 introduces about the integration of servers into the system. Section 6.3 shows definitions of LAA+. In Section 6.4 is the procedure of LAA+ in detail. The proof of schedulability guarantee is provided in Section 6.5. The evaluation of LAA+ is exhibited in Section 6.6. Finally in Section 6.7 is the conclusion where effectiveness and limitation of LAA+ are discussed.

6.2 Integration of servers

6.2.1 Server establishment

At first, we expected to extend the research of servers in EVRA to the context of multiprocessors. However, it is found that TBS servers are ineffective in multiprocessor systems due to the low schedulability. Therefore, we decide to use the model of servers in [49]. Namely, servers are modeled equivalently to a periodic task with two parameters: a utilization rate and a period. This approach has a big advantage that LAA can be utilized to schedule the system of periodic tasks and servers without complicated modification of the algorithm procedure.

Algorithm 3 shows the procedure to establish from the remaining system capacity unused by periodic tasks. Let n_s and U_s denote the number of servers and the total server utilization, respectively. To the best exploitation of system capacity, U_s is considered as the remaining system capacity unused by periodic tasks; that is, $U_s = m - U_p$ where m is the number of processors and U_p is the total utilization of periodic tasks. U_s is distributed to servers as their utilization so that each server S_j has a utilization rate $U_j \leq 1$ where $0 \leq j < n_s$. Materially, there are $\lfloor U_s \rfloor$ servers with utilization of 1 and one server with utilization of $U_s - \lfloor U_s \rfloor$. If $0 < U_p < m$, then $n_s = \lfloor U_s \rfloor + 1$.

The server establishment allows servers to present concurrently in the system. Note that, conceptually a server S_j can be assigned any utilization rate U_j in range $(0, 1]$. However, in this scheme of utilization distribution, servers are supposed to gain as high utilization rate as possible (utilization rate of 1 is desirable). This intention is to lower the number of servers, which tends to reduce the time complexity of the algorithm. In addition, servers with higher utilization rate are believed in better service for aperiodic tasks to improve the responsiveness.

Servers are supposed to have their periods arbitrarily large. Such servers are scheduled together with periodic tasks. The schedules of servers are dedicatedly preserved for aperiodic tasks' execution at runtime. Obviously, we can see that the servers together with periodic tasks fill the system capacity. In other words, we have:

$$\sum_{i=0}^{n_p-1} \mu_i + \sum_{j=0}^{n_s-1} U_j = m \tag{6.1}$$

Algorithm 3: Server Establishment

n_s: *number of servers*
U_s: *total utilization of servers*
j: *server index*
$n_s \leftarrow 0$
$U_s \leftarrow m - U_p$
$j \leftarrow 0$
(1) Server utilization:

$$U_j = \begin{cases} 1, & \text{if } U_s \geq 1 \\ U_s, & \text{otherwise} \end{cases}$$

$U_s \leftarrow U_s - U_j$
(2) Creating server:
create a server S_j with utilization rate U_j
$n_s \leftarrow n_s + 1$
$j \leftarrow j + 1$
(3) Check the system fullness

$$U_s : \begin{cases} U_s = 0, & \text{stop creating server} \\ U_s > 0, & \text{repeat (1)} \end{cases}$$

6.2.2 Assignment of aperiodic tasks to servers

Servers are responsible for serving aperiodic tasks in the following manner:

- Each server services at most one aperiodic task at a time. This requirement is involved to prevent an aperiodic task from concurrent executions on different processors.
- Servers without aperiodic tasks are known as empty servers. If an empty server is selected by the scheduler, the designated processor does nothing or comes into idle times.
- When a non-empty server is selected by the scheduler, the aperiodic task assigned to that server is executed on the designated processor.

Algorithm 4 describes procedures how aperiodic tasks are assigned to servers. Accordingly, two procedures are proposed for two events: aperiodic release and aperiodic completion. An event of aperiodic release is realized as an aperiodic task enters the system. The first procedure, *aperiodicRelease*, is invoked at this event to look for a server for the task. If there exists an empty server in the system, the released task is assigned to the empty server through a process *assignTask2Server*. The process *assignTask2Server* requires two arguments: *sid* is the identifier of the empty server and *Entry* is a pointer to the released task. In case that there is no empty server, the released task is added to the aperiodic ready queue ARQ in order to wait for serving. Aperiodic tasks are organized in ARQ in the manner of first-come-first-serve (FCFS).

On the other hand, an event of aperiodic completion is realized as an aperiodic task is completed its execution on a server. The second procedure, *aperiodicComplete*, is invoked at this event to seek for another aperiodic task from ARQ. Process *seekTaskFromReadyQueue* is introduced for the task seeking in ARQ. If a *task* is found, it is assigned to

the server via process *assignTask2Server* with input *sid* is the identifier of the server on which the completion event happens. When assigned, the aperiodic tasks can be executed on the designated processor. Otherwise, if ARQ is empty, the server becomes empty and the designated processor changes to idle time.

By this scheme, we convert the problem of scheduling aperiodic tasks to the problem of scheduling servers. The schedule of aperiodic tasks is therefore obtained from the schedule of servers.

Algorithm 4: Rule of assigning aperiodic tasks to servers

```
1: Aperiodic task release
   aperiodicRelease(Entry)
   {
      if (hasEmptyServer())then
      {
         sid ← getEmptyServerID();
         assignTask2Server(sid, Entry);
      }
      else
         addTask2ReadyQueue(ARQ, Entry);
   }
2: Aperiodic task completion
   aperiodicComplete(sid)
   {
      task ← seekTaskFromReadyQueue();
      if (task! = NULL)then
         assignTask2Server(sid, task);
      else
         setServerEmpty(sid);
   }
```

6.2.3 Consideration of acceptance test for aperiodic tasks

Acceptance test is a quite popular concept in scheduling context of mixture task sets. In order to assure the system feasible for scheduling, acceptance test is introduced to decide that an aperiodic task is accepted or rejected to enter the system. The assignment of aperiodic tasks to servers in Algorithm 4 implicitly implicates that aperiodic tasks are decided, at their release time, to exist in the system in one of two states:

1. Being assigned to a server or
2. Being waiting in the aperiodic waiting queue.

At runtime, an aperiodic task can be changed from the waiting state to the assigned state when it is moved to a server. As a result, the acceptance test for aperiodic tasks is not needed in our scheduling scheme.

6.3 Definitions of LAA+

Basically, definitions of LAA+ are similar to those of the original LAA. The difference is calculations extended for servers.

6.3.1 Time interval

Time interval is defined as the same as that in the original LAA; that is, the time period between any two consecutive periodic job releases is considered as one time interval. The starting times of intervals are considered as the scheduling events when the scheduler is invoked to make scheduling plans for the intervals. Since the end time of an interval is also the start time of the next one, start and end times of intervals are scheduling events. Then, a time interval $I = [t_1, t_2)$ has a length $L_I = t_2 - t_1$, where $t_1, t_2 \in N$ and $0 \leq t_1 < t_2$.

6.3.2 Proportionate scheduling

In the context of servers and periodic tasks, the notion of proportionate scheduling is exploited to calculate the local requested execution times in the following manner: By every time t as the end of an interval, a task τ_i and a server S_j must have been scheduled to receive at least $\lfloor \mu_i * t \rfloor$ and $\lfloor U_j * t \rfloor$ resources, respectively. In other words, using its utilization rate, a server is treated like a periodic task in LAA+. Let $S_i(t)$ and $Ss_j(t)$ denote resources that periodic task τ_i and server S_j should be received up to t. $S_i(t)$ is calculated using Equation 4.1. $Ss_j(t)$ is obtained using Equation 6.2.

$$Ss_j(t) = \lfloor U_j * t \rfloor \tag{6.2}$$

6.3.3 Local requested execution time

Local requested execution time (LRET) is amounts of time slots that periodic tasks and servers receive on an interval. Consider interval $I = [t_1, t_2)$ with its length $L_I = t_2 - t_1$. The LRETs of periodic tasks and servers are denoted as $E_i(t_1, t_2)$ and $Es_j(t_1, t_2)$, respectively. Figure 6.1 shows the structure of LRETs for periodic tasks and servers. LRETs consists of two parts: mandatory execution and extra execution. For periodic tasks, mandatory execution ($M_i(t_1, t_2)$) is calculated the same as that in the original LAA using Equation 4.4. The extra execution of periodic tasks ($P_i(t_1, t_2)$) is divided into two portions: P_i' is a unit addition (resulting in 0 or 1) and P_i'' is calculated using Equation 6.3.

$$P_i'' = min(JRE_j, slack, L_I - M_i - P_i') \tag{6.3}$$

For servers, mandatory executions are calculated based-on the proportionate scheduling as Equation 6.4. Extra executions for servers are obtained from the slack times using Equation 6.5.

$$Ms_j(t_1, t_2) = \lfloor U_j * t_2 \rfloor - As_j(t_1) \tag{6.4}$$

$$Ps_j(t_1, t_2) = min(slack, L_I - Ms_j) \tag{6.5}$$

In calculation of LRETs, $A_i(t_1)$ and $As_j(t_1)$ are the already-assigned resources of task τ_i and server S_j on the entire period from 0 up to t_1. Such already-assigned resources are easily recorded by the operating system.

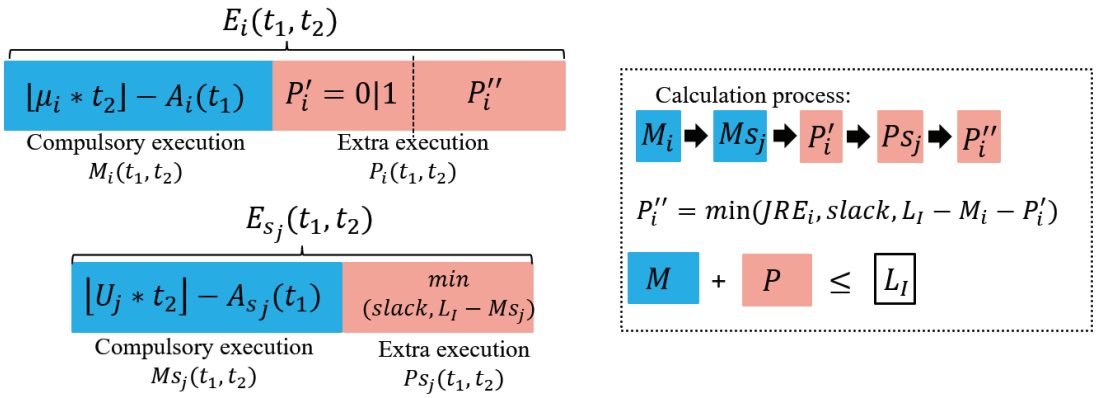

Figure 6.1: Local requested execution time of LAA+

Due to the limit of the system capacity on interval I, LRETs of periodic tasks and servers follows the following constraints:

$$M_i(t_1, t_2) + P_i(t_1, t_2) \leq L_I \tag{6.6}$$

$$Ms_j(t_1, t_2) + Ps_j(t_1, t_2) \leq L_I \tag{6.7}$$

These constraints guarantee for LRETs to be provided within the interval. The calculation process of parts of LRETs (also described later in detail in Section 6.4) is as follows: mandatory execution of periodic tasks ($M_i(t_1, t_2)$), mandatory execution of servers ($Ms_j(t_1, t_2)$), unit addition of periodic tasks (P_i'), extra execution of servers ($Ps_j(t_1, t_2)$), and remaining extra execution of periodic tasks (P_i'').

Finally, we can obtain the total local requested execution time on the interval, denoted as $E(t_1, t_2)$, using Equation 6.8.

$$E(t_1, t_2) = \sum_{i=0}^{n_p-1} E_i(t_1, t_2) + \sum_{j=0}^{n_s-1} Es_j(t_1, t_2) \tag{6.8}$$

6.3.4 Fully-assigned system

Similarly to the definition in the original LAA, fully-assigned system is defined as the system situation in which all time slots are assigned to periodic tasks and aperiodic servers for executions. As a result, if a system is considered as fully-assigned up to time t, the total amount of already-assigned resources up to t is equal to the capacity of the system. That is,

$$\sum_{i=0}^{n_p-1} A_i(t) + \sum_{j=0}^{n_s-1} As_j(t) = m * t \tag{6.9}$$

6.4 Procedure of LAA+

6.4.1 LAA+ algorithm

LAA+ is extended from the original LAA to the context of periodic tasks and servers. **Algorithm 5** shows the algorithm of LAA+ for making scheduling plans on intervals. In this pseudo code, m denotes the number of processors. Periodic tasks and servers are indexed by i and j, respectively, where $0 \leq i < n_p$ and $0 \leq j < n_s$. μ_i and U_j are the utilizations of periodic tasks and servers. JRE_i is the remaining execution time of the current job of task τ_i. Related to the interval information, t_s and t_e ($0 \leq t_s < t_e$) as normal indicate the start and end times of the involved interval on which the scheduling plan is applied. L_I is the interval's length, which is equal to $t_e - t_s$. LRETs of periodic tasks and servers on the interval are expressed as $E_i(t_s, t_e)$ and $Es_j(t_s, t_e)$, respectively. $A_i(t_s)$ and $As_j(t_s)$ are the already-assigned resources of periodic tasks and servers up to t_s (from 0 to t_s). Finally, $min(a, b)$ and $min(a, b, c)$ are functions that find the smallest value from their input arguments.

As shown in Algorithm 5, similar to the original LAA, LAA+ has three main steps: LRET Estimation, LRET Adjustment and Task Assignment. The first step, *LRET Estimation*, is to calculate the mandatory executions (M_i and Ms_j as defined in Section 6.3) for periodic tasks and servers on the interval. Equation 4.4 and Equation 6.4 are used for this calculation in which t_1 and t_2 are correspondingly replaced with t_s and t_e. In the second step, *LRET Adjustment*, LRETs are increased to exhaust the system capacity of the involved interval. To this end, the slack time, denoted as *slack*, is computed by subtracting LRETs obtained from the first step from the system capacity. *slack* is then distributed to periodic tasks and servers appropriately as their extra execution. First, LRETs of periodic tasks are increased by one unit execution (P'_i as defined in Section 6.3) if they have execution remaining and their already-assigned LRETs are less than the interval's length. Importantly, this unit extra execution guarantees the fairness for periodic tasks (similar approach is used in BF scheduling [35]).

Then, the extra execution (Ps_j as defined in Section 6.3) for servers are calculated. Accordingly, LRETs of servers are incremented by the smaller value between the remaining *slack* and the possible increment of the server on the interval defined by $L_I - Es_j(t_s, t_e)$. The possible increment implicitly indicates that LRETs of servers do not exceed the interval's length. The different ways of adjustment applied to periodic tasks and servers imply a preference for servers to increase their LRETs more than periodic tasks. This is aimed at improving the aperiodic responsiveness.

At the end of the adjustment step, the remaining *slack* is continuously distributed to periodic tasks. Since the system capacity is fully consumed, the LRET Adjustment guarantees the fully-assigned system on the interval. Therefore, the following result is obtained:

$$\sum_{i=0}^{n_p-1} E_i(t_s, t_e) + \sum_{j=0}^{n_s-1} Es_j(t_s, t_e) = m * (t_e - t_s) \tag{6.10}$$

In the final step, *Task Assignment*, periodic tasks and servers are arranged on processors for their execution. A process *consecutiveAssignment*, which is described in detail in **Algorithm 6**, is proposed for this purpose.

Algorithm 5: Enhanced Local Assignment Algorithm (LAA+)

i, j : periodic task and server indexes
t_s, t_e : the start and end times of the interval
$L_I \leftarrow t_e - t_s$;
1. LRET Estimation
For each task τ_i and each server S_j:
$E_i(t_s, t_e) \leftarrow \lfloor \mu_i * t_e \rfloor - A_i(t_s)$;
$Es_j(t_s, t_e) \leftarrow \lfloor U_j * t_e \rfloor - As_j(t_s)$;
$JRE_i \leftarrow JRE_i - E_i(t_s, t_e)$;
2. LRET Adjustment
$slack \leftarrow m * L_I - \sum E_i(t_s, t_e) - \sum Es_j(t_s, t_e)$;
while $slack > 0$ **do**
{
For each task τ_i:
if $(slack > 0$ & $JRE_i > 0$ & $E_i(t_s, t_e) < L_I)$
{
$E_i(t_s, t_e) \leftarrow E_i(t_s, t_e) + 1$;
$JRE_i \leftarrow JRE_i - 1$;
$slack \leftarrow slack - 1$;
}
For each server S_j:
if $(slack > 0)$
{
$min \leftarrow min(slack, L_I - Es_j(t_s, t_e))$;
$Es_j(t_s, t_e) \leftarrow Es_j(t_s, t_e) + min$;
$slack \leftarrow slack - min$;
}
For each task τ_i:
if $(slack > 0$ & $JRE_i > 0$ & $E_i(t_s, t_e) < L_I)$
{
$min \leftarrow min(slack, JRE_i, L_I - E_i(t_s, t_e))$;
$E_i(t_s, t_e) \leftarrow E_i(t_s, t_e) + min$;
$JRE_i \leftarrow JRE_i - min$;
$slack \leftarrow slack - min$;
}
}
3. Task Assignment
consecutiveAssignment();

6.4.2 Consecutive assignment of LAA+

A simple model of consecutive assignment was introduced in DP-WRAP to assign periodic tasks to processors [37]. This assignment model actually follows the McNaughton's wrap algorithm [53]. In this study, we come up with a more complex procedure of the consecutive assignment toward improving the aperiodic responsiveness as well as alleviating task preemption and migration. **Algorithm 6** shows the procedure of consecutive assignment of LAA+. The consecutive assignment process is conducted with the following

input data: $Tasks$ and $Servers$ are data structures as lists of periodic tasks and servers existing in the system; $E(t_s, t_e)$ and $Es(t_s, t_e)$ are LRETs of periodic tasks and servers obtained in the first two steps of Algorithm 5; m is the number of processors; and L_I is the interval's length. On the other hand, SP is a data structure that is used to store the expected scheduling plans on the interval. As a result, SP is the output of this process. Additionally, it is supposed that the record of past schedules on processors is available for retrieval.

Algorithm 6: Consecutive Assignment of LAA+

Input: $Tasks$, $Servers$, $E(t_s, t_e)$, $Es(t_s, t_e)$, m, L_I
Output: SP

```
consecutiveAssignment()
{
   UP ← establishUniProcessor(m, L_I);
   For each segment p from 0 to m-1:
         len[p] ← L_I;
   UP ← assignNonEmptyServer(UP, Servers, Es(t_s, t_e));
   p ← 0;
   while p < m & hasUnassignedTask() do
   {
      if (len[p] == 0)
         p ← p + 1;
      else
      {
         tsk ← seekTaskFit(p, Tasks, E(t_s, t_e));
         if (tsk == NULL)
            {
               tsk ← seekTaskLast(p + 1, Tasks, E(t_s, t_e));
               if (tsk == NULL)
                  tsk ← seekTaskRandom(Tasks);
            }
         UP ← assignTask(UP, p, tsk, E(t_s, t_e));
      }
   }
   UP ← assignEmptyServer(UP, Servers, Es(t_s, t_e));
   SP ← scheduleConverting(SP, UP);
}
```

The basic idea of arrangement is as the same as the original LAA algorithm in which a virtual uniprocessor is used. Modifications include (1) the order of allocating periodic tasks and servers and (2) the selection of periodic tasks for allocation. Specifically, a virtual uniprocessor is established first based on the number of processors (m) and the interval's length (L_I). Actual processors are joined to make a row on the virtual uniprocessor and Each of them corresponds to a segment of length L_I. The capacity of the virtual uniprocessor is therefore equal to $m * L_I$. Then, periodic tasks and servers are consecutively allocated on the virtual uniprocessor by the amounts of their LRETs. Since the total amount of LRETs is not greater than the system capacity on the interval, the

allocation on the virtual uniprocessor is achievable. Finally, the arrangement on each actual processor is given by converting (or extracting) the allocation on the corresponding segment on the virtual uniprocessor.

The detail of the consecutive assignment of LAA+ is as follows. The virtual uniprocessor, expressed as UP, is first generated using function $establishUniProcessor$. Each segment on UP is initiated with length L_I. The allocation on the virtual uniprocessor starts with non-empty servers first followed by periodic tasks and then empty servers. All non-empty servers, in the descending order of utilization rates, are assigned to the beginning of segments using function $assignNonEmptyServer$. Note that since the number of servers established with Algorithm 3 is less than or equal to m, the allocation of non-empty servers is possible. Servers are each given spaces equal to their LRET. When segments take in servers' requirement, their lengths are decreased by the amount.

Then, a loop is implemented to assign periodic tasks. Periodic tasks are selectively allocated in the following priority:

1. Task that has its LRET fitting to the remaining space of a segment;
2. Task that was executed on the next segment's processor at the slot just before the scheduling event;
3. Task selected randomly.

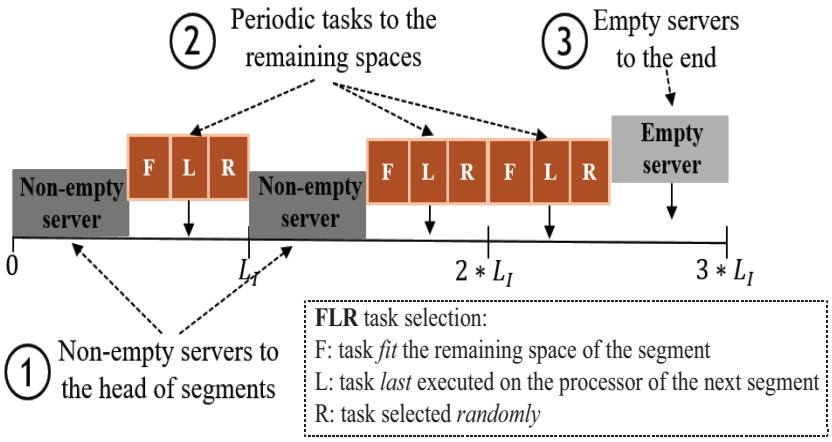

Figure 6.2: Task selection for processor allocation in LAA+

Figure 6.2 illustrates the priority of task and server selection. The first priority is suggested with an attempt to reduce the preemption while the second priority is to alleviate the migration. Three functions $seekTaskFit$, $seekTaskLast$, and $seekTaskRandom$ are introduced for the task selections. Function $seekTaskFit$ seeks a task the LRET of which is non-zero and fits with the remaining space of the specified segment. Function $seekTaskLast$ seeks a task that has just been executed on the corresponding processor of the specified segment and has a non-zero LRET. This function is supported with the schedule history at slot $t_s - 1$, the slot just before the scheduling event. Function $seekTaskRandom$ arbitrarily seeks a task that has non-zero LRET.

The selected task tsk is then assigned to the current processor's segment specified by p using function $assignTask$. The task is allocated space continuously after the already

assigned servers and tasks on the same segment. The given space is equal to the task's LRET and may belong to two segments (p and $p+1$). In this case, the already assigned server on $p+1$ is moved to give a fit space to the task. When segments take in task's requirement, their lengths are decreased by the amount.

Next, empty servers are continuously allocated to the remaining space on the segments. Assigning empty servers lastly has a meaning that the empty servers are preserved for the coming aperiodic tasks. As a result, it has potential to reduce unnecessary idle times of processors, which effectively improves the system performance.

Finally, the schedule on actual processors is generated by converting the order of periodic tasks and servers on the virtual uniprocessor using function *scheduleConverting*. This function maps the order of periodic tasks and servers on the segments of the virtual uniprocessor to the execution order on the actual processors. As the arrangement process is finished, SP is acquired as the expected scheduling plans for the interval. At runtime, SP is used as instruction for selecting tasks and deciding context switches during the interval.

6.4.3 Example of scheduling with LAA+

An example of scheduling periodic tasks and servers is displayed in Figure 6.3. In this example, there are four periodic tasks scheduled on a system of three identical processors. Periodic tasks have the same utilization of 0.6 and periods of 5, 10, 15, and 10, respectively. Using the unused capacity from periodic tasks, one server is established with its utilization of 0.6. Based on periodic tasks' periods, three job releases are identified at 0, 5, and 10 after the first ten slots. Therefore, two intervals (denoted as I_0 and I_1) are formed for this period. For the first interval, the scheduler is invoked at time 0 to make scheduling plans on processors. Executing Algorithm 5, calculations of LRET Estimation give periodic tasks and server three slots each as expressed by E_0. At the next step of LERT Adjustment, since *slack* is calculated equal to 0, no extra execution is added to LRETs. Eventually, each of periodic tasks and server receives three slots during I_0.

Next, at the step of Task Assignment, a virtual uniprocessor is established for I_0 with the capacity of 15. Because there is no non-empty server, the assignment process starts with periodic tasks. At time 0, no schedule history is available, periodic tasks are therefore allocated in a row on the virtual uniprocessor. As shown in 6.3, the order of periodic tasks is: τ_0, τ_1, τ_2, and τ_3. Lastly, the empty server is finally allocated at the end of the virtual uniprocessor.

Finally, based on the segments of processors on the virtual uniprocessor, scheduling plans of periodic tasks and server on actual processor are obtained. That is, $P0$ will execute τ_0 for three slots, then execute τ_1 for two slots; $P1$ will execute τ_1, τ_2 and τ_3 for one, three and one slots, respectively; $P2$ will execute τ_3 for two slots and give the remaining three slots for the empty server. The final scheduling plan of I_0 is found in the figure.

For the second interval, the scheduler is invoked at time 5 to make scheduling plans. Similarly, calculations of LRETs eventually give periodic tasks and server three slots each as expressed by E_1. These assigned LRETs are used for task assignment. After establishing virtual uniprocessor for I_1, the task assignment starts with periodic tasks since the server is still empty. τ_0 is assigned to $P0$' segment first as a fit task to the segment's capacity. Then, τ_3 is allocated as it has been executed at slot 4 on the second

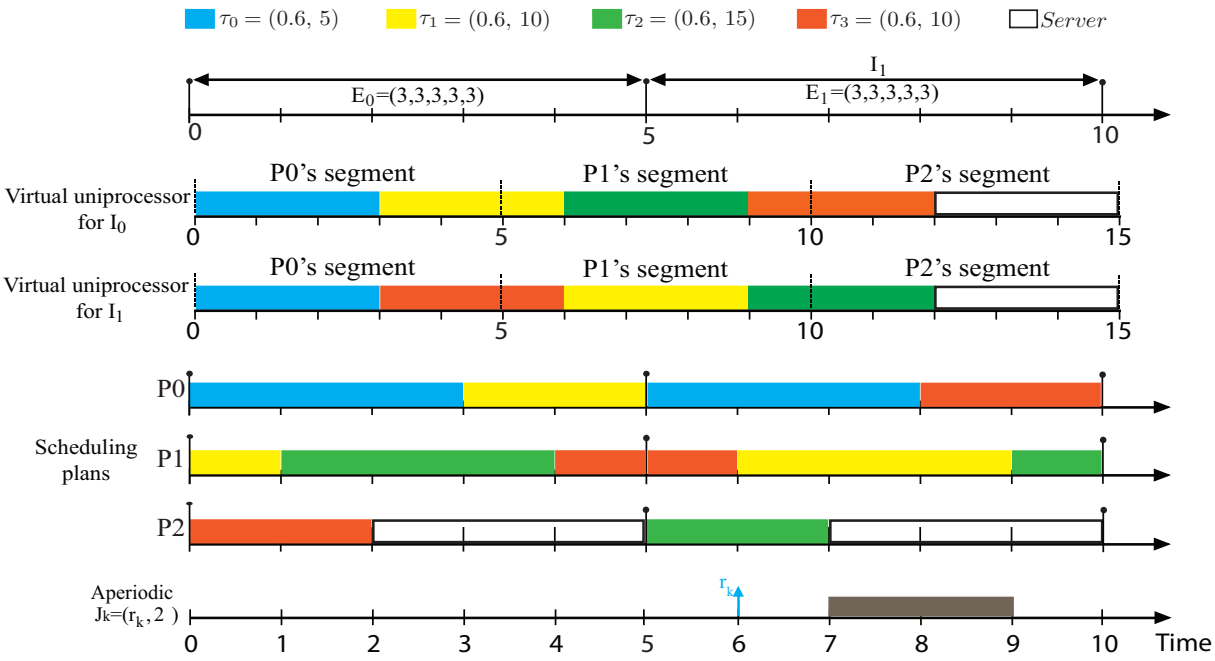

Figure 6.3: Example of scheduling with LAA+

processor $(p+1)$. τ_3 occupies the remaining two slots of $P0$'s segment and one slot of $P1$'s. Then, as a fit task to the remaining space of $P1$'s segment, τ_1 is assigned the space after τ_3. τ_2 is allocated space after τ_1 on the virtual uniprocessor. Finally, the empty server is assigned to the end of the virtual uniprocessor. The final scheduling plans of I_1 is obtained by the converting process as shown in the figure.

It is supposed that an aperiodic task enters the system at time 6 with its worst-case execution time of 2. The aperiodic task is assigned to the empty server and then executed on slot 7 and slot 8 as the server is scheduled. The aperiodic task is finished at time 9 with the response time of 3.

6.4.4 Secondary scheduling event

In the scheduling scheme introduced above, the scheduling events are decided as the release times of (jobs of) periodic tasks. We call them primary scheduling events (primary invocations). In this section, we suggest to add secondary scheduling events (secondary invocations). A secondary scheduling event is decided as the time within an interval when an aperiodic task is released and there exist empty servers. At that time, the aperiodic task is assigned to an empty server and the server becomes non-empty. Looking back at the example in Figure 6.3, time 6 can be considered as a secondary scheduling event since at that time the aperiodic task is released and the server is empty. The secondary scheduling event is introduced to reorder the non-empty servers' execution on the remaining span of the interval. Since the consecutive assignment in Algorithm 6 tends to assign non-empty servers ahead of periodic tasks on every processor, the reordering action has potential to improve the aperiodic responsiveness without violating scheduling constraints for periodic tasks.

At the secondary invocation, the calculation of LRETs (the first and second steps of

Algorithm 5) is not needed because LRETs remaining on the interval can be utilized. Step 3 of Algorithm 5, *Task Assignment*, is required only to make up the scheduling plan. Obviously, the secondary invocations add less runtime overhead than the primary ones in spite of increasing the number of scheduling points.

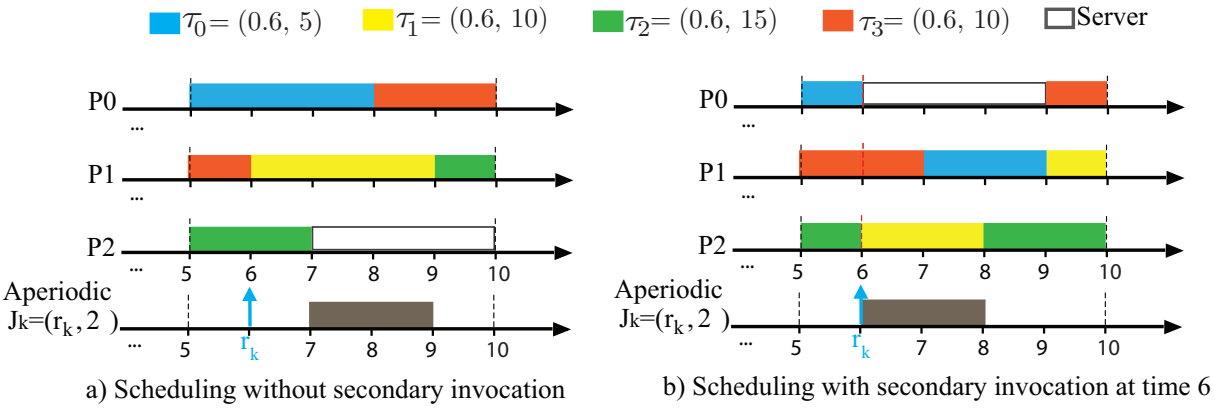

Figure 6.4: Example of scheduling with secondary invocation

Let us observe the example in Figure 6.3. When the aperiodic task is released, it must wait for execution up to time 7 as the server is scheduled. Now, consider time 6 as a secondary scheduling event, the scheduler is invoked to make up the scheduling plan for the remaining time span from time 6 to 10. Figure 6.4 shows the results of the reordered scheduling plan from the secondary invocation. In Figure 6.4(a) without secondary invocation, the aperiodic task is finished at time 9 with the response time of 3. In Figure 6.4(b), the scheduling plan is made up at time 6, which arranges the non-empty server at the beginning of $P0$. As a result, the aperiodic task can be executed immediately and then finished at time 8, one slot earlier.

6.5 Schedulability guarantee of LAA+

In this section, we show that LAA+ in Algorithm 5 can guarantee the schedulability of up to 100%. The theoretical proof is provided concisely with a lemma which presents the schedulability of an individual interval. Then, the theorem for the schedulability of the whole system is proven.

Lemma 6.5.1. *Given an interval $I = [t_1, t_2)$ $(0 \leq t_1 < t_2)$. If the system is fully assigned up to t_1, the system is then schedulable on I when using Algorithm 5 as the scheduling algorithm.*

Proof. Applying definitions in Section 6.3 and Algorithm 5, the total mandatory execution of periodic tasks and servers is obtained as follows:

$$\sum_{i=0}^{n_p-1} M_i(t_1,t_2) + \sum_{j=0}^{n_s-1} Ms_j(t_1,t_2) = \sum_{i=0}^{n_p-1} \lfloor \mu_i * t_2 \rfloor - \sum_{i=0}^{n_p-1} A_i(t_1) + \sum_{j=0}^{n_s-1} \lfloor U_j * t_2 \rfloor - \sum_{j=0}^{n_s-1} As_j(t_1) \tag{6.11}$$

Since the system is fully assigned up to time t_1, the total already assigned resources on $[0, t_1)$ is equal to the system capacity. Namely, Equation 6.9 holds on $[0, t_1)$. Then, we can derive the following result:

$$\sum_{i=0}^{n_p-1} M_i(t_1, t_2) + \sum_{j=0}^{n_s-1} Ms_j(t_1, t_2) = \sum_{i=0}^{n_p-1} \lfloor \mu_i * t_2 \rfloor + \sum_{j=0}^{n_s-1} \lfloor U_j * t_2 \rfloor - m * t_1 \tag{6.12}$$

Additionally, using Equation 6.1, we obtain the following result:

$$\sum_{i=0}^{n_p-1} M_i(t_1, t_2) + \sum_{j=0}^{n_s-1} Ms_j(t_1, t_2) \leq m * t_2 - m * t_1 \tag{6.13}$$

On the other hand, according to Algorithm 5, the total extra execution of periodic tasks and servers is equal to the slack time on the interval after calculating the mandatory executions. In the other words, the total extra execution can be calculated as follows:

$$\sum_{i=0}^{n_p-1} P_i(t_1, t_2) + \sum_{j=0}^{n_s-1} Ps_j(t_1, t_2) = m * (t_2 - t_1) - \sum_{i=0}^{n_p-1} M_i(t_1, t_2) + \sum_{j=0}^{n_s-1} Ms_j(t_1, t_2) \tag{6.14}$$

Combining Equation 6.13 and Equation 6.14, we have:

$$E(t_1, t_2) = \sum_{i=0}^{n_p-1} E_i(t_1, t_2) + \sum_{j=0}^{n_s-1} Es_j(t_1, t_2) = m * (t_2 - t_1) \tag{6.15}$$

Equation 6.15 guarantees that all LRETs are accommodated within time the span of the time interval. Hence, the schedulability is guaranteed on interval $I = [t_1, t_2)$. Furthermore, the system is fully assigned on the interval. □

Theorem 6.5.2. *Given $D = \{0, d_1, d_2, d_3, ...\}$ $(d_1, d_2, d_3, ... \in Z^+$ and $0 < d_1 < d_2 < d_3 <)$ as the set of primary scheduling events. Then, using Algorithm 5 as the scheduling algorithm guarantees that the whole system is schedulable.*

Proof. Let $I_0 = [0, d_1)$, $I_1 = [d_1, d_2)$, $I_2 = [d_2, d_3)$,... be time intervals generated in the system based on job releases. Applying Algorithm 5, the total mandatory execution of periodic tasks and servers on I_0 is calculated as follows:

$$\sum_{i=0}^{n_p-1} M_i(0, d_1) + \sum_{j=0}^{n_s-1} Ms_j(0, d_1) = \sum_{i=0}^{n_p-1} \lfloor \mu_i * d_1 \rfloor - \sum_{i=0}^{n_p-1} A_i(0) + \sum_{j=0}^{n_s-1} \lfloor U_j * d_1 \rfloor - \sum_{j=0}^{n_s-1} As_j(0) \tag{6.16}$$

Conventionally, no resource has been assigned to periodic tasks and servers at time 0. That is,

$$\sum_{i=0}^{n_p-1} A_i(0) + \sum_{j=0}^{n_s-1} As_j(0) = 0 \tag{6.17}$$

Therefore, using Equation 6.1 the total mandatory execution on I_0 is derived as follows:

$$\sum_{i=0}^{n_p-1} M_i(0, d_1) + \sum_{j=0}^{n_s-1} Ms_j(0, d_1) \leq m * d_1 \tag{6.18}$$

The total extra execution of periodic tasks and servers on I_0 is calculated as:

$$\sum_{i=0}^{n_p-1} P_i(0,d_1) + \sum_{j=0}^{n_s-1} Ps_j(0,d_1) = m * d_1 - \sum_{i=0}^{n_p-1} M_i(0,d_1) + \sum_{j=0}^{n_s-1} Ms_j(0,d_1) \quad (6.19)$$

Combining Equation 6.18 and Equation 6.19, we can obtain the total LRETs on I_0 as follows:

$$E(0,d_1) = m * d_1 \quad (6.20)$$

Equation 6.20 indicates that the system is schedulable and the fully-assigned system is achievable on I_0.

Since the system is fully assigned on I_0 (or up to d_1), according to Lemma 6.5.1 the system is schedulable on I_1 using Algorithm 5. The fully-assigned system is then maintained up to d_2.

This induction is repeated for the successive intervals. The schedulability is therefore guaranteed for the whole system. □

6.6 Evaluation of LAA+

6.6.1 Simulation environment

Evaluation criteria

LAA+ is evaluated by simulation. The targeted evaluation criteria are responsiveness, time complexity, and scheduler invocation. The calculation methods of these criteria are the same as those in the original LAA (see Section 4.2 for definitions in detail).

Simulations are observed during a period of 100,000 ticks.

Task set

Similar to the evaluation of the original LAA, the simulation environment of LAA+ is set up for cases of 4, 8, 16, and 32 identical processors. Task sets are generated as mixture sets of periodic and aperiodic tasks. The utilization of periodic tasks is from 75% to 95% with increment of 5%. The maximum number of periodic tasks in a task set is 100. Generation of periodic task sets is similar to that in the evaluation of the original LAA (see Section 4.6.1).

The utilization of aperiodic tasks is from 2% to 5%. For instance, during the observation period of 100,000 ticks, the execution of aperiodic tasks on the case of 4 processors will totally occupy from 8,000 to 20,000 ticks corresponding to the utilizations from 2% to 5%. The number of aperiodic releases in a task set is at least 3,000 for the observation period of 100,000 ticks. For each scenario of the periodic task utilization and the number of processors, for example periodic utilization of 75% and 4 processors, all combinations of ten periodic task sets and twenty aperiodic task sets (totally 200 combinational sets) are simulated and the average result is exhibited.

Comparison candidate

The effectiveness of LAA+ is assessed in comparison with the original LAA, Pfair [32], RUN [38], and semi-partition reservation (SPR) [29, 30]. In order to schedule aperiodic

tasks, model of aperiodic servers in [49] is applied for Pfair. On the other hand, the aperiodic tasks are scheduled in background of periodic tasks in the original LAA, RUN and SPR so that they are executed at unused slots (or idle times) of processors.

6.6.2 Simulation results of LAA+

Responsiveness

Figure 6.5 shows results of average response times of aperiodic tasks. LAA likely shows the worst results since it prioritizes periodic tasks to receive seamless executions ahead of aperiodic tasks (scheduled in background) on every interval. On the case of 4 processors, LAA+ clearly shows the best results. Specifically, LAA+ reduces the response times of LAA by about 30% in average. The corresponding reductions are about 10% compared with PFAIR and about 7% compared with RUN and SPR. LAA+ maintains the similar reduction rates compared with LAA in the other cases of processors. However, when the number of processors increases, results of LAA+ become overlapping with that of RUN while still maintain better than PFAIR and SPR in almost cases. (Additionally as our observing, numerical results of simulations exhibit that, except cases of 95%, LAA+ is not worst than RUN in terms of response time.) The improvement gap of LAA+ is decreased with the increase of processors since the number of empty slots increases together with the number of processors and empty processors tend to arise more frequently. Exceptionally, at utilization of 95% on large number of processors, LAA+ tends to be worse than RUN and SPR. The reason is that schedules of servers (and then the aperiodic tasks' execution) become fragmentary at heavy loads due to the proportionate calculation.

Time complexity

Time complexity is assessed through the maximum runtime overhead per tick. Figure 6.6 shows results for the runtime overhead. Due to the complex procedure of task classification, Pfair causes significantly higher runtime overhead than the other algorithms in our all cases of simulation. Therefore, it is not plotted in the figure. Table 6.1 additionally depicts the average number of operations invoked in algorithms' execution at $U_p = 95\%$ on cases of 4 and 16 processors. Using information in Table 6.1 can figure out how invoked operations contribute to the runtime overheads.

Overall, as a semi-partitioned scheduling, SPR shows the lowest runtime overhead. Except case of periodic utilizations of 80% and 85% on 4 processors, LAA+ exhibits lower runtime overheads than RUN. Accordingly, LAA+ has about 14% of runtime overhead lower than RUN for heavy task sets at utilizations of 90% and 95% on 4 processors; the average reduction rates for cases of 8, 16 and 32 processors are 12%, 16% and 20%, respectively. Obviously, including servers the calculation of LAA+ becomes more complexed than LAA, which results in higher runtime overhead for most of the cases. Exceptionally, at $U_p = 80\%$ and $U_p = 95\%$ on 32 processors, LAA+ show lower runtime overheads than LAA. This exception can be reasoned that the calculation of servers' LRETs at step 1 in Algorithm 5 may occasionally decrease slack time on step 2 and therefore reduce the calculation complexity at step 2. The results indicate that LAA+ can maintain the optimality and improve the aperiodic responsiveness of LAA without significantly increasing the time complexity.

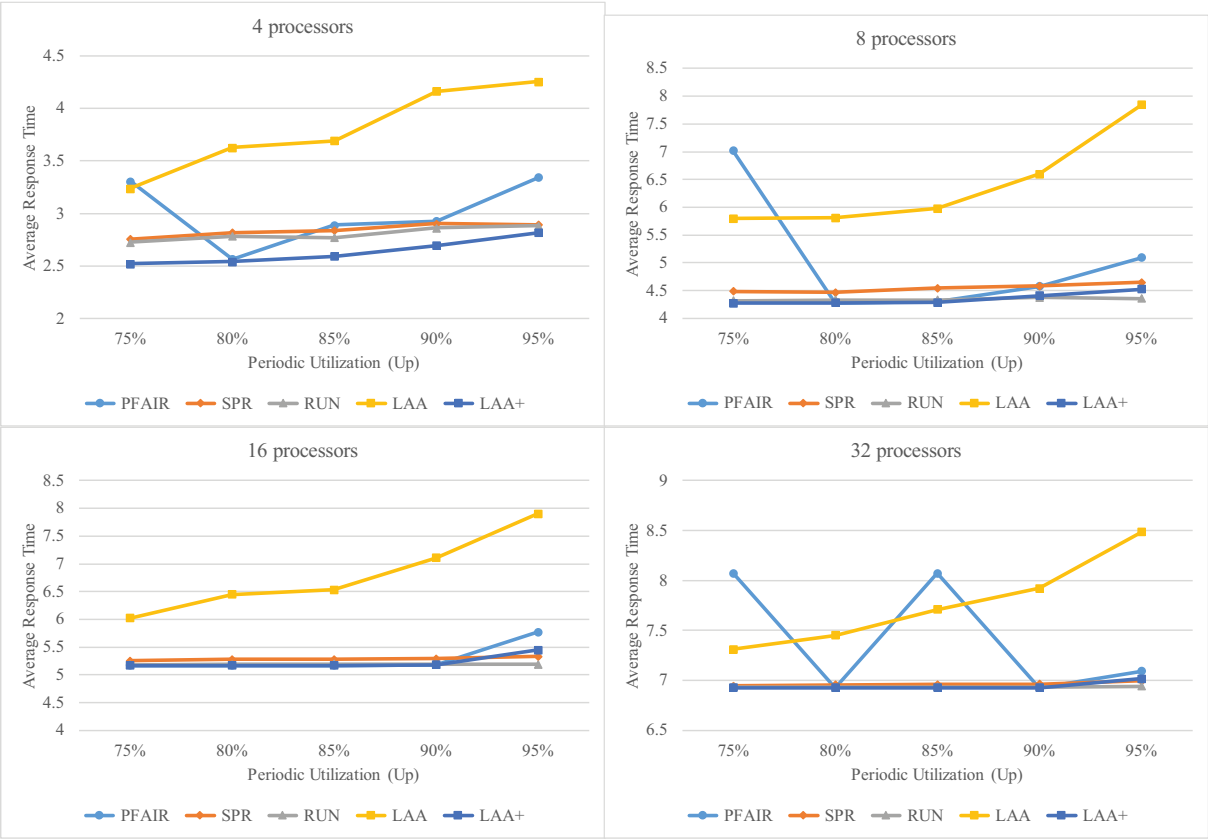

Figure 6.5: Response time of LAA+

Scheduler invocation

Figure 6.7 displays results for the number of scheduler invocations. With introduction of secondary invocations, LAA+ causes a bit larger number of scheduling points than the original LAA. Overall, LAA and LAA+ outperform the other candidates in this criterion by over 50%. Combining with lower runtime overheads, fewer scheduler invocations also unveil efficiency of our solution in terms of improving the accumulative runtime overhead. This can improve the system performance since processors are preferably utilized for tasks' execution rather than executing the scheduling algorithm.

6.7 Conclusion: Effectiveness and Limitation of LAA+

6.7.1 Effectiveness of LAA+

In this chapter, we presented an effective solution called LAA+ for the problem of scheduling hybrid task sets of periodic and aperiodic tasks. The proposed solution exploited the Local Assignment Algorithm with the integration of aperiodic servers.

Aperiodic servers are inserted into the system with their characteristics alike periodic tasks. This approach allows Local Assignment Algorithm to be utilized effectively to schedule the context of periodic tasks and servers. The aperiodic tasks then get scheduled through the preservation of servers. LAA+ still maintains the optimality of the Local

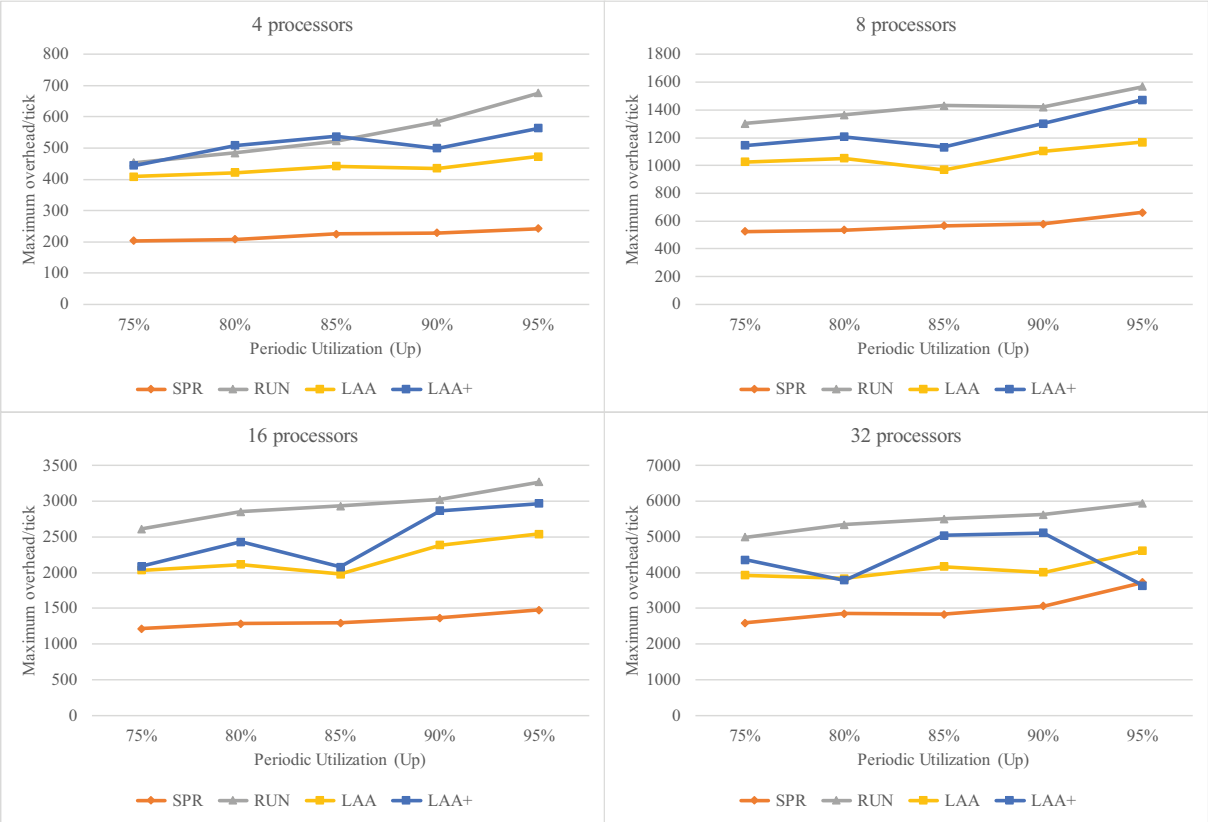

Figure 6.6: Runtime overhead of LAA+

Assignment Algorithm without significantly increasing the time complexity.

LAA+ introduces secondary scheduling events and a selective model of consecutive task assignment. These techniques effectively reduce the response times of aperiodic tasks while still guaranteeing the schedule of periodic tasks. As shown in simulation results, LAA+ achieves responsiveness equivalent to the existing algorithms with extremely less number of scheduling points.

6.7.2 Limitation of LAA+

LAA+ has the same limitations with the original LAA. That is, the system is actually not work-conserving within intervals. The processor times preserved for periodic tasks and servers based on scheduling plans are unusable by the other tasks with work remaining.

Like the original LAA, LAA+ is introduced to schedule task sets of periodic and aperiodic tasks. Future works are considered to extend LAA+ to deal with other type of tasks such as sporadic one.

Table 6.1: Number of operations in execution of LAA+

	4 processors				
	PFAIR	SPR	RUN	LAA	LAA+
Overhead	**2994.3**	**250.7**	**675.2**	**472.4**	**568.3**
IADD	430.2	14.1	42.4	55.6	32.76
IMUL	0	0	0	1	1
FMUL	340	0	8.5	7.9	8.8
COMP	429.1	162.5	341.8	201.7	250.5
ASSIGN	283.9	74.1	248.5	166.3	231.2
FLOOR	151.1	0	0	6.9	7.8
	16 processors				
	PFAIR	SPR	RUN	LAA	LAA+
Overhead	**24029.9**	**1061.6**	**3271.5**	**2416.2**	**2971.3**
IADD	5615.2	52.54	211.9	133.9	115.3
IMUL	0	0	0	1	1
FMUL	512.8	0	29.4	29.2	36.9
COMP	8885	1061.6	1757.8	1456.9	1614.6
ASSIGN	5017.8	360.6	1153.7	605.9	1020.5
FLOOR	151.1	0	0	6.9	7.8

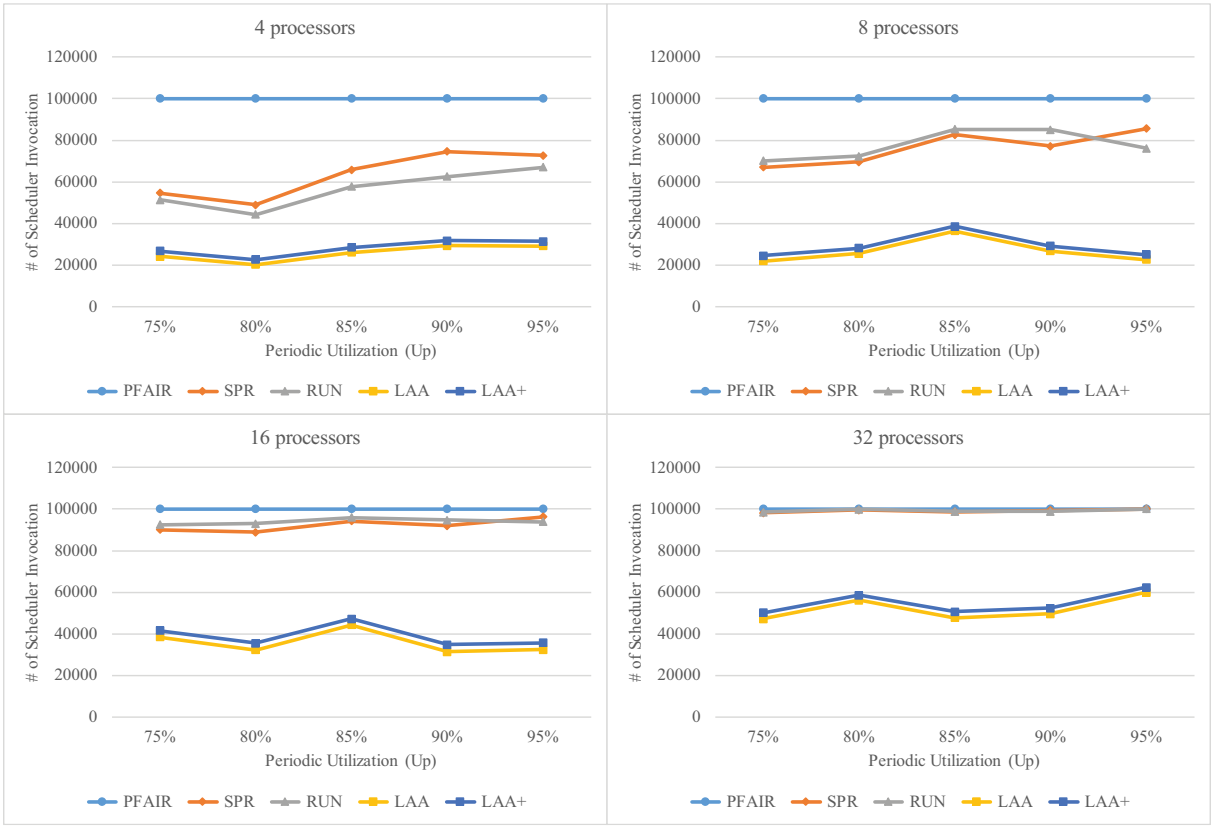

Figure 6.7: Scheduler invocation of LAA+

Chapter 7

Implementation of multiprocessor real-time operating system

7.1 Introduction of multiprocessor real-time operating system

In the last decades, together with the dramatic increase in quantity and diversity of real-time embedded applications, hardware platforms designed for such applications have been numerously introduced. Multiprocessors systems have become dominant in this trend. Systems with multiple processors allows diverse and complicated applications to be executed concurrently. As a essential element of real-time embedded systems, real-time operating system becomes attractive to researchers and engineers. A multiprocessors real-time operating system (M-RTOS) respected a scheduling algorithm importantly needs to handle different types of tasks and manage multiple shared resource, especially shared processors. Local Assignment Algorithm is introduced as an effective scheduling algorithm for real-time task scheduling on multiprocessors. Effectiveness of the algorithm was assessed by simulation; however, the applicability on practical environment is also desired. Implementation of M-RTOS which applies Local Assignment Algorithm as scheduling algorithm is therefore the goal of this chapter.

Savana Real-time Operating System [50] is a compact and reliable operating system kernel introduced for uniprocessor system. The operating system kernel is basically developed in C language and provides flexible mechanisms as system calls for task and system managements. Significant functions are listed in Table 7.1. The Savana operating system kernel is selected to extend to the multiprocessor context with focus on developing scheduling mechanism using Local Assignment Algorithm and integrating synchronization mechanism among processors.

The embedded system architecture is targeted to a symmetric multiprocessor system (SMP). Accordingly, a single version of the implemented M-RTOS is commonly accessed by all processors available in the system. Application tasks involved in the system are also common so that they can executed on any processor.

The remaining of this chapter is structured in five sections as follows. Section 7.2 introduces the hardware platform on which the M-RTOS is implemented. Section 7.3 determines requirements and functions that the implemented M-RTOS needs to achieve. Besides, difficulties of implementation are explained in this section. Section 7.4 describes

Table 7.1: Basic system calls provided in Savana RTOS

Group	**Functions provided**
Task management	Register task
	Activate task
	Cancel task activation
	Terminate invoking task
	Change task's priority
Task synchronization	Change a task to sleep state
	Wake a task up from sleep state
	Cancel a task wakeup
	Delay a task
Time management	Set system time
	Reference to system time
	Register a cyclic handler
	Start a cyclic handler
System management	Schedule task
	Dispatch task
	Reference to CPU state
	Change CPU state

design models of important parts of the implemented M-RTOS. In Section 7.5 is the actual system implementation including the hardware platform design and M-RTOS deployment. In Section 7.6 shows a test conducted to evaluate the M-RTOS. Finally, Section 7.7 is the conclusion of the chapter.

7.2 Hardware platform

7.2.1 Xilinx Zedboard Evaluation Kit

The Zedboard Evaluation Kit [9] is used as the targeted hardware platform for the implementation. Zedboard is a development board based on the Xilinx Zynq-7000 All Programmable SoC [59]. Zynq-7000 system is integrated with an ARM Cortex-A9 dual-core processors which can be set for running in SMP mode. Figure 7.1 shows an overview of the Zedboard board. Together with Xilinx Zynq-7000 All Programmable SoC, the board also provides Jtag connection, two USB connections, DDR3 memory, eight LEDs and eight switches which are useful for implementing and testing.

In addition, different from the traditional systems where hardware platform includ-

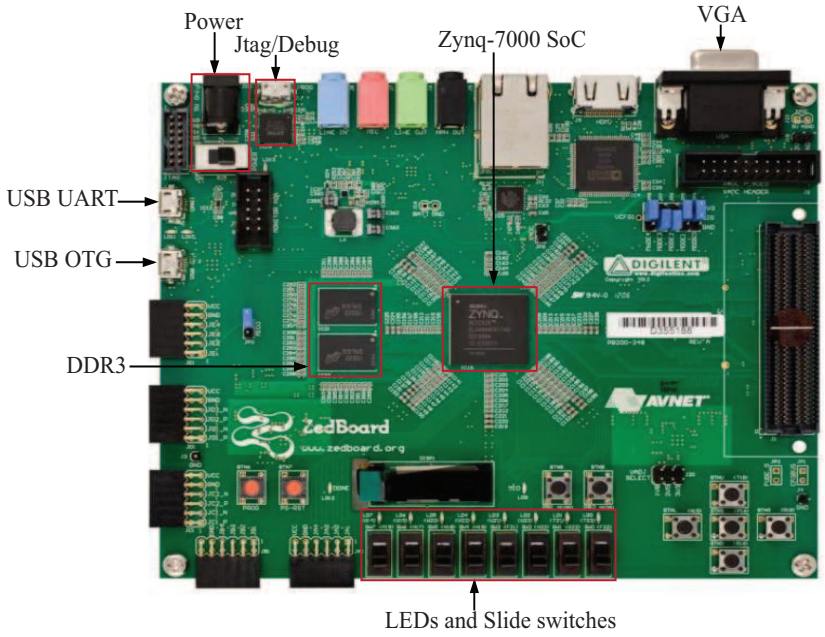

Figure 7.1: Overview of Zedboard Evaluation Kit

ing processors and built-in peripherals is predefined by manufacturers, Zynq-7000 All Programmable SoC is designed on field programmable gate array (FPGA) which allows flexible designs. In order work, Using Zynq-7000 All Programmable SoC, engineers can define their own system flexibly according to the design purposes.

7.2.2 ARM Cortex-A9 processor

ARM Cortex-A9 processor based on ARMv7 architecture is a low power and high performance processor which is widely encountered in embedded systems. Actually, the Cortex-A9 processor integrated in Zynq-7000 All Programmable SoC consists of two cores although the original Cortex-A9 can be configured with up to 4 cores.

Figure 7.2 shows an overview of the Cortex-A9 MPCore processor. The processor provides the following features:

- Two of ARMv7 32bit CPU (processor);
- Each processor has two separated 32KB L1-caches for instruction and data;
- Each processor also supports floating-point unit and NEON data engine;
- Each processor has its own private peripherals including private timer, watchdog timer, and interrupt interface, which are excluded on the figure;
- Two processors share snoop control unit (SCU), cache-to-cache transfer mechanism, one global timer, generic interrupt control (GIC), interrupt distributor, and accelerator coherence port (ACP).
- A shared dual 64 bit AMBA-3 bus.

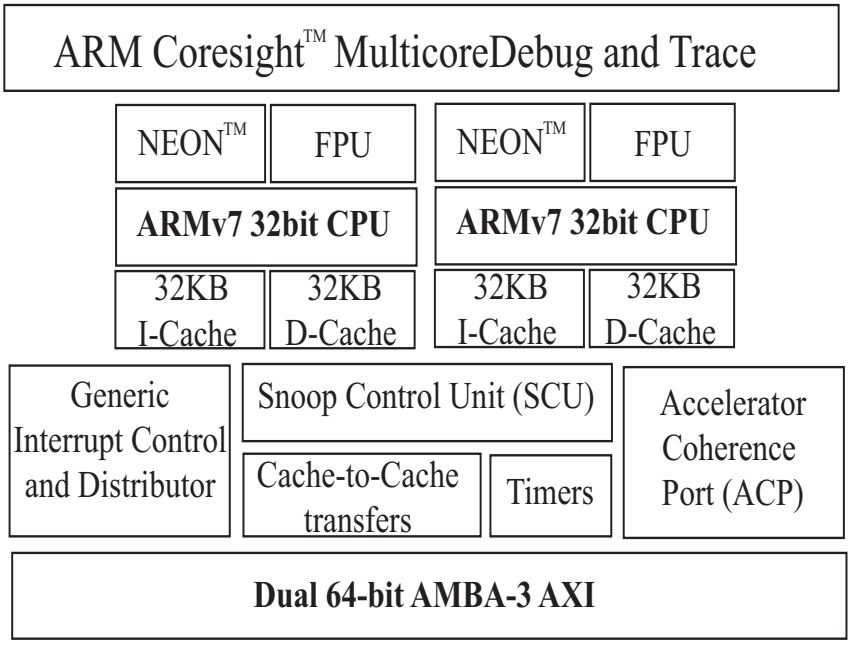

Figure 7.2: Arm Cortex-A9 MPCore processor

In addition, the system also supports a shared 512 KB level-two (L2) cache for instruction and data in parallel with a 256KB on-chip-memory (OCM) module. OCM allows low-latency memory accesses [57].

7.2.3 Operation mode and banked register in ARM Cortex-A9

ARM Cortex-A9 processor provides eight main modes of operation. Based on operation modes, the limitation of resource access is decided. There are two group of operation modes: privileged mode and non-privileged mode. The unprivileged modes have more restricted access to the system resources than the privileged modes. Table 7.2 lists main operation modes available in ARM Cortex-A9 processor with defined mode bits and privilege.

The operation modes are decide by the lowest five bits of the current processor status register (CPSR) [57]. Accordingly, user mode is the only non-privileged mode. This is the basic mode on which application programs are executed. IRQ and FIQ modes are corresponding interrupt and fast interrupt requests, respectively. Supervisor mode is often entered when system setting is required. The operating system is run in this mode. Abort and undefined modes are introduced to processor for abnormal events; that is, abort mode is used when processor requests to access an unreachable memory location and undefined mode is invoked when an undefined instruction is executed. System mode has permission to access full system resources. This mode can be entered from the other modes except the user one.

In ARM Cortex-A9 processor, banked registers are introduced as different registers shared the same address. There are two main types of banked registers: banked per mode and banked per processor. Banked registers are accessed using the address coordinated with operation mode or processor property. For example, figure 7.3 shows the general

Table 7.2: Operation modes in ARM Cortex-A9 processors

Mode	**Mode bits**	**Privilege**
User (USR)	Ox10	Non-privileged
FIQ	0x11	Privileged
IRQ	Ox12	Privileged
Supervisor (SVC)	Ox13	Privileged
Abort	Ox17	Privileged
Undefined	Ox1C	Privileged
System	Ox1F	Privileged

purpose registers in ARM Cortex-A9 processor respect to operation modes. As shown in the figure, R0-R12, R15 and CPSR are common among all operation modes, except FIQ; namely, these registers are accessed normally using the register address. R13 (stack pointer), R14 (link register), and SPSR (saved processor status register) are banked registers; based on the operation mode, the corresponding registers will be accessed. In FIQ mode, R8-R14 and SPSR are banked registers.

For the register banked per processor, the processor identifier is used to refer to the specific register. Comparator registers of global timer are example of registers banked per processor. Two processors have their own comparator registers which can be accessed by one address [59].

Being banked is also introduce to bits of some registers in ARM Cortex-A9. The status bit (the first bit) of the global timer interrupt status register is for example banked per processor. Banked registers and banked bits support rapid switching when processing processor exceptions in privileged modes. Moreover, this approach are also helpful to save the address field when one address can be used to refer to different registers.

7.2.4 Software tools

In order to design the hardware platform, Vivado Design Suite (version 2014.2) provided by Xilinx is used. Vivado allows users to quickly initialize the hardware systems, customize involved IP cores, and configure all peripherals. Using Vivado, users can also conveniently generate a wrap of hardware design for software development with Xilinx SDK.

Xilinx SDK is provided with integrated tools for use to develop softwares with Zedboard Evaluation Kit. It is compatible with several high level programming languages including C and C++ for software development. Besides, SDK also supports a debugger which is helpful for debugging and testing.

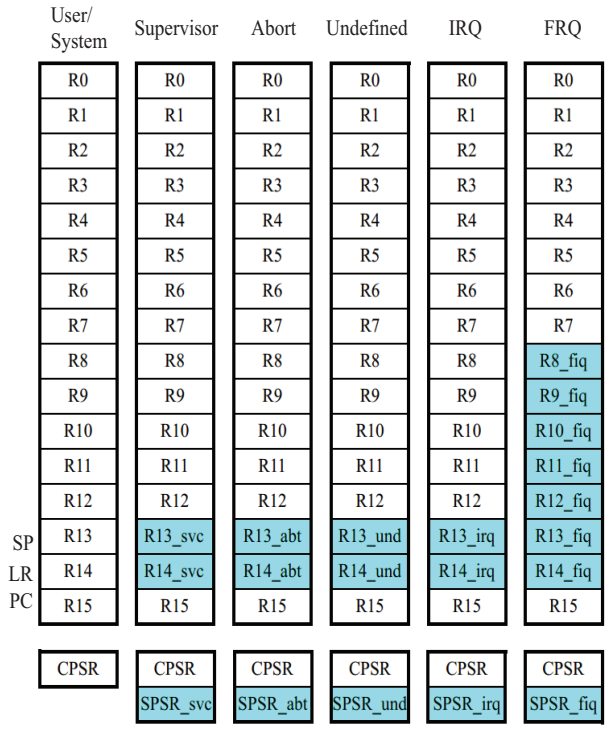

Figure 7.3: General-purpose registers corresponding to operation modes [60]

7.3 Requirements and difficulties of the implementation

7.3.1 Requirements of the implementation

The principle goal of this work is to implement a version of M-RTOS so as to evaluate the applicability of the proposed scheduling algorithm. Therefore, the compulsory requirement is that Local Assignment Algorithm is used for task scheduling.

The implemented M-RTOS is need to be embedded into a system involving multiple processors (at least two ones) for evaluation. The system has to allow different tasks to be executed concurrently among processors with guarantee of timing constraints including deadline meets.

Since the M-RTOS is common among processors in SMP architecture, synchronization mechanisms are required to synchronize processors' works when requesting system services, especially task scheduling and dispatching.

7.3.2 Difficulties of the implementation

Cache coherence

It is normal that each processor in the system has their own cache level 1 (L1-cache). Processors are manipulating on data available on L1-caches only. When multiple processors request for shared data, multiple copy of the data can exist on processors' L1-caches as a result. Therefore, it is important to maintain caches in coherent state for data consis-

tency. Cache coherence guarantees that shared data is updated throughout all processors' L1-caches when a modification is done on the data.

There are several approaches to manage cache coherence: setting access permission for share data and using snooping system. On the first approach, shared data is stored in a common directory on the primary memory which is coherent among L1-caches. When a processor requests to load shared data from the primary memory to its own L1-cache, it must acquire a permission. When the shared data is changed, the directory updates the content and invalidates L1-caches with the updated content.

On the second approach, all L1-caches snoop the bus to check the validation of the copy of the shared data which is requested on the bus. This approach is supported in many multiprocessor architectures such as ARM Cortex-A9 processor [57].

Consistency of task execution

In SMP system, application tasks are common among processors. In other words, application tasks can be considered as a kind of shared data. Since application tasks are allowed to migrate from a processor to one another, the execution of tasks needs to be guaranteed for consistency.

In multiprocessor systems, task migrations happen at scheduling points only. Store and restore procedures are therefore required to save the execution status of application tasks at the scheduling points. These procedures actually cause runtime overhead to the system performance.

7.4 System design of M-RTOS

7.4.1 Booting sequences

In multiprocessor system, the booting sequence is designed for dual booting on the two processors. In our implementation, the purpose of booting is to enable private memory management units (MMU), initialize the common memory for the synchronization between the two processors, and configure the UART connection. In the dual boot sequence, processor 0 is assigned as primary one which will be automatically started as power is turned on. The other processor serves as a secondary one. Besides initializing its private components, the primary processor is in charge of performing hardware initialization and executing initialization code for common settings, like UART connection, which need to be performed once only. In addition, the primary processor importantly also performs needed procedure to wake the secondary one up from the (default) idle state. Whereas, the secondary processor just executes initialization code for its private components after woken up. The secondary processor then sends a notification signal back to the primary one when all needed initialization are done. After receiving the signal that the secondary processor is woken up successfully, the primary processor can continue to prepare for the M-RTOS.

Figure 7.4 shows a flow of boot sequences on two processors.

Accordingly, the primary processor will process the following sequence:

1. Start at the designated address, 0x01000000 as designed in this implementation;

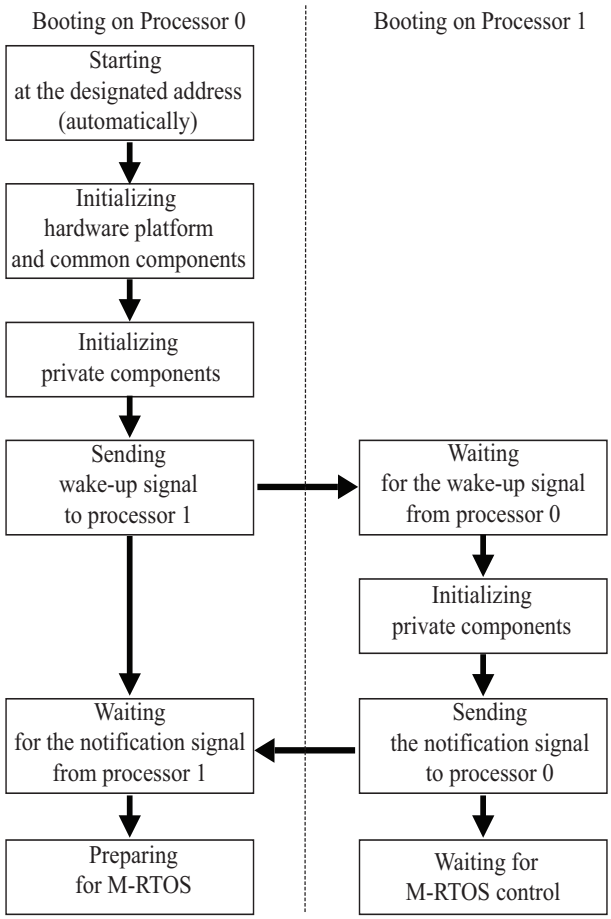

Figure 7.4: Dual boot sequence on two processors

2. Initialize the hardware platform including cache transfer mechanism, UART connection;
3. Initialize its private components including MMU, OCM, Cache L1;
4. Send wake-up signal to the secondary processor;
5. Wait for the notification signal from the secondary processor;
6. Prepare for downloading M-RTOS from computer to the main memory.

The secondary processor will process the following sequence:

1. At power on, the primary processor is in idle mode and waiting for wake-up signal at the designated address, 0xFFFFFFF0 as default;
2. Start by wake-up signal from the primary processor;
3. Initialize its private components including MMU, OCM, Cache L1;
4. Send notification signal to the primary processor that it is woken up successfully;
5. Wait for the control signal from the operating system.

7.4.2 Memory mapping

For the implementation of SMP architecture, the memory is divided into seven segments with the address map showed in Figure 7.5. The lowest address area from 0 to 0x00FFFFFF is preserved for first stage boot loader (FSBL) program and BootROM. As showed in the figure, These two segments are mapped to the OCM low address. Consecutively, the memory area from 0x01000000 to 0x02FFFFFF is used to store the booting program of processor 0 (P0) and the memory area from 0x03FFFFFF to 0x0FFFFFFF is for the booting program of processor 1 (P1).

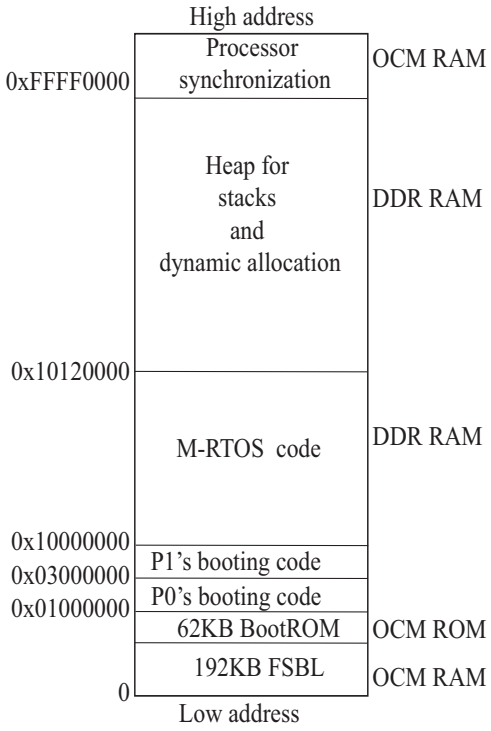

Figure 7.5: Memory address mapping

The executable code of M-RTOS is stored in the segment from address 0x10000000 to address 0x1011FFFF. The next segment from address 0x10120000 to address 0xFFFEFFFF is preserved for heap memory. This area is used for implementation of stacks and dynamic allocations. The two segments for M-RTOS and heap as showed are mapped to DDR RAM, which is the primary memory in the system. The remaining address field from 0xFFFF0000 is used for synchronization data between two processors. This area is mapped to the OCM high address.

Figure 7.6 shows a address plan for the top of stack pointers in the heap segment. Stack pointers are designed respect to operation modes for each processor. Stacks are implemented for supervisor, FIQ, IRQ and user modes. Abort and undefined modes are not implemented in this research while system mode shares stacks with user mode. A stack entry is preserved to save the status of system registers when dealing with processor exceptions and interrupts. Registers considered to save include R0-R3, SP and LR, which leads the entry size of 24 words.

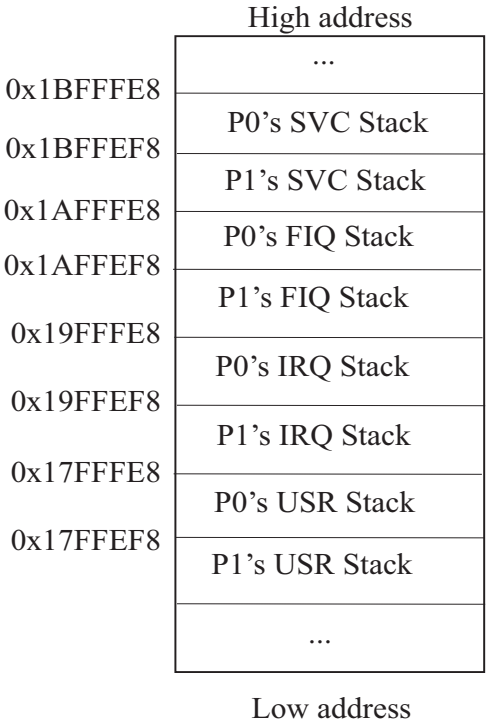

Figure 7.6: Stack pointers planned for operation modes

7.4.3 Organization of ready queue

The system is designed with 16 priorities from 0 to 15 for tasks. The priority of tasks decreases in the increase of priority index. The task indexed as priority 0 has the highest priority while the task indexed as priority 15 has the lowest priority. Priorities 0 and 1 are preserved for kernel tasks including initial task for processor 0, initial task for processor 1, timer task All application tasks are assigned the same priority of 2. Priorities 3 to 13 are preserved for future tasks. The remaining priorities of 14 and 15 are used for idle tasks of processor 0 and processor 1, respectively.

The system ready queue is decided based on task priorities. Therefore, 16 ready queues are established for tasks, which is depicted in Figure 7.7. Accordingly, ready queue 0 is queue of the initial task of processor 0's kernel and timer task, and ready queue 0 is queue of the initial task of processor 1's kernel. Application tasks are queued on ready queue 2 when they are released. Ready queues 3 to 13 are preserved for future tasks. The remaining two ready queue are utilized by idle tasks of processor 0 and processor 1. After initialization finished, processors will executed idle tasks when there exists no timer task and application tasks to be executed.

7.4.4 Dual initialization for M-RTOS

After the boot stage, M-RTOS is downloaded from computer to the primary memory at address 0x10000000 as designated above and then started. This work is done by processor 0 as the primary processor. M-RTOS starts execution with a dual initialization which initializes system environment required for the operating system. Figure 7.8 shows the steps of the dual initialization on two processors.

ready_queue[0]	Kernel tasks of P0: kernel 0 init, timer
ready_queue[1]	Kernel task of P1: kernel 1 init
ready_queue[2]	Application tasks
ready_queue[3] to ready_queue[13]	Preserved
ready_queue[14]	Kernel task of P0 in idle state
ready_queue[15]	Kernel task of P1 in idle state

Figure 7.7: Organization of ready queues

In the dual initialization, the primary processor P0 will do the following works:

1. Start at the designated address 0x10000000;
2. Initialize stack pointers for the processor;
3. Set vector base address (VBAR) and enable performance monitor unit (PMU), snoop control unit (SCU);
4. Initialize global elements including global variables, system queues;
5. Initialize private elements including kernel initial task, timer task (intentionally, timer task is executed on processor 0 only), and private peripherals and interrupt (PPI). This initialization step includes configurations to the generic interrupt control (GIC) and interrupt distributor;
6. Send a signal to processor 1 to start its own initialization. Before sending the signal, processor 0 needs preparing the starting address for processor 1 in the common synchronization area;
7. Wait for the notification signal that processor 1 finished its initialization successfully;
8. Start scheduler for task execution.

In its initialization, the secondary processor P1 will do the following works:

1. Wait for the signal to start the initialization needed from the primary processor. Processor 1 goes into this waiting state as finishing its own boot sequence;
2. Start the initialization program at the designated address (prepared by the primary processor);
3. Initialize stack pointers for the processor;
4. Set its own VBAR and PMU;

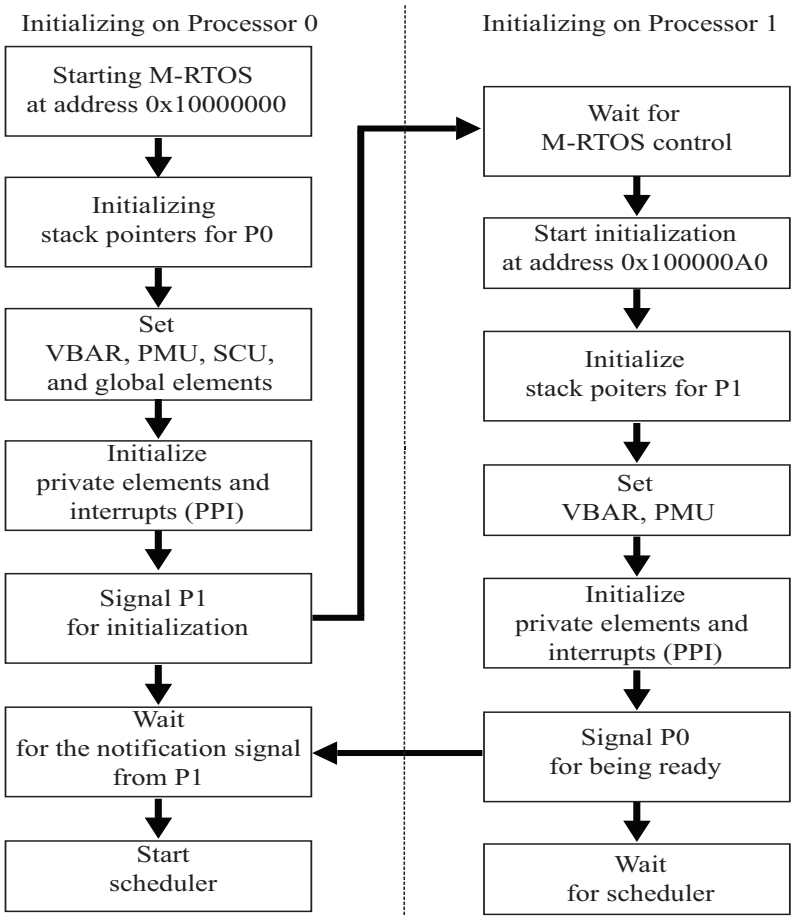

Figure 7.8: Dual initialization for M-RTOS on two processors

5. Initialize private elements including its own kernel initial task, and private peripherals and interrupt (PPI).
6. Send a signal to the primary processor to notify that the initialization is done;
7. Wait for the scheduler to fetch task.

7.4.5 Synchronization required for two processors

As described in the dual boot sequence and M-RTOS dual initialization, several communications and data are needed between two processors:

- Mutual communication to wake a processor up from the idle state at the boot stage;
- Mutual communication to start the initialization on a processor;
- Data as starting address for the secondary processor.

In addition, later the mutual communications are also required between the two processors for interrupt handling and task scheduling.

In order to support such synchronizations between the two processors, a common structure of data is used. Figure 7.9 shows the structure of the common block data,

which is allocated on the segment of processor synchronization (OCM high) of memory (see Section 7.4.2). The data structure includes seven elements as follows:

- *isP1Start*: used at the boot stage. The primary processor will clear this data to 0 before sending the wake-up signal to the secondary one. Then, it is waiting for this data to be set. By setting this data to 1, the secondary processor notifies the primary one that it is started successfully.
- *isP1Ready*: used at the initialization of M-RTOS in the similar manner to *isP1Start*. That is, it is used by the secondary processor to confirm that it finished initialization and is ready for task execution.
- *start_add*: the start address of the secondary processor which is prepared by the primary one.
- it sche_done: used for the scheduler.
- The other elements are preserved for future usage.

```
/* Common Block definitions */
typedef  struct {
 volatile  BOOL   isP1Start;
 volatile  BOOL   isP1Ready;
 volatile  FP     start_add ;
 volatile  INT    sche_done;
 volatile  INT    data1;
 volatile  INT    data2;
 volatile  INT    data3;
} cpuComBlock;
```

Figure 7.9: Structure of the common block data for processor synchronization

7.5 Implementation

7.5.1 Hardware platform design

The hardware platform is designed using Xilinx Vidado Suite which provides flexible configurations for Zynq-7000 All Programmable SoC. Figure 7.10 exhibits the block diagram of the hardware system. The block diagram is generated using Xilinx Vivado. There are four main blocks:

- Zynq7 Processing system integrated with ARM Cortex-A9 dual core processor.
- Processor system reset to generate a global reset signal for the whole system.
- AXI Interconnection to synchronize the connections between the main processing system (Zynq7) and input/output ports.

- AXI GPIO to provide direct connection to input and output port.

The system is configured in symmetric multiprocessor mode (SMP) with two processors. SCU is enable for cache coherence and data transfers. GIC is activated for interrupt control and distributor. Global timer is used for implementation of system time (tick). AXI GPIO is set as dual 8-bit port in which one input 8-bit port is connected to slide switches and the other output 8-bit port is connected LEDs. UART protocol is used for UART connection at bound rate of 11500. Clocks of 666 MHz and 333 MHz are used for processor and peripherals (including global timer), respectively.

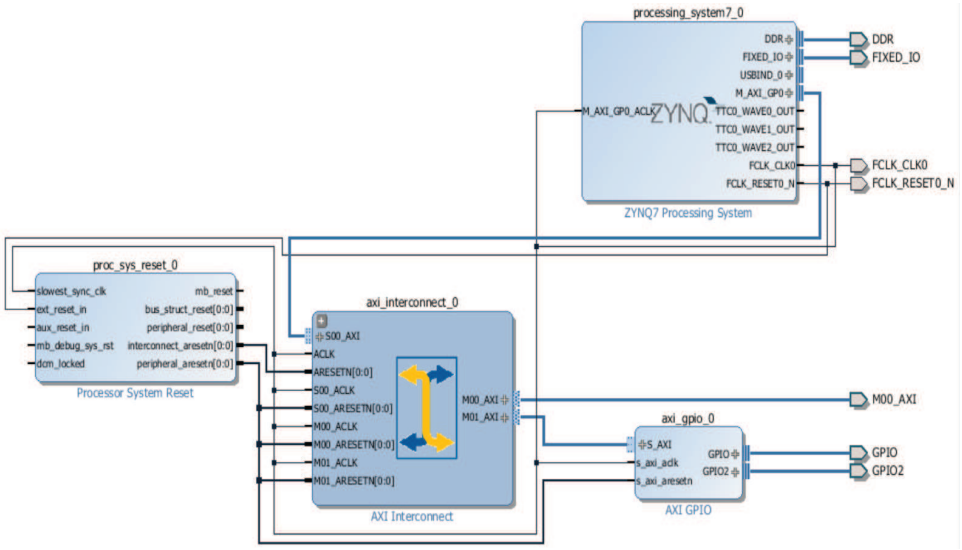

Figure 7.10: Block diagram of the hardware platform generated by Xilinx Vivado

Using Vivado, the system is implemented and the bit stream of hardware design is generated for FPGA programming. Xilinx SDK is used to create a first stage boot loader for two processors of the system. Boot programs and bit stream of hardware design are then programmed to FPGA on Zedboard Evaluation Kit.

7.5.2 Basic components of M-RTOS

Vector table and exception handling

In the ARM architecture, vector table provides entries of exception handling that are executed by processor to handle events. Exceptions can be caused by internal or external sources. When an exception occurs, the execution on the processor is forced from a predefined memory address of the corresponding exception handling. There are eight types of exceptions defined in the ARM architecture including reset, undefined instruction, software interrupt, abort instruction, abort data, IRQ, FIQ, and one preserved type [61]. The address of the exception handling is decided in reference to the base address stored in VBAR. Exception handlings and base addresses for two processors can be designed differently.

In this implementation, the base address for processor 0 is decided at 0x10000000. Figure 7.11 shows defined addresses of exceptions with the corresponding exception handling

for processor 0. Three important exceptions involved in this implementation are reset, software interrupt and IRQ. Reset exception is forced at system reset, software interrupt is used for task dispatching, and IRQ is used to handle the global timer interrupt. The corresponding handling for the three exceptions are *_kernel_set_sp*, *_kernel_swi_entry*, and *_kernel_irq_entry*, respectively. The address of these handlings are found in the figure referred to the processor 0's base address.

```
_kernel_vector :
    b    _kernel_set_sp              @ 0x10000000 SP setting
    b    _kernel_undef_inst_entry    @ 0x10000004 Undefined instruction entry
    b    _kernel_swi_entry           @ 0x10000008 SWI entry
    b    _kernel_inst_abort_entry    @ 0x1000000C Abort instruction entry
    b    _kernel_data_abort_entry    @ 0x10000010 Abort data entry
    nop                              @ 0x10000014
    b    _kernel_irq_entry           @ 0x10000018 IRQ entry
    b    _kernel_fiq_entry           @ 0x1000001C FIQ entry
```

Figure 7.11: Vector table for processor 0

Similarly, in Figure 7.12, the exception handlings and predefined addresses of exceptions for processor 1 are shown. Address 0x100000A0 is decided as the base address of processor 1, which is stored in processor 1's VBAR. Corresponding to three involved exceptions, exception handlings for processor 1 are *_kernel_set_sp_p*1, *_kernel_swi_entry_p*1, and *_kernel_irq_entry_p*1.

```
_kernel_vector_p1 :
    b    _kernel_set_sp_p1           @ 0x100000A0 SP setting
    b    _kernel_undef_inst_entry    @ 0x100000A4 Undefined instruction entry
    b    _kernel_swi_entry_p1        @ 0x100000A8 SWI entry
    b    _kernel_inst_abort_entry    @ 0x100000AC Abort instruction entry
    b    _kernel_data_abort_entry    @ 0x100000B0 Abort data entry
    nop                              @ 0x100000B4
    b    _kernel_irq_entry_p1        @ 0x100000B8 IRQ entry
    b    _kernel_fiq_entry           @ 0x100000BC FIQ entry
```

Figure 7.12: Vector table for processor 1

Kernel stack pointers

Figure 7.13 describes the program to set the stack pointers for processor 0, which is following the stack pointer plan in Figure 7.6. As showed, stack pointers are set in the exception handling of reset in the initialization of processor 0 (see Figure 7.8). In the program, *_KERNEL_STK_PTR* is predefined as 0x1C000000. Notably, at the setting for supervisor mode stack, a routine *_kernel_init_proc* is executed for the processor initialization. The routine is to configure VBAR, MMU, SCU and global elements for processor 0. At the setting for user mode stack, a routine $_{k}ernel_{m}ain$ is invoked to enter the kernel main task on processor 0.

```
_kernel_set_sp :
    mrs   r0, cpsr                                  @ 0x10000020 Starting SP setting
    bic   r0, r0, #0x1F                             @ 0x10000024
    orr   r0, r0, #0x11                             @ 0x10000028
    msr   cpsr, r0                                  @ 0x1000002C move to fiq mode
    mov   r1, #_KERNEL_STK_PTR−0x1000000 @ 0x10000030 prepare sp(fiq)
    mov   sp, r1                                    @ 0x10000034 write sp(fiq)
    mrs   r0, cpsr                                  @ 0x10000038
    bic   r0, r0, #0x1F                             @ 0x1000003C
    orr   r0, r0, #0x12                             @ 0x10000040
    msr   cpsr, r0                                  @ 0x10000044 move to irq mode
    mov   r1, #_KERNEL_STK_PTR−0x2000000 @ 0x10000048 prepare sp(irq)
    mov   sp, r1                                    @ 0x1000004C write sp(irq)
    mrs   r0, cpsr                                  @ 0x10000050
    bic   r0, r0, #0x1F                             @ 0x10000054
    orr   r0, r0, #0x13                             @ 0x10000058
    msr   cpsr, r0                                  @ 0x1000005C move to svc mode
    mov   r1, #_KERNEL_STK_PTR                      @ 0x10000060 prepare sp(svc)
    mov   sp, r1                                    @ 0x10000064 write sp(svc)
    bl  _kernel_init_proc                           @ 0x10000068 P0 Initialization
    nop                                             @ 0x1000006C
    mrs   r0, cpsr                                  @ 0x10000070
    bic   r0, r0, #0x1F                             @ 0x10000074
    orr   r0, r0, #0x10                             @ 0x10000078
    msr   cpsr, r0                                  @ 0x1000007C move to usr mode
    mov   r1, #_KERNEL_STK_PTR−0x4000000  @ 0x10000080 prepare sp(usr)
    mov   sp, r1                                    @ 0x10000084 write sp(usr)
    bl _kernel_main                                 @ 0x10000088 P0 kernel main
    nop                                             @ 0x1000008C
```

Figure 7.13: Setting stack pointers for processor 0

Similarly, Figure 7.14 shows the setting program for stack pointers of processor 1. As showed, each stack pointer of processor 1 is distinguished by 240 words lower than the corresponding stack pointers of processor 0. It means that, ten stack entries size of which is 24 words are kept for processor 0. At settings of stack pointers for supervisor and user modes, needed routines are executed to initialize the private elements for processor 1.

Interrupt routines

As mentioned above, interrupt routines are designed for processors to handle the global timer interrupt which is used to implement the system time. To handle the global timer interrupt, processor 0 will do the following things:

- Read interrupt acknowledge register (ICCIAR) to get information about the received interrupt;
- Clear the pending interrupt in the interrupt interface by writing the interrupt ID to the end of interrupt register (ICCEOIR);

```
_kernel_set_sp_p1 :
    mrs   r0, cpsr                                  @ 0x100000C0 Starting SP setting
    bic   r0, r0, #0x1F                             @ 0x100000C4
    orr   r0, r0, #0x11                             @ 0x100000C8
    msr   cpsr, r0                                  @ 0x100000CC move to fiq mode
    mov   r1, #_KERNEL_STK_PTR-0x1000000 @ 0x100000D0 prepare sp(fiq)
    sub r1,r1,#240                                  @ 0x100000D4
    mov   sp, r1                                    @ 0x100000D8 write sp(fiq)
    mrs   r0, cpsr                                  @ 0x100000DC
    bic   r0, r0, #0x1F                             @ 0x100000E0
    orr   r0, r0, #0x12                             @ 0x100000E4
    msr   cpsr, r0                                  @ 0x100000E8 move to irq mode
    mov   r1, #_KERNEL_STK_PTR-0x2000000 @ 0x100000EC prepare sp(irq)
    sub r1,r1,#240                                  @ 0x100000F0
    mov   sp, r1                                    @ 0x100000F4 write sp(irq)
    mrs   r0, cpsr                                  @ 0x100000F8
    bic   r0, r0, #0x1F                             @ 0x100000FC
    orr   r0, r0, #0x13                             @ 0x10000100
    msr   cpsr, r0                                  @ 0x10000104 move to svc mode
    mov   r1, #_KERNEL_STK_PTR                      @ 0x10000108 prepare sp(svc)
    sub r1,r1,#240                                  @ 0x1000010C
    mov   sp, r1                                    @ 0x10000110 write sp(svc)
    bl  _kernel_init_proc_p1                        @ 0x10000114 P1 Initialization
    nop                                             @ 0x10000118
    mrs   r0, cpsr                                  @ 0x1000011C
    bic   r0, r0, #0x1F                             @ 0x10000120
    orr   r0, r0, #0x10                             @ 0x10000124
    msr   cpsr, r0                                  @ 0x10000128 move to usr mode
    mov   r1, #_KERNEL_STK_PTR-0x4000000 @ 0x1000012C prepare sp(usr)
    sub r1,r1,#240                                  @ 0x10000130
    mov   sp, r1                                    @ 0x10000134 write sp(usr)
    bl _kernel_main_p1                              @ 0x10000138 P1 kernel main
    nop                                             @ 0x1000013C
```

Figure 7.14: Setting stack pointers for processor 1

- Wake up timer task which is designed to manage system time and task release;
- Clear the status bit in the global timer status register.

Figure 7.15 describes the interrupt routine of global timer interrupt for processor 0 in detail.

```
void _kernel_intr ( void )
{
    unsigned long *ICCIAR = (unsigned long *)0xF8F0010C;
    unsigned long *ICCEOIR = (unsigned long *)0xF8F00110;
    unsigned long icciar;

    icciar = *ICCIAR; // [12:10] CPUID, [9:0] Int ID
    *ICCEOIR = icciar; // Clear pending interrupt
    _kernel_int = 1;

    INT int_ID;
    int_ID = icciar & 0x3FF;

    switch ( int_ID ) {
        case 27: // ID27: global timer
            iwup_tsk ( _KERNEL_TIM );
            break;
        default:
            a9_usr_printf ( "int_ID = ", int_ID );
            outstr ( " is not implemented\r\n" );
            break;
    }

    _kernel_int = 0;
    *GT_Status_Reg = 1;

    return;
}
```

Figure 7.15: Timer interrupt routine for processor 0

Compared to processor 0, processor 1 has to do a lighter work to handle the global timer interrupt. Processor 1 will do the following things:

- Read register ICCIAR to get information about the received interrupt;
- Clear the pending interrupt in the interrupt interface by writing the interrupt ID to register ICCEOIR;
- Wait for the confirmation of scheduling from processor 0;
- Clear synchronization data;
- Clear the status bit in the global timer status register;

- Check for task dispatching.

Figure 7.16 shows details of the interrupt routine of global timer interrupt for processor 1.

```
void _kernel_intr_p1 ( void )
{
    unsigned long *ICCIAR = (unsigned long *)0xF8F0010C;
    unsigned long *ICCEOIR = (unsigned long *)0xF8F00110;
    unsigned long icciar;

    icciar = *ICCIAR; // [12:10] CPUID, [9:0] Int ID
    *ICCEOIR = icciar; // Clear pending interrupt

    while(cpuComBlockPtr -> sche_done == 0);

    cpuComBlockPtr -> sche_done = 0;
    *GT_Status_Reg = 1;

    if( _kernel_schtsk_p1 != _kernel_schtsk_p1 )
        _kernel_dispatch ();

    return;
}
```

Figure 7.16: Timer interrupt routine for processor 1

System time

The system time (tick) is decided to be interval of 1 ms. The global timer is configured to release an interrupt signal for every 1 ms. Figure 7.17 displays the configuration of global timer registers for this purpose. Reminding that global timer in ARM Cortex-A9 processor consists of five registers:

- A 32-bit global timer control register (GT_Cntrl_Reg);
- A 64-bit comparator which is divided into two 32-bit registers ($GT\_Comp\_Reg\_Low$ and $GT\_Comp\_Reg\_High$);
- A 64-bit counter which is divided into two 32-bit registers ($GT\_Count\_Reg\_Low$ and $GT\_Count\_Reg\_High$);
- A 32-bit Auto-increment register ($GT\_AutoInc\_Reg$);
- A 32-bit status register (GT_Status_Reg).

A global timer interrupt is released whenever the timer counter reaches the value in the comparator register. The global timer can be configured for single shot interrupt or multiple interrupts with auto-increment mode. In the auto-increment mode, as the

```
outstr("==-- Resetting Global Timer"); outstr("\r\n");
*GT_Cntrl_Reg = 0;
*GT_Comp_Reg_Low = GT_AutoInc_Val;
*GT_Comp_Reg_High = 0;
*GT_AutoInc_Reg = GT_AutoInc_Val;
*GT_Count_Reg_Low = 0;
*GT_Count_Reg_High = 0;
*GT_Status_Reg = 1;
```

Figure 7.17: Configuration of global timer for system time

interrupt is sent the comparator register is automatically increased by the value in the auto-increment register for the next interrupt event.

Respect to the operation of global timer, the auto-increment value for comparator is decided equal to 0x00028A00 for the interval of 1 ms. The calculation of interval can be obtained using Equation 7.1 where *prescale_value* is the value in the *prescaler* field (bits 15-8) of the global timer control register [62], *increment_value* is the value of register $GT\_AutoInc\_Reg$, and GT_CLK is the clock of 333 MHz for global timer.

$$interval = \frac{(prescale\_value + 1) * (increment\_value + 1)}{GT\_CLK} \tag{7.1}$$

Scheduling process

Overall, the scheduling rule is designed based on the system priority. Accordingly, when the scheduler is invoked, the highest prioritized task existing in the system is selected for execution. Except application tasks which all have priority of 2, tasks with the same priority are selected using the first-come-first-serve strategy. As the purpose of this implementation, Local Assignment Algorithm is applied as the main strategy for scheduling application tasks.

Figure 7.18 shows the main segment of codes implemented the selection strategy of the scheduler. A loop is encountered to traverse the system ready queues from the lowest indexed ready queue tasks on which have the highest priority to the highest indexed ready queue tasks on which have the lowest priority. When a non-empty ready queue is reached, tasks on the queue are assigned to processors for execution and the traversing is stopped. Since all ready queues are initialized with the first element pointing to the queue itself, a non-empty queue is detected by expression $queue- > next! = queue$.

As showed in the figure, when application tasks on ready queue 2 are assigned to processors, the scheduling plan of Local Assignment Algorithm is referred to. $\_laa\_pHead[0]$ and $\_laa\_pHead[1]$ are the head of two scheduling plans corresponding to the two processors, which provide the identifier of scheduled tasks. $\_kernel\_schtsk$ and $\_kernel\_schtsk\_p1$ indicate the selected tasks for processor 0 and processor 1 respectively. If the head of scheduling plan of processor is non empty ($NULL$), the scheduled task is assigned to processor through procedure $getTask$. Otherwise, the processor is assigned the predefined idle task through procedure $idleTask$.

When the other ready queues (rather than ready queue 2) are involved to select task, the first task on the queue is assigned to processor 0 while processor 1 is assigned the predefine idle task. This is because processor 0 as the primary processor will be in charge

```
...
/* Select tasks from ready queues */
for ( i = 0; i <= TMAX_TPRI - 1; i++ )
{
    queue = (_KERNEL_QUEUE *)&_kernel_ready_q[i];
    if ( queue->next != queue ) // exist task in the queue
    {
        if(i == 2) // selecting application tasks
        {
            // select task for processor 0
            if(_laa_pHead[0] != NULL)
                _kernel_schtsk = getTask(_laa_pHead[0] -> tid);
            else
                _kernel_schtsk = idleTask(0);

            // select task for processor 1
            if(_laa_pHead[1] != NULL)
                _kernel_schtsk_p1 = getTask(_laa_pHead[1] -> tid);
            else
                _kernel_schtsk_p1 = idleTask(1);
        }
        else // selecting another prioritized tasks
        {
            _kernel_schtsk = queue -> next -> self;
            _kernel_schtsk_p1 = idleTask(1);
        }
        break;
    }
}
...
// Signal Processor 1 for its dispatching
cpuComBlockPtr -> sche_done = 1;
...
```

Figure 7.18: Schedule tasks using scheduling plan of LAA algorithm

of executing system tasks besides application tasks while processor 1 as the secondary one will be in charge of executing application tasks only. As mentioned before, this intentional assignment scheme is for the simplicity.

When the scheduling process is completed, the scheduler need to notify processor 1 so that it can process for dispatching task. This work is done by setting the synchronization value $cpuComBlockPtr->sche\_done$ to 1.

Informatively, the idle task of processor 0 is simply designed as a infinite loop the content of which is empty. In other words, processor 0 executes nothing in its predefined idle task. Whereas, the idle task of processor 1 is designed so that it is waiting for the synchronization signal of scheduling from processor 0. Figure 7.19 shows the source code of the predefined idle task for processor 1.

```
void  idleP1()
{
    extern  cpuComBlock *cpuComBlockPtr __asm__ ( "cpuComBlockPtr" );
    while(cpuComBlockPtr -> sche_done == 0)
    {
        if(cpuComBlockPtr -> sche_done == 1)
        {
            cpuComBlockPtr -> sche_done = 0;
            if( _kernel_schtsk_c1  !=  _kernel_schtsk_c1 )
                _kernel_dispatch ();
        }
    }
    return;
}
```

Figure 7.19: Idle task for processor 1

7.6 M-RTOS evaluation

7.6.1 Test scenario

In order to evaluate the implemented M-RTOS, a simple test scenario is established. The purpose of testing is task scheduling of M-RTOS with Local Assignment Algorithm. The test scenario is supposed to involve four periodic tasks identified as TSK_A, TSK_B, TSK_C, and TSK_D. Tasks' periods are equal to 15; that is all tasks are set to regularly release their jobs for every 15 ticks. The execution time of tasks are managed to approximately equal to 5 ticks. In other words, periodic tasks have identical utilizations of 1/3. The total utilization of tasks is approximately 1.3 so that the system is feasible to be scheduled on two processors. All periodic tasks start releasing their jobs at time 15.

Figure 7.20 shows an example program of application task involved in the test. When task is scheduled to execute on a processor, the program is executed. The program requests $exinf$ as the input argument. The input argument is utilized as the loop bound in the program, which mainly decides the execution span of the task. For the execution time of 5 ticks, input argument $exinf$ is assigned value equal to 3,500,000.

```
void task_a ( VP_INT exinf )
{
    SYSTIM    stim, ftim;
    volatile  INT i, j, x;

    get_tim ( &stim );
    a9_usr_printf ( "TSK_A starts ", stim );
    for ( i=0, x=0; i<exinf; i++ )
        x += i;
    get_tim ( &ftim );

    a9_usr_printf ( ", finish ", ftim);
    a9_usr_printf ( ", spends ", ftim - stim); outstr ( "\r\n" );
    return;
}
```

Figure 7.20: Example of application task involved in the system

7.6.2 Testing results

Tera Term emulator [63] is installed for communication with Zedboard Evaluation Kit via USB UART port. Since in the test, the executions of tasks on two processor may concurrently request to use USB UART, a simple synchronization was implemented using a shared value of the common block data (Section 7.4.5).

```
*********************************************
*  Dual boot program for Zedboard Zynq7000     *
*                    By Doan Duy, May 2019    *
*********************************************
P0: Platform initialization is done.
P0: Common Block configuration is done.
P0: Starting Core 1...
P1 is started.
P1: Platform initialization is done.
P1: Common Block configuration is done.
P1: Going to STANDBY.
P0: Preparing for downloading RTOS...
>
```

Figure 7.21: Result of dual boot on two processor

Figure 7.21 displays the screen capture of the dual boot on two processors. The displayed results follows the dual boot sequence designed in 7.4. The boot step is finished with processor 1 put into standby mode, waiting for the M-RTOS control, and processor 0 ready for downloading M-RTOS. Downloading M-RTOS means the compiled M-RTOS on computer is downloaded to the primary memory of the board. After downloading, M-RTOS is started at the designated address (0x10000000).

Figure 7.22 shows the results of schedule of four periodic tasks. At time 15 (0x0f

```
PPI_STATUS:
    ppi_status  = 00000000
    ([1] and [4] is active LOW, [0], [2], [3] are active HIGH
    ([0] corresponds to the global timer)
P1 is initialized sucessfully .
Start scheduler ...
TSK_A starts 0000000f, finish 00000014, spends 00000005
TSK_B starts 0000000f, finish 00000014, spends 00000005
TSK_D starts 00000014, finish 00000019, spends 00000005
TSK_C starts 00000014, finish 00000019, spends 00000005
TSK_A starts 0000001e, finish 00000023, spends 00000005
TSK_B starts 0000001e, finish 00000023, spends 00000005
TSK_D starts 00000023, finish 00000028, spends 00000005
TSK_C starts 00000023, finish 00000028, spends 00000005
TSK_A starts 0000002d, finish 00000032, spends 00000005
TSK_B starts 0000002d, finish 00000032, spends 00000005
TSK_D starts 00000032, finish 00000037, spends 00000005
TSK_C starts 00000032, finish 00000037, spends 00000005
```

Figure 7.22: Results of schedule multiple tasks on multiple processor

in hexadecimal) when tasks are first released, TSK_A and TSK_B are scheduled to be executed the two processors. They are finished their execution at time 20 (0x14) for 5-tick executions. Then, TSK_C and TSK_D start their executions up to time 25 (0x19). The task executions are repeated for the successive periods.

7.7 Conclusion

In this chapter, a compact M-RTOS has been presented. Local Assignment Algorithm (LAA) is employed as the scheduling strategy for periodic tasks. This is to evaluate the applicability of LAA algorithm in a practical real-time system. The targeted system environment is Xilinx Zynq-7000 All Programmable SoC which is built in FPGA. A dual-core ARM Cortex-A9 processor is involved as the targeted processor. The M-RTOS is therefore designed with comparability to symmetric multiprocessor system.

Important parts of the M-RTOS are presented including dual system initialization, memory map, stack plan, ready queue structure, and task selection strategy (scheduling). In addition, we also show how the global timer is used to implement system time. Different interrupt routines are introduced to two processors to handle the global timer interrupt. Together with the M-RTOS structure, a dual boot sequence for two processors are also designed.

The implemented M-RTOS is booted and run successfully on the targeted system on Zedboard Evaluation Kit. A simple test scenario is used for testing the scheduling algorithm. Although the system consists of two processors only, testing results confirm that LAA algorithm has potential to be applied to practical system.

Chapter 8

Conclusion of dissertation

8.1 Summary of the dissertation

It is fact that multiprocessor architectures have become popular in real-time computing systems. This trend allows real-time applications to be developed increasingly in diversity and complexity. Hybrid task sets which combine of different types of tasks frequently occur in embedded systems and cause challenges to researchers of the field. Scheduling mixture systems of periodic and aperiodic tasks on multiprocessors is the motivation of the work in this dissertation.

This work first focuses on the problem of periodic schedule on multiprocessors. Based on investigation of the introduced methods and existing problems, an enhanced scheduling algorithm called Local Assignment Algorithm (LAA) was proposed. LAA exploits the notion of fairness and interval-based scheduling to pursue the optimality. LAA suggests a calculation method to achieve fully-assigned systems on time intervals so that the system capacity is entirely spent for task execution. This has potential to improve the system performance when reducing unnecessary idle times of processors. In addition, for assigning tasks to processors, the consecutive task assignment scheme of McNaughton's algorithm is expanded with improvements that tasks are distributed selectively using the schedule history and the remaining slack times of processors. It is found that LAA can effectively schedule periodic tasks with fewest scheduler invocation and relatively low time complexity while still keeping task preemption and task migration comparable to the existing algorithm.

In success of scheduling periodic tasks, LAA is then enhanced to adapt with the mixture context of periodic and aperiodic tasks. In the new context, the scheduling goal is to improve the responsiveness of aperiodic tasks while guaranteeing valid schedules for periodic tasks. To this end, concept of servers is employed. Servers are modeled like a periodic task with utilization rate and period. The remaining system capacity unused from periodic tasks are distributed to servers' utilization rates and servers' periods are arbitrarily large. The integration of servers are considered as a preservation for aperiodic tasks' execution. The enhanced Local Assignment Algorithm is called LAA+. With the integration of servers, LAA+ is applied effectively to schedule periodic and aperiodic tasks altogether. Furthermore, several techniques are introduced intentionally to improve aperiodic responsiveness. First, allocation of servers and periodic tasks on processors is decided in order of servers associated with aperiodic task, periodic tasks and empty servers. This has potential to execute aperiodic tasks (already assigned to servers) are

executed ahead periodic ones on a time interval and preserve empty servers for coming aperiodic tasks. Second, introduction of secondary scheduling event gives just-coming aperiodic tasks a benefit of being executed ahead periodic ones. Simulation results show that LAA+ can effective improve response times of aperiodic tasks while not significantly increasing the time complexity.

In addition to the propose of scheduling algorithm, a compact multiprocessor real-time operating system (M-RTOS) is developed for the applicability of the proposed algorithm on practical system. M-RTOS is implemented for a symmetric multiprocessor system of two processors. Xilinx Zynq-7000 All Programmable SoC with an integrated dual-core ARM Cortex-A9 processors is selected as the system environment. M-RTOS successfully using LAA to schedule periodic tasks in a simple scenario.

Overall, it is found that the proposed scheduling algorithm is more effective than the other existing optimal algorithms in terms of scheduler invocation and time complexity. It also has potential to be applied to practical real-time system.

8.2 Future work

It is fact that time intervals of LAA are decided using release times of periodic tasks only. Since release times are predictable, the information of intervals including interval length, involved tasks is obtainable in advance. This leads possibility of predicting scheduling plan of an interval in advance before the starting time of the interval actually occurs.

Another observation is risen from the implementation of M-RTOS. That is, during the scheduling time, only the primary processor is utilized while the secondary processor is free. The longer the scheduling time is spent, the more wasted the system capacity is. If the scheduling time is reduced, the system is exploited more effectively. Since the advanced scheduling plan is possible, an hardware accelerator which is in charge of calculating the scheduling plan for the next interval during the span of the current interval is promising.

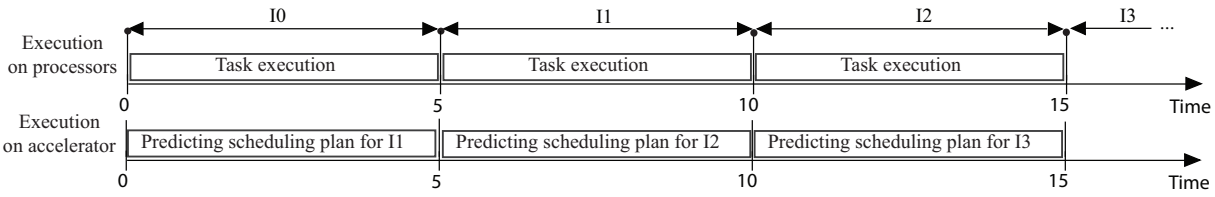

Figure 8.1: Making scheduling plans in advance with hardware accelerator

Figure 8.1 illustrates the idea of advanced scheduling plan with hardware accelerator. In the original system without hardware accelerator, scheduling events are decided at time 0, 5, 10 and 15 and the scheduling algorithm is executed at these points of time to make scheduling plans. Therefore, the issued runtime overhead tends to reduce the system performance. Now, if a hardware accelerator is integrated to be in charge of predicting scheduling plans, the scheduling plan of an interval can be made in advance. For example, the scheduling plan for interval I_1 can be predicted by the accelerator during the period of interval I_0 when processors are executing tasks. At the scheduling point at time 5, the

scheduler can use the predicted scheduling plan to make the schedule on interval I_1. This approach tends to reduce the runtime overhead of scheduling.

Hardware accelerator for LAA scheduling is therefore considered as future work of this dissertation.

References

[1] B. Ackland, A. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. J. Nicol, J. H. O'Neill, J. Othmer, E. Sackinger, K. J. Singh, J. Sweet, C. J. Terman, and J. Williams, "A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP," IEEE J. SolidState Circuits, Vol. 35, No. 3, Mar. 2000, pp. 412-424.

[2] "C-5 Network Processor Architecture Guide," C-Port Corp., North Andover, MA, May 31, 2001. Online available: http://www.freescale.com/files/netcomm/doc/ref_manual/C5NPD0-AG.pdf?fsrch=1

[3] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," IEEE Des. Test. Comput., vol. 18, no. 5, Sep./Oct. 2001, pp. 21-31.

[4] A. Artieri, V. D'Alto, R. Chesson, M. Hopkins, and M. C. Rossi, "Nomadik:Open Multimedia Platform for Next Generation Mobile Devices," Technical article TA305, 2003. Online available: http://www.st.com

[5] "OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide," Texas Instruments Inc., Dallas, TX, Mar. 2004. Online available: http://www.ti.com

[6] J. Goodacre and A. N. Sloss, "Parallelism and the ARM instruction set architecture," Computer, Vol. 38, No. 7, July, 2005, pp. 42-50.

[7] "Intel IXP2855 Network Processor," Intel Corp., Santa Clara, CA, 2005. Online available: http://www.intel.com

[8] W. Eatherton, "The push of network processing to the top of the pyramid," Proc. of Symp. Architectures Netw. Commun. Syst., Princeton, NJ, October, 2005.

[9] Zedboard Reference Manual. Online retrieved at http://zedboard.org/product/zedboard

[10] Intel®Stratix^{TM}10 TX Signal Integrity Development Kit https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kits-s10-tx-si.html

[11] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," Proc. of AFIPS'67 Conference, New Jersey, US, April 1967, pp. 483-485.

[12] 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, 1973, pp. 46-61.

[13] M.L. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," Information Processing 74, North-Holland Publishing Company, 1974.

[14] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the Scalability of Real-time Scheduling Algorithms on Multicore Platform: A Case Study," IEEE Real-time Systems Symposium, 2008, pp. 157-169.

[15] J. Goossens, S. Funk, and S. Baruah, "Priority-driven Scheduling of Periodic Task Systems on Multiprocessors," Real-Time Systems, Vol. 25, 2003, pp. 187-205.

[16] A. Srinivasan and S. K. Baruah, "Deadline-based Scheduling of Peroidic Task Systems on Multiprocessors," Information Processing Letters, Vol.84, No.2, pp. 93-98.

[17] T. P. Baker, "An Analysis of EDF Schedulability on a Multiprocessor. IEEE Trans. on Parallel and Distributed Systems, Vol.16, 2005, pp. 760-768.

[18] S. Cho, S. K. Lee, A. Han, and K. J. Lin, "Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems," IEICE Trans. on Communications, E85-B(12), 2002, pp. 2859-2867.

[19] M. Cirinei and T. P. Baker (2007) "EDZL Scheduling Analysis," Proc. of the Euromicro Conference on Real-Time Systems, 2007, pp. 9-18.

[20] S. Kato and N. Yamasaki, "Global EDF-Based Scheduling with Efficient Priority Promotion," Proc. of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Kaohsiung, 2008, pp. 197-206.

[21] Daeyoung Kim and Yann-Hang Lee, "Periodic and Aperiodic Task Scheduling in Strongly Partitioned Integrated Real-time Systems," The Computer Journal, Volume 45, Issue 4, Jan. 2002, pp. 395-409.

[22] A. Burns, R. Davis, P. Wang, and F. Zhang, "Partitioned EDF Scheduling for Multiprocessors using C=D task Splitting Scheme," Real Time Systems, Vol. 48, Iss. 1, 2012, pp. 3-33.

[23] X. Piao, S. Han, H. Kim, M. Park, Y. Cho, and S. Cho, "Predictability of Earliest Deadline Zero Laxity Algorithm for Multiprocessor Real-time Systems," Proc. Of the IEEE International Symposium on Object and Component-Oriented Real-time Distributed Computing, 2006, pp. 359-364.

[24] J. Anderson, V. Bud, and U. C. Devi, "An EDF-based Scheduling Algorithm for MUltiprocessor Soft Real-time Systems," Proc. of the Euromicro Conference on Real-time Systems, Balearic Islands, Spain, July 2005, pp. 199-208.

[25] B. Andersson and E. Tovar, "Multiprocessor Scheduling with Few Preemptions," Proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney, Australia, 2006, pp. 322-334.

[26] B. Andersson and K. Bletsas, "Sporadic Multiprocessor Scheduling with Few Preemptions," Proc. of the 8th Euromicro Conference of Real-time Systems, IEEE, Prague, Czech Republic, July 2008, pp. 243-252.

[27] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-Partitioned Scheduling of Sporadic Tasks on Multiprocessors," Proc. of the 21st Euromicro Conference of Real-time Systems, IEEE, Dublin, Ireland, July 2009, pp. 249-258.

[28] J. A. Santos, G. Lima, K. Bletsasa, and S. Katoc, "Multiprocessor real-time scheduling with a few migrating tasks," Proc. of the 34th Real-Time Systems Symposium, 2013, pp. 170-181.

[29] B. Brandenburg, and M. Gul, "Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations," Proc. of the 37th IEEE Real-Time Systems Symposium (RTSS 2016), Porto, Portugal, 2016, pp. 99-110.

[30] A. Burns, R. Davis, P. Wang, F. Zhang, "Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme," Real-Time Systems, Vol. 48, 2012, pp. 3-33.

[31] M. L. Dertouzos and A. K. Mok, "Multiprocessor On-line Scheduling of Hard-Real-time Tasks," IEEE Trans. on Software Engineering, Vol. 15, No. 12, 1989, pp. 1497-1506.

[32] S.K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate Progress: a Notion of Fairness in Resource Allocation," Algorithmica, Vol. 15, Issue 6, June 1996, pp. 600-625.

[33] S.K. Baruah, J. E. Gehrke, and C. G. Plaxton, "Fast Scheduling of Periodic Tasks on Multiple Resources," Proc. of the 9th International Parallel Processing Symposium, April 1995, pp. 280-288.

[34] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," Proc. of 12th Euromicro Conference on Real-Time Systems, Euromicro RTS 2000, Stockholm, 2000, pp. 35-43.

[35] Dakai Zhu, D. Mosse, and Rami Melhem, "Multiple-Resource Periodic Scheduling Problem: How Much Fairness is Necessary?," Proc. of the 24 IEEE Real-time Systems Symposium, Mexico, 2003, pp. 142-151.

[36] H. Cho, B. Ravindran, and E. Jensen, "An Optimal Real-Time Scheduling Algorithm of EDF on Multiprocessor Platform," Proc. of the IEEE Real-Time Systems Symposium, 2006, pp. 101-110.

[37] G. Levin, S. Funk, C. Sadowski, I. Pye and S. Brandt, "DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling," Proc. of the 22nd Euromicro Conference on Real-Time Systems, Brussels, 2010, pp. 3-13.

[38] P. Regnier, G. Lima, E. Massa, G. Levin and S. Brandt, "RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor," 2011 IEEE 32nd Real-Time Systems Symposium, Vienna, 2011, pp. 104-115.

[39] E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt, "OUTSTANDING PAPER: Optimal and adaptive multiprocessor real-time scheduling:The quasi-partitioning approach," Proc. of ECRTS 2014, Madrid, Spain, 2014, pp. 291-300.

[40] M. Spuri and G. C. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline First Scheduling," Proc. of Real-Time Systems Symposium, Puerto Rico, USA, December, 1994, pp.2-11.

[41] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," Real-Time Systems, Vol. 10, Iss. 2, March 1996, pp 179-210.

[42] L. Abeni and G. C. Buttazzo, "Integrating Multimedia Application in Hard Real-Time Systems," Proc. of Real-Time Systems Symposium, December 1998, pp. 4-13.

[43] Tanaka, K., "Virtual Release Advancing for Earlier Deadlines," ACM SIGBED Review, Vol.12, Iss. 3, pp 28-31.

[44] G. C. Buttazzo and M. Caccamo, "Minimizing Aperiodic Response Times in a Firm Real-Time Environment," IEEE Trans. on Software Engineering, February 1999, pp. 22-32.

[45] G. C. Buttazzo and F. Sensini, "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environment," IEEE Trans. on Computer, October 1999, pp. 39-48.

[46] K. Tanaka, "Adaptive Total Bandwidth Server: Using Predictive Execution Time," Proc. of International Embedded Systems Symposium (IESS), Springer, 2013, pp.250-261.

[47] S. Baruah, J. Gossens, and G. Lipari, "Implementing Constant-Bandwidth Servers upon Multiprocessor Platforms," Proc. of the 8th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS'02), IEEE Computer Society Press, California, US, Sep. 2002, pp. 154-163.

[48] S. Baruah, and G. Lipari, "A Multiprocessor Implementation of the Total Bandwidth Server," Proc. of the 8th International Parallel and Distributed Processing Symposium (IPDPS'04), New Mexico, US, 2004, pp. 541-550.

[49] A. Srinivasan, P. Holman and J. H. Anderson, "Integrating a periodic and recurrent tasks on fair-scheduled multiprocessors," Proc. of the 14th Euromicro Conference on Real-Time Systems (Euromicro RTS 2002), Vienna, Austria, 2002, pp. 17-26.

[50] ITRON4.0 Speciation. ITRON Committee, TRON ASSOCIATION(Japan). Version 4.00.00. Tanaka, K., "Real-Time Operating System Kernel for Multithreaded Processor," Proc. of IEEE International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA), pp. 91-99.

[51] C. L. Liu, "Scheduling Algorithms for Multiprocessors in a Hard-Real-Time Environment," JPS Space Programs Summary 37-60, Vol. II, Jet Propulsion Lab., California Institute of Technology, CA, US, November, 1969, pp. 28-37.

[52] P. Holman and J. H. Anderson, "Adapting Pfair scheduling for symmetric multiprocessors," Journal of Embedded Computing, Vol. 1, Vo. 4, 2005, pp. 543-564.

[53] R. McNaughton, "Scheduling with Deadlines and Loss Functions," Management Science, Vol. 6, No. 1, October 1959, pp. 1-12.

[54] Duy D. and Tanaka K., "Enhanced virtual release advancing algorithm for real-time task scheduling," Journal of Information and Telecommunication, Taylor&Francis, Vol.1, Jan. 2018.

[55] Duy D. and Tanaka K., "An Effective Approach for Improving Responsiveness of Total Bandwidth Server," Proc. of the 8th International Conference on Information and Communication Technology for Embedded Systems (IC-ICTES 2017), Chonburi, Thailand, May 2017, pp. 1-6.

[56] [3] Duy D. and Tanaka K., "A Hardware Implementation of the Enhanced Virtual Release Advancing Algorithm for Real-Time Task Scheduling," Proc. of the 18th IEEE International Conference on Industrial Technology (ICIT2017) ,Toronto, Canada, March 2017, pp. 953-958.

[57] Cortex-A9 Technical Reference Manual. ARM. Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f.

[58] Cortex-A9 Floating-Point Unit Technical Reference Manual. ARM. Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0408i

[59] Zynq-7000 SoC Technical Reference Manual, Zilinx, Ver. 1.12.2, July, 2018.

[60] Using the ARM Generic Interrupt Controller, Altera Corporation, April 2014.

[61] ARM Architecture Reference Manual, ARM Limited, 2005.

[62] Cortex-A9 MPCore Technical Reference, ARM, Revision: r3p0, 2011.

[63] T. Teranishi, Tera Term, Version 4.9, Mar 1998. Online source: svn.osdn.jp/svnroot/ttssh2/trunk/

Publications

[1] Duy. D and Tanaka K., "Enhanced virtual release advancing algorithm for real-time task scheduling", Journal of Information and Telecommunication, Taylor&Francis, Vol.1, January 2018, pp. 246-264.
DOI: 10.1080/24751839.2018.1423789

[2] Duy D. and Tanaka K., "Adaptive Local Assignment Algorithm for Scheduling Soft-Aperiodic Tasks on Multiprocessors," The 25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Hangzhou, China, August 2019. To appear.

[3] Duy D. and Tanaka K., "A Novel Task-to-Processor Assignment Approach for Optimal Multiprocessor Real-Time Scheduling," Proc. of 2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Vietnam, September 2018, pp. 101-108.
DOI: 10.1109/MCSoC2018.2018.00028

[4] Duy D. and Tanaka K., "An Effective Approach for Improving Responsiveness of Total Bandwidth Server," Proc. of the 8th International Conference on Information and Communication Technology for Embedded Systems (IC-ICTES 2017), Chonburi, Thailand, May 2017, pp. 1-6.
DOI: 10.1109/ICTEmSys.2017.7958777

[5] Duy D. and Tanaka K., "Hardware Implementation of Enhanced Virtual Release Advancing Algorithm for Real-Time Task Scheduling," Proc. of the 18th IEEE International Conference on Industrial Technology (ICIT2017), Toronto, Canada, March 2017, pp. 953-958.
DOI:10.1109/ICIT.2017.7915489

[6] Duy D., Tanaka, K., "Enhanced Virtual Release Advancing for EDF-based Scheduling on Precise Real-Time Systems," Proc. of the Eighth International Conference on Knowledge and Systems Engineering (KSE2016), Hanoi, Vietnam, Oct. 6, 2016, pp. 43-48.
DOI: 10.1109/KSE.2016.7758027

Awards

[1] Best Paper Award for a conference paper:
Duy D. and Tanaka K., "A Novel Task-to-Processor Assignment Approach for Optimal Multiprocessor Real-Time Scheduling," the IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), September 2018, Hanoi, Vietnam.

[2] Best Paper Award for a conference:
Duy D. and Tanaka K., "An Effective Approach for Improving Responsiveness of Total Bandwidth Server," the 8th International Conference on Information and Communication Technology for Embedded Systems (IC-ICTES 2017), May 2017, Chonburi, Thailand.