Stack Stealing

Chen Shih
Hochschule Hamm-Lippstadt
Electronic Engineering
Lippstadt, Germany
shih.chen@hshl.stud.de

***Abstract*—This paper presents an algorithm for scheduling mixed deadline tasks in real-time systems, which is called "Slack stealing". Periodic tasks and aperiodic tasks are processed according to the priorities, the slack stealing technique enables a multitasking system to make use of the slacks in periodic intervals and allocate them to the non-critical aperiodic tasks. The system actively computes and updates the amount of slack to ensure the correctness of scheduling results. By utilizing idle time in the system, the algorithm "Slack stealing" minimizes the average response time of aperiodic tasks, thereby improving overall scheduling performance.**

***Index Terms*—Slack, Hard real-time, aperiodic**

I. Introduction

Modern real-time systems are widely used in various application domains to cope with computationally intensive requirements. Typically, a set of hard deadline tasks in real-time systems must complete the execution within a specific time duration. Failure to meet these hard deadlines leads to catastrophic consequences. For example, the system must perform the acquisition of the sensor signal periodically to guarantee stability and correctness. Apart from the periodic tasks, there is another aperiodic instance that takes place due to external events such as inputs from the user interface. Although these tasks do not have certain rigid deadlines, they require a prompt response to achieve the desired execution. The deadline of the periodic tasks is set conservatively to avoid missing deadlines, e.g. using Worst-case execution time(WECT), even though they are usually completed earlier than expected. The conservative estimate of the deadline leads to idle time in intervals. One can think of slack as the time left over after periodic work. The slack stealer of the slack algorithm uses the slack time to minimize the average response time of aperiodic tasks without putting hard real-time periodic tasks at risk.

II. Slack Stealing Algorithm

Slack Stealing Algorithm enables scheduler in a system schedule aperiodic tasks in slack time among periodic tasks and sporadic tasks. Scheduler keeps track of periodic tasks in order to calculate the amount of available slacks and consumed slacks, thus ensuring that time-critical tasks can be deferred without the risk of missing deadlines. These aperiodic tasks are handled by a slack stealer, which has the highest execution priority when there is any aperiodic task in the queue. If the queue is empty, the slack stealer is given the lowest priority and the execution is suspended. Slack stealing algorithm is called greedy since it constantly look for available slacks to schedule aperiodic tasks in the queue as soon as possible.

III. Slack-Stealing in Clock-driven system

A. Slack-Stealing in Clock-driven system

Safety-critical applications such as railroad systems, aircraft, or nuclear power plants are highly dependent on stable system performance, where every execution must be preprocessed and guaranteed to be absolutely correct. Due to the high degree of determinism, cyclic executives are therefore suitable candidates for such applications as described above. They are rather simple, reliable, and anomaly-free. Clock-driven scheduling is the foundation of cyclic executives, and it is applicable when the task parameters and resource requirements are known in advance. The system computes the schedule offline and stores them for use at run-time, and this scheduling decision is made periodically. By the initialization of each instance, the scheduler chooses and dispatches tasks. Clock-driven scheduling promises that all the periodic hard real-time jobs can be finished by their deadline. Nevertheless, the scheduler is also required to handle aperiodic tasks triggered by external events. Typically, hard real-time tasks have higher priority than aperiodic tasks, which means aperiodic tasks are scheduled in the background and processed after the completion of all hard real-time tasks. Typically, hard real-time tasks are prioritized over aperiodic tasks. Aperiodic tasks are scheduled in the background and delayed until all the tasks with a hard real-time deadline are completed. The delay of execution can lead to an undesirably long response time for aperiodic tasks, which becomes a design issue in the clock-driven system. Therefore an approach is proposed to resolve this problem. The approach is called Slack-stealing, and its objective is to minimize the response time of periodic tasks. In real-time systems, completing a hard real-time task early is pointless, as long as it finishes by its deadline. Note that the sooner the aperiodic tasks are processed, the sooner the system responds. Since the early completion really brings no benefit but reduces efficiency, the strategy is to safely delay those tasks without missing the deadline and give higher execution priority to aperiodic tasks whenever possible [1].

B. Basic slack computation in clock-driven system

Assuming that the amount of time allocated to slice in the frame F_k is denoted by X_k, the slack during frame F_k can be expressed as S_k in the following formula: S_k = F_k - X_k.

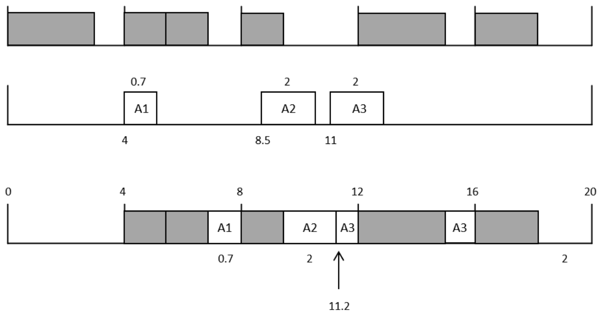

Fig. 1. background scheduling in clock-driven system

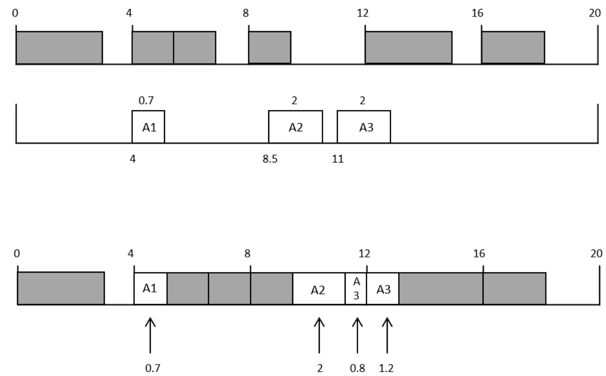

Fig. 2. slack scheduling in clock-driven system

Whenever S_k is large than 0, in other words, the aperiodic tasks queue is nonempty, the cyclic executive gives Slack stealer the highest execution priority to process aperiodic tasks. As long as S_k is non-zero, tasks will never miss any deadline. Cyclic executives must keep track of the number of available slack and the consumed slack, and this can be done by an interval timer. At the beginning of each cycle, the interval timer is set to the value of the initial slack in the frame. The timer counts down as Slack stealer consumes the available slack. When the value comes to 0, the timer expires and the cyclic executive suspends Slack stealer, returning the highest execution priority to periodic tasks. Note that some senior operating systems do not support high-resolution timers, so this method is only applicable if the system can guarantee granularity and accuracy in hundreds of milliseconds or seconds. After periodic tasks finish by their deadline, before the next instance, the scheduler again examines the aperiodic task queue and repeats the same procedure [1]. Figure 1 provides an illustrative example of background scheduling in clock-driven system. The gray-shaded blocks in the top queue stand for hard real-time tasks which are executed periodically. Cyclic executives initiate a new cycle every 4 time units. Beneath the top queue lies the aperiodic tasks queue. The first aperiodic task arrives at the time 4 and has 0.7 units of execution time. The second and the third aperiodic tasks come at the time 8.5 and 11 respectively, and both execute for 2 time units. As shown in the figure, the scheduler dispatches periodic tasks first and starts executing aperiodic tasks afterward. The first aperiodic task A1 arrives at time 4 and completes execution at time 8, which requires a response time of 4 units. The response time for A2 is 2.7 time units. Task A3 starts executing 0.8 time units after A2, and it is preempted due to the deadline. In the next frame, A3 resumes execution and finishes at time 16, with a response time of 4.8.

In Figure 2, the scheduler adopts the slack stealing approach to process the same task queues. Instead of prioritizing the hard real-time tasks first, the scheduler defers them and executes aperiodic tasks first to achieve a shorter response time. At time 4, a new execution cycle begins with the arrival of task A1. The aperiodic task queue is non-empty so the scheduler dispatches A1 first. After 0.7 time units, the available slack time is completely consumed, so the scheduler suspends the stack stealer and continues with the remained periodic tasks in the frame. At time 8 starts the new frame, the periodic task executes directly since the aperiodic task queue is empty at this time. At time 10, A2 takes over the priority of execution and proceeds with A3 after its completion. Finally, A3 pauses at time 12 and resumes execution, finishing at 13.7.

C. Background Scheduling and Slack stealing

A comparison is made to show the difference in response time of background scheduling and slack stealing. By and large, aperiodic tasks have a relatively short response time when slack stealing is adopted.(see figure.3) As described in the previous section, clock-driven systems are widely deployed in safety-critical applications due to their high level of reliability and predictability. However, these characteristics also imply that clock-driven systems are rather inflexible. Any changes in temporal parameters, even minor ones, require a complete rescheduling. The constant modifications of systems would increase the cost of development and time to market, which are undesirable. Therefore, for systems with diverse timing parameters and resource requirements, an alternative scheduling approach must be introduced. Priority-driven systems refer to the system that schedules tasks based on their deadline or other timing constraints. Rather than doing pre-calculation, the priority-driven approach makes scheduling decisions when pre-emption is allowed, i.e when the release or completion of a task occurs. In principle, priority-driven algorithms always make locally optimal scheduling decisions, trying to exploit any possible resource, which is described as greedy. Moreover, priority-driven algorithms never intentionally leave resources idle, as they would contradict the concept of locally optimal decisions. The term starvation refers to a phenomenon regarding low-priority tasks waiting indefinitely in a priority-driven schedule. In some cases, the scheduler lets tasks with lower priority wait indefinitely. For instance, in a system that is heavily loaded with high-priority tasks, the completion time of low-priority tasks can be difficult to estimate [1].

IV. Slack stealing in Priority-driven system

In principle, slack stealing in a priority-driven system works in a similar way to how it works in a clock-driven system.

	A1	A2	A3
Background Scheduling	4	2.7	4.8
Slack Stealing	0.7	2.7	1.7

Fig. 3. Response time of background scheduling and slack stealing

The core function is to compute and keep track of the amount of slack in the systems. However, slack stealing is computationally more complex in a priority-driven system. Figure 3 gives an example of how slack stealing contributes to improving the response time of aperiodic tasks in a priority-driven system, where the scheduling of periodic tasks is on the the basis of earliest-deadline-first (EDF) algorithm [1].

Consider that an EDF system consists of two periodic tasks T1(4,1.5) and T2(6,0.5). Aperiodic tasks A1 and A2 are released at time 2.5 and time 5.5 respectively. At the start, the aperiodic task queue is empty so the Job J(2,1) executes first. The slack stealer resumes execution as the first aperiodic task arrives at time 2.5, which has an execution time of 1 time unit. After A1 is finished, J(1,1) proceeds with the rest of the execution which is postponed to time 3.5. The second aperiodic task A2 is released shortly after, and the slack stealer continues because the periodic task queue is empty by then. The available slack is completely consumed at time 6.5 and the slack stealer is suspended. The scheduler carries on executing periodic task J(1,2) and finishes at time 8. At the beginning of the next frame, the slack stealer is again freed from suspension and then completes the rest part of A2.

V. Slack computation

Slack computation algorithms refer to the algorithms that calculate and update the amount of available slack in the system. A slack computation algorithm is considered correct when it correctly reports the presence of a slack. An incorrect computation may result in a missed deadline, which is not acceptable in any real-time system. Slack computation can be either static or dynamic. In the case of a clock-driven system, the slack is calculated using a static computing approach. The scheduler calculates all the slack based on given temporal parameters and other information regarding periodic tasks before run-time. This is called offline computation. During run-time, the scheduler only needs to update the current information. Therefore, the static approach has a relatively low run-time overhead. However, the problem with this approach is that it has a rather low jitters tolerance. When the actual release time of a periodic task differs from the release time used to generate the message, the pre-computed slack may be incorrect, leading to serious consequences. A dynamic approach calculates the amount of slack during the run-time and it is applied when the periodic tasks have a wide range of inter-release time. High run-time overhead is an apparent drawback of dynamic slack computation. However, the dynamic approach can be integrated with unused processor run-time and task overruns [1].

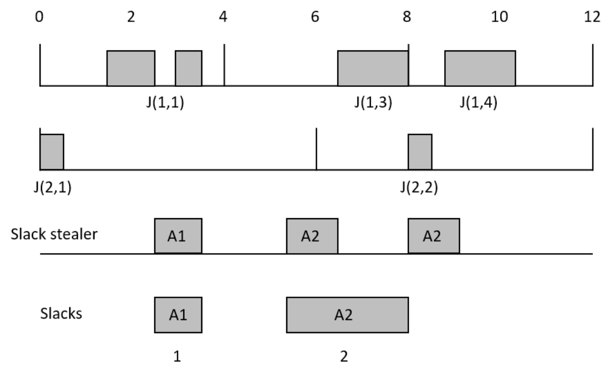

Fig. 4.

A. Static-slack computation

As the number of tasks increases, the complexity of the slack calculation can be quite high. Figures 4 and 5 give an illustrative example of how the relaxation calculation is performed and the problems that exist. In Figure 4, a system is performing two periodic tasks T_1 (4,2) and T_2 (6,2.75). Since the total execution time required to complete these two tasks by the deadline of $J_{2,1}$ is 1.25 (allocated time - execution time of T_1 - execution time of T_2), it is intuitive to assume that this system has a slack of 1.25 time units.

However, this assumption is only correct within this time slot. As the time scale expands, it may lead to a false slack computation, causing the deadlines to be missed. For example, in Figure 5, where the system takes $J_{1,3}$ and $J_{2,2}$ into account, there may be a situation where $J_{1,3}$ misses the deadline because the system assumes that the available slack time is 1.25 time units, when in fact it is only 0.5 time units.

An optimal static-slack computation algorithm is proposed as a convenient way to analyze the complexity of the calculation of slack in the system. The algorithm contains n periodic tasks in an N square size table and assumes that the system actually releases the periodic tasks periodically. (see Figure 6) For a better understanding of the algorithm, the boundaries of the tasks can be ignored, i.e., it does not matter to which task each job belongs. It creates a hyper period that contains jobs labeled with a corresponding serial number, e.g. J_1,J_2, J_3,...J_k. The algorithm represents the deadline of each job as d_n (n = n_{th} job) and indexes all jobs in non-increasing priority order. Consider t_c to be the time at which the slack calculation occurs and $\sigma_i(t_c)$ to be the amount of slack in a periodic job at time t_c. The amount of slack $\sigma_i(t_c)$ is the result of the total allocated time minus the total time required for the selected job to complete its execution. The slack of individual jobs changes over time once the system starts execution. However, the processor idles, execution of lower priority jobs, and the execution of the slack stealer are not taken into account in the pre-computed initial slacks. Therefore, if any of the above events occurs, it takes up a portion of the slack, making the total amount of pre-computed slack smaller [1].

In Fig. 6, the execution of job J_6 lasts 0.5 time units before

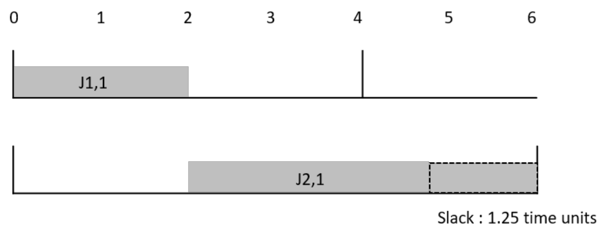

Fig. 5.

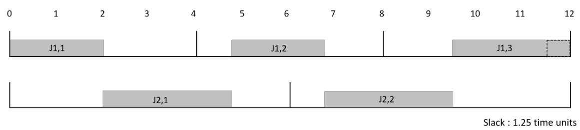

Fig. 6.

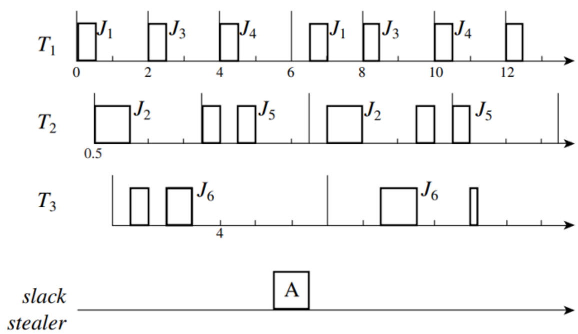

Fig. 7.

W(i; j) (i \ j)	1	2	3	4	5	6
1	1.5	1.5	1.5	1.5	1.5	1.5
2		2	2	2	2	2
3			2	2	2	2
4				3.5	3	2.3
5					3	2.3
6						2.3

Fig. 8. slack table

time 2, but these 0.5 time units are not taken into account by the pre-computed initial slack of J_3. At time 2, the slack of J_3 has a value of 1.5 instead of 2 units of time. In addition, since the slacker performs 0.5 time units of work at the beginning of the second time period, it reduces the total slack by 2 time units for all jobs in this time period.

B. practical consideration

In general, there are two important factors to consider when analyzing the implementation of static slack computation: the effect of phase and the effect of release time jitters. Through observing the pre-computed static-slack graph it can be seen that the end of the first hyper period may differ from the end of a busy interval of the periodic tasks in the absence of aperiodic jobs. Therefore, when the periodic tasks in the system are out of phase, ignoring this difference will lead to an incorrect calculation of the initial amount of slack, which will result in missed deadlines. As it exemplifies in figure 7 when the hyper period and the busy interval both end at the same time, the schedule executes cyclically with a fixed period without aperiodic tasks and release-time jitters. If the mentioned conditions are satisfied, it is sufficient to compute the slack in the first hyper period and apply them to all times in the system. On the flip side, when the hyper period comes to the end early than the busy interval, there exists an initial transient segment that ends at the end of the first busy interval and is proceeded with a fixed cyclic period [1]. Release-time jitters are another common issue existing in periodic task models and priority-driven algorithms. In most cases, when the jitters are of negligibly small size, the priority of jobs remains correct and does not require any change. Considering that the jitters of the release time is relatively larger than the interval between the deadlines, the pre-calculated values will no longer be correct and the jobs in the system will need to be rescheduled. Again figure 7 provides insights into this issue. Assume that periodic job J_3 is postponed till time 2.1 whereas the job J_4 is delayed to time 4.3. Except for J_3 and J_4, the rest of the jobs in the hyper period are released as scheduled. According to the pre-computed slack table, the initial slack for the jobs J_1, J_3, and J_4 are 1.5, 2, and 3.5 respectively. While the deadline for job J_1 is still time 2, the deadline for J_3 and J_4 are 4.1 and 6.3. Assuming that J_5has a small delay (0.1 units of time), the actual slack of J_5 would have an additional slack of 0.1 compared to the result of the pre-computed slack table. In this case, despite the release time being delayed, the results given by the pre-computed slack table are still applicable to the rest of the jobs. In this case, the results given by the pre-computed slack table are still applicable to the rest of the jobs, despite the delay in the release of J_5. This creates a lower bound of the actual initial slack in the system [1].

C. Dynamic-slack computation

To cope with the unavoidable release-time jitters, the system would make use of dynamic computation in run time. It is achieved by computing lower bounds of the slack without relying on a priori information. The following equation describes how the dynamic slack computation algorithm estimation of the busy interval is done in two separate steps. The currently executing job is marked as J_{ci}, and its completed part is ξ_{ci}. AS the name of dynamic slack computation implies, the scheduler always keeps the value of ξ_{ci} up-to-date. Considering that

$\overline{\phi}_i$ is the release time of the next earliest incoming job, the scheduler updates $\overline{\phi}_i$by summing the actual release times of pi and J_{c_i} at the time of job release. The first step is computing the length of time X from the periodic tasks to the end of the current busy interval. X reaches its maximum value if all the jobs in the existing tasks are released periodically with period pi. $w_{(x)}$ on the left-hand side of the equation provided represents the maximum processor time required for all the periodic jobs within the interval (t_c,t_c+x). The right-hand side of the equation consists of two sums, the first one shows the amount of time remaining to execute all current periodic jobs at time t_c, while the second sum is the maximum execution time of all released jobs in the interval (t_c,t_c+x). By solving the equation w(x) = x, it can be found that the length x has an equal value to the minimum solution of the equation [1].

$$w(x) = \Sigma_{i=1}^{n}(e_i - \xi_i) + \Sigma_{i=1}^{n}\left[\frac{(t_c + x - \overline{\phi_i})u_{-1}(t_c + x - \overline{\phi_i})}{p_i}\right]$$

VI. Implementation of Slack stealer

This paper presents an implementation of slack stealer on the basis of a clock-driven system in which periodic and aperiodic tasks coexist. Periodic tasks are performed regularly to keep the system operating under normal, stable, and correct circumstances. Missing the deadlines of these hard real-time periodic tasks leads to system failures. Aperiodic tasks stimulated by external instances need to be responded to as quickly as possible to achieve desirable performance. The activity diagram gives a graphical presentation of a step-wise workflow of the scheduler. The execution first starts from a decision node. The scheduler checks whether the aperiodic task queue is empty or not. If there are any aperiodic tasks waiting in the queue, the scheduler sets the counter to the slack value for the current interval and lets the slack stealer executes the task. When the counter counts down to 0 and expires, the slack stealer is suspended while the scheduler continues to execute the periodic tasks and completes them before the deadline. On the other hand, the state transition of the scheduler is described by the state machine diagram (see Figure x). It contains 3 different states, namely idle, non-periodic tasks, and periodic tasks. In the Idle state, no tasks are being performed at this time. For example, in the absence of aperiodic tasks, the scheduler is idle when the periodic tasks are completed.

VII. Summary and conclusion

In summary, the slack stealing technique was proposed to improve the scheduling in the presence of aperiodic instances. It contributes to forming a feasible schedule by improving the response time of aperiodic tasks without causing the hard real-time tasks to miss their deadlines. It is commonly applied in both clock-driven domains and priority-driven domains to cope with mixed task scheduling. In a clock-driven system, the slack is determined using static computation before execution, while in a priority-driven system, the slack is computed dynamically during run time.

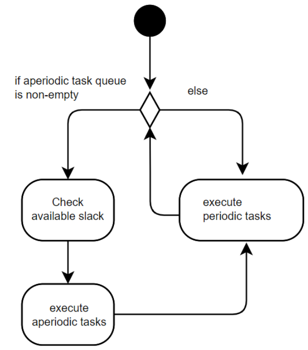

Fig. 9. Activity-diagram

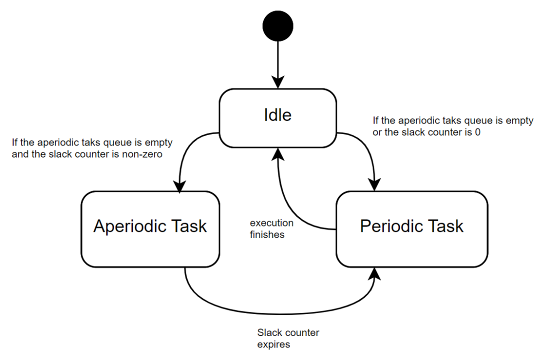

Fig. 10. -State-machine-diagram

References

[1] Liu Jane W S. *Real-time systems*. Pearson Education, 2004.