# Slack Stealing

Gordan Konevski
gordan.konevski@stud.hshl.de
Hochschule Hamm-Lippstadt, 2021

***Abstract*—This paper deals with the uses and relevancy of slack stealing in embedded real-time systems. It ultimately aims to define the environment in which it is typically used as a method and, through some examples, show its use in practice, in order to prove it as a viable and useful method.**

## I. Real-Time Systems

### A. Basic Concept

Typically software is seen through the perspective of an interactive system, in which the user issues commands that are (typically) visibly displayed. Interactive software, as it is referred to, is always subject to delays. It is this time delay that we wish to distinguish from software that is based on real-time systems. We may define it as such

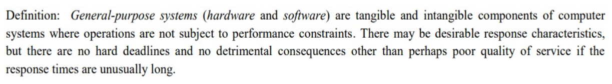

Definition: *General-purpose systems* (*hardware* and *software*) are tangible and intangible components of computer systems where operations are not subject to performance constraints. There may be desirable response characteristics, but there are no hard deadlines and no detrimental consequences other than perhaps poor quality of service if the response times are unusually long.

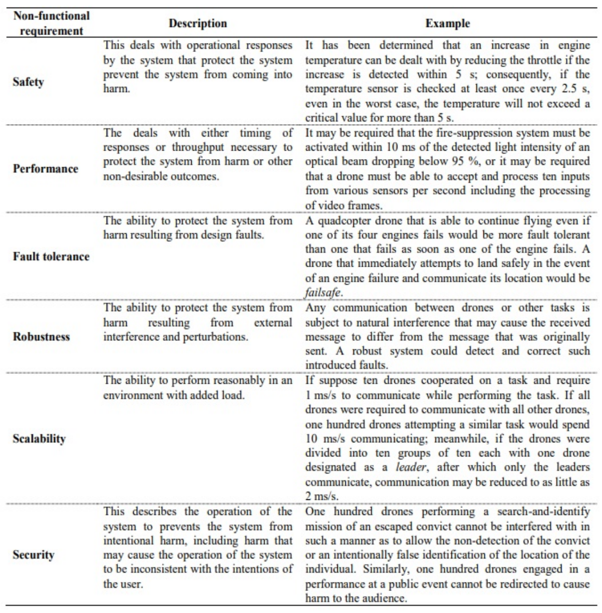
Fig. 1. General Definition

In contrast with general-purpose systems, real-time systems are meant to monitor, interact with, control, or respond to the physical environment. The interface is set-up through sensors, communications systems, actuators, and other input and output devices. Under such circumstances, it is necessary to respond to incoming information in a timely manner. Delays may prove dangerous or even catastrophic. Consequently, we will define a real-time system as one where:

- the time at which a response is delivered is as important as the correctness of that response. So whether a command is executed within the specified time range is as important as the command itself, and a failure to execute the command on time is regarded as a fatal error.
- the consequences of a late response are just as hazardous as the consequences of an incorrect response.

Those requirements that describe how the system should respond to a given set of inputs (both from sensors and messages received from communication systems) given the current state of the system and what the expected outputs (both signals to actuators and messages sent through communication systems) and changes of state of the system are described as functional requirements. Other non-functional requirements may include availability, configurability and regulatory compliance. Real-time systems are not meant to be fast, per se; instead, they should be just fast enough to ensure that all functional requirements and non-functional requirements including, but not limited to, performance requirements. Some examples of real time systems include:

- transportation, for e.g. control systems for vehicles, spacecrafts, ships etc.
- telecommunication
- building management: security, heating, ventilation, air conditioning and lighting
- production control in an industrialized environment



| Non-functional requirement | Description | Example |
|---|---|---|
| **Safety** | This deals with operational responses by the system that protect the system prevent the system from coming into harm. | It has been determined that an increase in engine temperature can be dealt with by reducing the throttle if the increase is detected within 5 s; consequently, if the temperature sensor is checked at least once every 2.5 s, even in the worst case, the temperature will not exceed a critical value for more than 5 s. |
| **Performance** | The deals with either timing of responses or throughput necessary to protect the system from harm or other non-desirable outcomes. | It may be required that the fire-suppression system must be activated within 10 ms of the detected light intensity of an optical beam dropping below 95 %, or it may be required that a drone must be able to accept and process ten inputs from various sensors per second including the processing of video frames. |
| **Fault tolerance** | The ability to protect the system from harm resulting from design faults. | A quadcopter drone that is able to continue flying even if one of its four engines fails would be more fault tolerant than one that fails as soon as one of the engine fails. A drone that immediately attempts to land safely in the event of an engine failure and communicate its location would be *failsafe*. |
| **Robustness** | The ability to protect the system from harm resulting from external interference and perturbations. | Any communication between drones or other tasks is subject to natural interference that may cause the received message to differ from the message that was originally sent. A robust system could detect and correct such introduced faults. |
| **Scalability** | The ability to perform reasonably in an environment with added load. | If suppose ten drones cooperated on a task and require 1 ms/s to communicate while performing the task. If all drones were required to communicate with all other drones, one hundred drones attempting a similar task would spend 10 ms/s communicating; meanwhile, if the drones were divided into ten groups of ten each with one drone designated as a *leader*, after which only the leaders communicate, communication may be reduced to as little as 2 ms/s. |
| **Security** | This describes the operation of the system to prevents the system from intentional harm, including harm that may cause the operation of the system to be inconsistent with the intentions of the user. | One hundred drones performing a search-and-identify mission of an escaped convict cannot be interfered with in such a manner as to allow the non-detection of the convict or an intentionally false identification of the location of the individual. Similarly, one hundred drones engaged in a performance at a public event cannot be redirected to cause harm to the audience. |

Fig. 2. Some Non-Functional Requirements of Real-Time Systems

### B. Defining Time

A highly technical definition of time would be that time is a natural phenomenon where one "second" is the duration of exactly 9192631770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom at rest at a temperature of 0 Kelvin, as defined by the Bureau international des poids et mesures. With the exception of the kilogram, all other units are defined relative to the second. Atomic clocks are used to measure time, and coordinated universal time (UTC) is an international standard for time. But most systems, however, use quartz clocks, where a quartz crystal is carved to vibrate at 215 Hz = 32768 Hz when an electric field is placed across it. A 5-bit digital counter will overflow once per second as it counts the oscillations. With 86400 s/day, such clocks tend to drift less than 1 s/day and therefore different systems will have different times even if they start synchronized (more expensive crystals will have less drift). Depending on the system in place, the accuracy of the clock vary in their importance. Naturally, systems built for e.g. astrophysical research require atomic clocks as opposed to quartz clocks.

## II. Slack Stealing

A big portion of the scheduling algorithms used deal with homogeneous sets of tasks, where all computational activities are either aperiodic or periodic. It may just be the case, however, that certain systems require both types of processes, which may also differ in their criticality. Typically, periodic tasks are time-driven and execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates. Aperiodic tasks, on the other hand, are usually event-driven and may have hard, soft, or non-real-time requirements depending on the specific application. When dealing with hybrid task sets, the main objective of the kernel is to guarantee the schedulability of all critical tasks in worst-case conditions and provide good average response times for soft and non-real-time activities. Off-line guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is, by assuming a maximum arrival rate for each critical event. This implies that aperiodic tasks associated with critical events are characterized by a minimum interarrival time between consecutive instances, which bounds the aperiodic load. Aperiodic tasks characterized by a minimum inter-arrival time are called sporadic. They are guaranteed under peak-load situations by assuming their maximum arrival rate. If the maximum arrival rate of some event cannot be bounded a priori, the associated aperiodic task cannot be guaranteed off-line, although an online guarantee of individual aperiodic requests can still be done. Aperiodic tasks requiring online guarantee on individual instances are called firm. Whenever a firm aperiodic request enters the system, an acceptance test can be executed by the kernel to verify whether the request can be served within its deadline. If such a guarantee cannot be done, the request is rejected. In the next sections, we present a number of scheduling algorithms for handling hybrid task sets consisting of a subset of hard periodic tasks and a subset of soft aperiodic tasks. All algorithms presented in this chapter rely on the following assumptions:

- Periodic tasks are scheduled based on a fixed-priority assignment; namely, the Rate-Monotonic (RM) algorithm;
- All periodic tasks start simultaneously at time t = 0 and their relative deadlines are equal to their periods.
- Arrival times of aperiodic requests are unknown.
- When not explicitly specified, the minimum inter-arrival time of a sporadic task is assumed to be equal to its deadline.
- All tasks are fully preemptable.
- Aperiodic scheduling under dynamic priority assignment is discussed in the next chapter.

The Slack Stealing algorithm is another aperiodic service technique, which offers substantial improvements in response time over alternative service methods. Unlike such methods, the Slack Stealing algorithm does not create a periodic server for aperiodic task service. Rather it creates a passive task, referred to as the Slack Stealer, which attempts to make time for servicing aperiodic tasks by "stealing" all the processing time it can from the periodic tasks without causing their deadlines to be missed. This is equivalent to stealing slack from the periodic tasks. We recall that if ci(t) is the remaining computation time at time t, the slack of a task Ti is:

$$slack_i(t) = d_i - t - c_i(t).$$

Fig. 3. Slack Stealing Task

The main idea behind slack stealing is that, typically, there is no benefit in early completion of the periodic tasks. Hence, when an aperiodic request arrives, the Slack Stealer steals all the available slack from periodic tasks and uses it to execute aperiodic requests as soon as possible. If no aperiodic requests are pending, periodic tasks are normally scheduled by Rate Monotonic scheduling. Figure 5 shows the behavior of the Slack Stealer on a set of two periodic tasks, T1 and T2, with periods T1 = 4, T2 = 5 and execution times C1 = 1, C2 = 2. In particular, Figure 5 a) shows the schedule produced by RM when no aperiodic tasks are processed, whereas Figure 5 b) illustrates the case in which an aperiodic request of three units arrives at time t = 8 and receives immediate service. In this case, a slack of three units is obtained by delaying the third instance of T1 and T2. Note that in the example of Figure 5, no other server algorithms can schedule the aperiodic requests at the highest priority and still guarantee the periodic tasks. For example, since U1 = 1/4 and U2 = 2/5, the P factor for the task set is P = 7/4; hence, the maximum server utilization, according to Figure 4 is:

$$U_{SS}^{max} = \frac{2}{P} - 1 = \frac{1}{7} \simeq 0.14.$$

Fig. 4. Maximum Server Utilization

This means that, even with Cs = 1, the shortest server period that can be set with this utilization factor is Ts = [Cs/Us] = 7, which is greater than both task periods. Thus, the execution of the server would be equivalent to a background service, and the aperiodic request would be completed at time 15.

In order to schedule an aperiodic request Ja(ra, Ca) according to the Slack Stealing algorithm, we need to determine the earliest time t such that at least Ca units of slack are available in [ra, t]. The computation of the slack is carried out through the use of a slack function A(s, t), which returns the maximum amount of computation time that can be assigned to

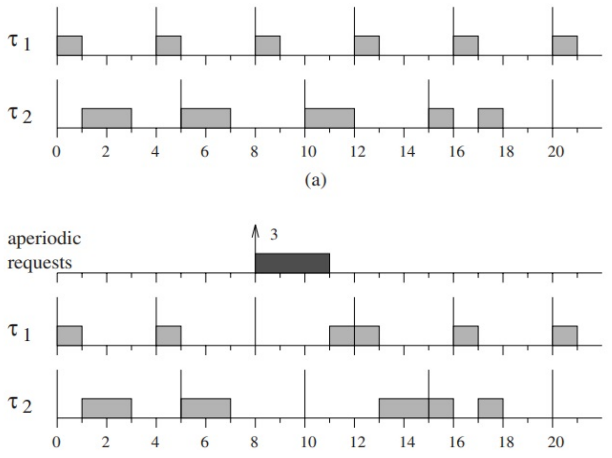


Fig. 5. Slack Stealing Behavior

aperiodic requests in the interval [s, t] without compromising the schedulability of periodic tasks. Figure 6 shows the slack function at time s = 0 for the periodic task set considered in the previous example. For a given s, A(s, t) is a non-decreasing step function defined over the hyperperiod, with jump points corresponding to the beginning of the intervals where the slack is available. As s varies, the slack function needs to be recomputed, and this requires a relatively large amount of calculation, especially for long hyperperiods. Figure 7 shows how the slack function A(s, t) changes at time s = 6 for the same periodic task set. According to the original algorithm, the slack function at time s = 0 is precomputed and stored in a table. During runtime, the actual function A(s, t) is then computed by updating A(0, t) based on the periodic execution time, the aperiodic service time, and the idle time. The complexity for computing the current slack from the table is O(n), where n is the number of periodic tasks; however, depending on the periods of the tasks, the size of the table can be too large for practical implementations. We can also use an alternative to this algorithm, where the

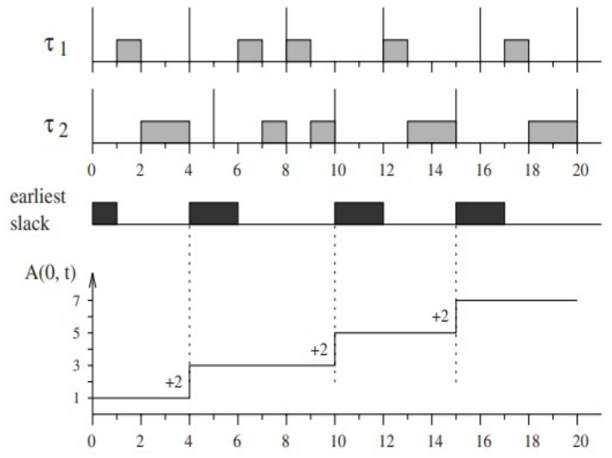


Fig. 6. Slack Stealing Behavior at Time s = 0

available slack is computed whenever an aperiodic requests enters the system. This method is more complex than the previous static approach, but it requires much less memory and allows handling of periodic tasks with release jitter or synchronization requirements.

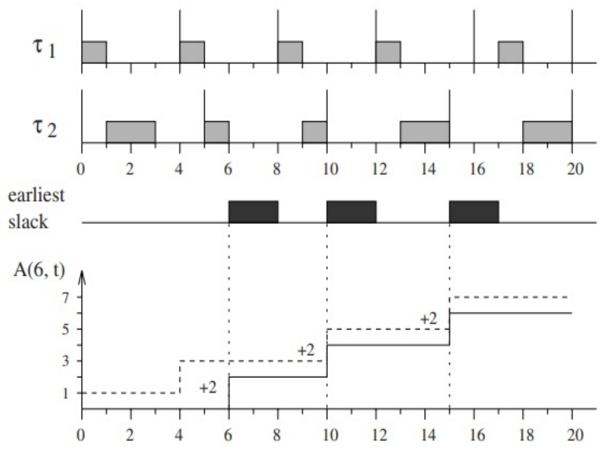
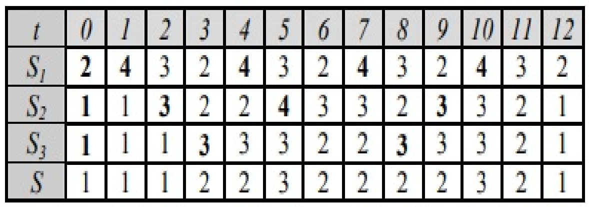


Fig. 7. Slack Stealing Behavior at Time s = 6

## III. Fast Slack Method

### A. Application

The Fast Slack method is an alternative use of slack stealing developed by Jose Manuel Urriza, whose example this paper will use, where a set of counters is implemented during runtime in order to reduce the computational cost of the Fast Slack method. A set of counters keeps track of the slack which may be stolen at each priority level. The slack available to



| *t* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *$S_1$* | **2** | **4** | 3 | 2 | **4** | 3 | 2 | **4** | 3 | 2 | **4** | 3 | 2 |
| *$S_2$* | **1** | 1 | **3** | 2 | 2 | **4** | 3 | 3 | 2 | **3** | 3 | 2 | 1 |
| *$S_3$* | **1** | 1 | 1 | **3** | 3 | 3 | 2 | 2 | **3** | 3 | 3 | 2 | 1 |
| *S* | 1 | 1 | 1 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 3 | 2 | 1 |

Fig. 8. Fast Slack Showcase Table

execute non-critical tasks will be the minimum value of the counters. By analyzing how the slack varies during runtime, we can state the following rules to manage the counters:

- When a lower priority task is executed, higher priority counter should be decremented in an amount equal to the time executed.
- When a non-critical task is executed or there exists an idle time, all the counters should be decremented in an amount of time equal to the time consumed.
- When a task finishes its execution, the Fast Slack method is calculated to initialize its counter.
- When a task finishes its execution before its Worst Case Execution Time, all lower priority counters should be

incremented in a time equal to the Worst Case Execution Time minus the time that the task used to complete.

We can note that if the system is schedulable, then no counter can hold a negative value. In Figure 8 we can see the counters of the system of the example in the previous section. The bold numbers are basically indicating that the slack was calculated because the task finished its execution. The last row shows the slack available to execute a non-critical task.

## B. Results

In order to compare Stack Stealing to other methods, three different groups of tasks are used:

- Group A: comprised of 10 real-time tasks. The utilization factor of the systems ranges from 0.4 to 0.9 with steps of 0.1, and there are 200 real-time systems for each utilization factor. The periods of the tasks follow an exponential composition; 4 tasks with periods in the range 25 to 100 units, 3 tasks with periods between 100 and 1000 units and a further 3 tasks with periods between 1000 and 10000 units.
- Group B: comprised of 20 real-time tasks. The utilization factor of the systems ranges from 0.4 to 0.9 with steps of 0.1, and there are 200 real-time systems for each utilization factor. The periods of the tasks follow an exponential composition; 7 tasks with periods in the range 25 to 100 units, 7 tasks with periods between 100 and 1000 units and a further 6 tasks with periods between 1000 and 10000 units.
- Group C: comprised of 50 real-time tasks. The utilization factor of the systems ranges from 0.5 to 0.9 with steps of 0.1, and there are 200 real-time systems for each utilization factor. The periods of the tasks follow an exponential composition; 17 tasks with periods in the range 25 to 100 units, 17 tasks with periods between 100 and 1000 units and a further 16 tasks with periods between 1000 and 10000 units.

Each real-time system was generated as follows. First, the periods of the tasks were chosen at random from the desire range. Deadlines were set equal to the periods. The tasks were then sorted into deadline monotonic priority order. Next, random execution times were assigned, highest priority first. The computation times were constrained such that the partial task remained feasible according to a sufficient and necessary schedulability test. Finally tasks sets with an utilization level differing by more than 0.5% from that required were discarded. We simulated each real-time system for 15 consecutives releases of the lowest priority tasks after the worst case of load. We counted the number of iterations that each algorithm needs to get the slack available. The results presented are the averages over each group. Figure 9 shows the average number of iterations for each utilization factor of groups A, B and C that Fast Slack and the alternative slack-stealing mechanism required to get the slack available. We can note that the number of iterations that Fast Slack requires is much less for low utilization factors and remains lower for high utilization factors. From these results we can conclude that the number of iterations that Fast Slack requires increases as the utilization factor increases because the inspection interval increases as well and consequently the equation should be calculated more times. The number of iterations of the method proposed previously decreases as the utilization factor increases because it calculates the slack based on the intervals when the system is idle. At low utilization factors the systems is idle most of the time and consequently the mechanism has to be evaluated at each one of these idle intervals. As the utilization factor increases, the number of idle intervals decreases and then the number of iteration decreases. The simulations were performed considering integer execution times. When fractional executions times are considered, then the Davis's method (as an alternative to the Slack Stealing one) increases the number of iterations whilst the Fast Slack method remains with the same performance.

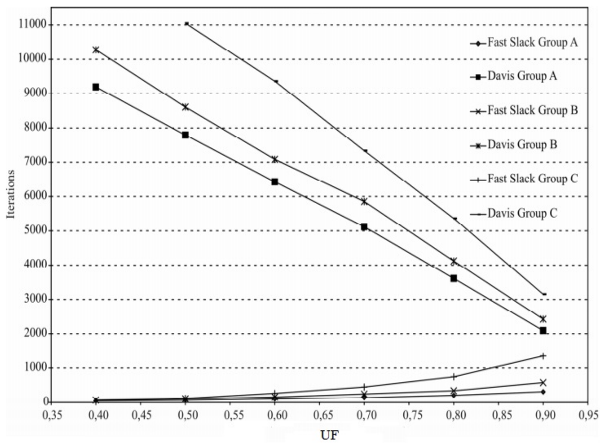


Fig. 9. Comparison of Fast Slack and Other Methods

## IV. Approximate slack stealing algorithms

### A. Delineation

Another useful topic to discuss is approximations to the optimal dynamic algorithm. The objective is to produce slack stealing algorithms which are efficient enough for runtime usage. First we consider stealing slack from purely periodic task sets. We indicate how our analysis can be used in an algorithm which replicates the behavior of an optimal Slack Stealer. We then present an approximate algorithm which can be used to steal slack from both hard deadline periodic and sporadic tasks. We compare the performance of this algorithm to the optimal and background processing methods. Finally, we discuss the overheads of the approximate algorithm and their justification in terms of improved soft task response times. For hard deadline periodic task sets, the slack available at priority level i only increases when task i completes. We may therefore replicate the behavior of an optimal Slack Stealer by using an algorithm (Figure 10) to calculate Smax at each completion of task i. Figure 11 (a and b) are then used to keep track of the slack available at other times. The

```
$S_{i,t} = 0$
$w_{i,t}^{m+1} = 0$
do  while  $w_{i,t}^{m+1} \le d_{i,t}$
    $w_{i,t}^{m} = w_{i,t}^{m+1}$
    $w_{i,t}^{m+1} = S_{i,t} + \sum_{\forall j \in hp(i) \cup i} \left[ c_{j,t} + \left\lceil \frac{(w_{i,t}^{m} - x_{j,t})_0}{T_j} \right\rceil C_j \right]$
    if  $w_{i,t}^{m} = w_{i,t}^{m+1}$
    then  $S_{i,t} = S_{i,t} + v_{i,t}(w_{i,t}^{m}) + \varepsilon$
          $w_{i,t}^{m+1} = w_{i,t}^{m+1} + v_{i,t}(w_{i,t}^{m}) + \varepsilon$
    endif
$S_{i,t}^{\max} = S_{i,t} - \varepsilon$
```

Fig. 10. Task i Algorithm

overhead of computing the slack can be reduced by a priori calculation of the least additional slack, Sadd, which becomes available at every completion of task i. This enables a less pessimistic initial value for Sid to be used in Figure 10, thus reducing computation. The least additional level i slack is

a)

$$\forall j \in hp(i) \cup lp(i): \quad S_{j,t'}^{\max} = S_{j,t}^{\max} - (t' - t)$$

b)

$$\forall j \in hp(i): \quad S_{j,t'}^{\max} = S_{j,t}^{\max} - (t' - t)$$

Fig. 11. Dynamic Algorithm

generated when task i completes as late as possible (i.e. at its deadline) and the subsequent invocation is subject to the maximum interference from higher priority tasks. Maximum interference occurs when all higher priority tasks are released at the above deadline. This is effectively a critical instant for task i. Thus is equivalent to the level i idle time in the interval [O , Ti) where time 0 is a critical instant. The algorithm in Figure 10 may be used to calculate the value of Sadd off-line. We may implement a simple slack stealing algorithm by incrementing the level i slack available by Sadd at each completion of task i, whilst again using the equations in Figure 11 (a and b) to keep track of the slack at other times. The approximate slack stealing algorithm comprises this simple algorithm augmented by periodically evaluating the exact slack available at all priority levels. Varying the period at which this is done enables the overhead of the dynamic slack stealer to be traded off against a decrease in the responsiveness of soft tasks. A very short period minimizes the response times of soft tasks at the expense of a large overhead (c.f. the optimal dynamic algorithm). Whilst a very long period minimizes the overhead, but increases the response times. In the next section, we examine this performance trade-off.

## B. Results

For evaluation purposes, sets of hard deadline periodic tasks were used, enabling comparisons to be made between the approximate algorithm described above and the optimal Slack Stealer algorithm. The results are averaged over 10 task sets, each comprising 10 hard deadline periodic tasks. The periods of the hard deadline tasks were chosen at random in the range 2 to 1000 ticks. Deadlines were randomly chosen, but constrained to be less than or equal to the period. Finally computation times were adjusted randomly until the total processor utilization of the hard deadline task set was approximately 50%. Each task set was then subject to a sufficient and necessary schedulability test. Unschedulable task sets were discarded.

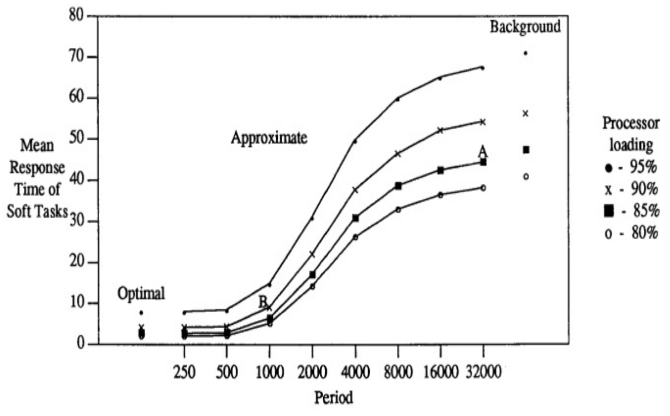


Fig. 12. Approximate Slack Stealing Result Graph

The soft task load was simulated by a FIFO queue of tasks, each requiring 1 tick of processing time. The arrival times of the soft tasks followed a uniform distribution over the test duration (lo0000 ticks). The number of soft tasks was varied to produce a simulated total processor loading of 80%, 85%, 90% and 95%. For each utilization level, we recorded the response time of each soft task scheduled by background, optimal and approximate slack stealing algorithms. In the case of the approximate algorithm, the simulation was repeated using periods of 250,500, 1000,2000,4000,8000, 16000 and 32000 ticks between calculations of the exact slack available. For comparison purposes, the mean response times of soft tasks scheduled by the optimal algorithm are plotted on the left, whilst corresponding values for background processing are plotted on the right (Figure 12). The graph shows that for all ”loadings” the mean response time of soft tasks, scheduled by the approximate algorithm, was very close to the optimal provided the period of the approximate algorithm was less than the mean task period (500 ticks). Increasing the period of the approximate algorithm resulted in decreased responsiveness, until with a large period (32000), it was close to that of background.

## V. Conclusion

The apparent versatility and, almost paradoxically, the strict nature of its definition, allow for a useful and predictable tool for Real-Time embedded systems. Its performance and its load on any system can be adjusted flexibly, allowing for a greater degree in practicality. It is safe to say that Stack Stealing will stand the test of time in the current generation of computing.

## VI. Statement of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at HSHL or any other educational institution, except where due acknowledgment is made in the thesis. Any contribution made to the research by others, with whom I have worked at HSHL or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this research paper is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

## References

[1] C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications* , 3rd ed. New York: Springer, 2011
Figures 1,3-7 are from this material.

[2] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami and Allyson Giannikouris, *A practical introduction to real-time systems for undergraduate engineering*, 2014
Figure 2 is from this material

[3] Jose Manuel Urriza, *A Fast Slack Stealing method for embedded Real-Time Systems*, 2005
Figures 8 and 9 are from this material

[4] R.I.Davis, K.W.Tindel1, ABurns , *Scheduling Slack Time in Fixed Priority Pre-emptive Systems*, 1993
Figures 10-12 are from this material

[5] R. Mall, *Real-Time Systems: Theory and Practice*, 2009
Other material used:

1. Slack Stealing - Real Time System: Chapter 5 (https://www.youtube.com/watch?v=9VA4eyT5w2M&ab_channel=SagunRajLage)
2. What is Slack Stealing in Deadline Driven System? (https://www.youtube.com/watch?v=5U_pLwPtVY4&ab_channel=InformationTechnologyHelpDeskOnline)