02223 Fundamental models for modern embedded systems E10

[Scheduling deterministic] *

Kim Rostgaard Christensen
s084283@student.dtu.dk

ABSTRACT

1. INTRODUCTION

2. THE WCET

The Worst Case Execution Time is the maximum time a given task can take up the cpu. It has a complimentary cousin called Best Case Execution Time.

2.1 Obtaining WCET

To be able to obtain the exact WCET it is essential that the all about the hardware architecture is known.

Modern processors tend to try and make things run faster by utilizing pipelines, instruction caches and branch prediction.

2.1.1 Branch prediction

When a branch in the program is reached (for example an if statement), the processor will try to predict which route the software will take. This saves cpu cycles, when guessed correctly, but costs extra cycles when an incorrect prediction is made, due to the fact that all the instructions that was lined up, now has to be replaced.

2.1.2 Pipelining

2.1.3 Instruction cache

2.1.4 Virtual memory

A few suppliers of real-time operating systems gives the programmer the option of using virtual memory, giving the benefit of being able to extend the application. QNX for example has this feature. The majority of suppliers does not implement virtual memory though, so in most cases this is not an issue.

*This report should also be available online at `www.retrospekt.dk/02223report`

3. SCHEDULABILTY TEST

In order to determine if a task set is schedulable with a scheduling algorithm a schedulability test is perfomed. Schedulability tests falls into three categories; sufficient, exact or necessary

Sufficient

A sufficient test means that a when the test passes, the task set i definitely schedulable. A fail provides no additional information.

Exact

An exact test guarantees schedulability on a pass. On fail it means the task set could fail to meet deadline, but not in all cases.

Necessary

If a task set fails a necessary schedulabilty test, then the task set will always fail.

4. VERY SIMPLE SIMULATOR

4.1 Rate monotonic scheduling

Rate monotonic scheduling (RMS) is a preemptive scheduling algorithm used when you have set of strictly periodic tasks with deadlines equal to their periods. A number of other assumptions are also required. All of them are listed here.

- Single processor
- Task deadlines are equal to their periods
- Periodic tasks
- All tasks are released as soon as they arrive
- All tasks start at the same time
- All tasks are independent
- No precedence or resource constraints
- No task can suspend itself
- All overheads in the kernel are assumed to be zero

4.2 Schedulability

To determine schedulability of a task set, a utilization test is applied. The formal version of the test is as follows:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \quad (1)$$

Or, in other words: The cpu usage represented on the left hand side by the sum of all the individual tasks utilization of the cpu in their period must be less that $n\left(2^{\frac{1}{n}} - 1\right)$, where n is the number of tasks.
This is a sufficient test, and task sets that fail this test are not necessarily unschedulable.

Futhermore, it holds that:

$$\lim_{y \to 0} U_{lub}(n) = \ln 2 \quad (2)$$

Proven by Liu and Leyland.

4.3 Simulation

For simulating a task set under rate monotonic scheduling, we first have to find a LCM of all the periods for the tasks in the set. This is also known as the hyperperiod. As C_i needs to be randomized, the simulation should run for a number multiples of the hyperperiod. The multiple of LCM will be denoted n

Priority assignment

The priorities is in RMS defined as the inverse of the period. In this implementation, the priority is relative to the hyperperiod and defined as $P_i = \frac{LCM}{T_i}$. This is done to avoid rounding errors and floating point arithmetic in simulation.

Job initialization

To initialize the jobs we insert them into a job queue with release time equal to $\tau_i.period \cdot (j-1)$ where τ_i is the task of the job, and j is j'th occurrence of the task. The job's time (remaining execution time) is also randomized in this step.

Simulation

The jobs are sorted on start time and priority, and the simulation runs by going through each cycle from 1 to n. In each cycle, a list of ready jobs are generated and the job with the highest priority is picked to execute. Ready jobs are jobs that have time > 0 and release $<$ current cycle.
Execution is done by a tick method call to the job that decreases time.
When the job terminates (time $= 0$), the response time is recorded and compared to the overall worst-case response time of the task, recording if it is worse that any previous.

4.4 Output

The simulator produces a textual output in the form:(name) (WCRT) newline. An example is shown below:

```
T1 1
T2 54
T3 2
T4 4
T5 6
T6 10
T7 28
```

It also returns the schedulability determined by asserting $D_i > WCRT_i$ for each task:

```
Simulator.RateMonotonicSimulation Reports Schedulable
```

In addition the simulator also produces a graphical output in the form of a timeline in svg format. An example is shown i figure 1

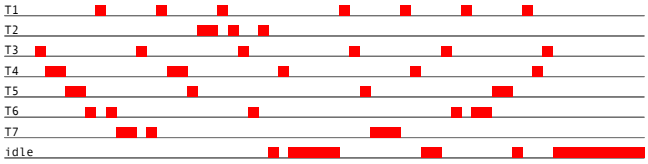

Figure 1: Example timeline

4.5 Implementation details

4.5.1 *Generation of the random numbers*

In order to perform a simulation having both a BCET and WCET, a randomization is needed. For this simulators purpose, either uniform or Gaussian distribution is used, depending on the configuration parameters. On an implementation note, Java's is used java.util.Random class is used for generating the random numbers. When using the uniform distribution the following should be taken into account:

> ... Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive) ... All n possible int values are produced with (approximately) equal probability[2]

As the upper value is exclusive, we need a value in the range $[BCET : WCET + 1[$ when using a uniform distribution.
Future improvement could also include to possibility to use a seed, to be able to recreate the random numbers generated.

4.5.2 *Traceability*

To be able to determine the situation of the first time overflow, a timeline is maintained, raising a global flag when the overflow occurs, and record the cycle.
To be able to trace what other task interrupted it, we can go back to the point where the overflown task last stopped (in time) and record the tasks between them.

4.5.3 *Response time guarantee*

When using random execution times are used for simulation, no guarantee can be provided. Although if you have a random distribution that is similar to the one in the actual application, then you are able give a better estimate.

When the execution time is always set to WCET, You end up with a very pessimistic estimate on the response time -

although guaranteed to be accurate. In praxis, a lot of CPU time will be wasted, especially if the execution time is much larger than the typical execution and only happens in very rare or perhaps even in theoretical cases.
Effectively you get the same figures as the response time analysis explained in section 5.

4.6 Final thoughts

Although one of the assumptions is that tasks must be independent, it is not guaranteed that task are statistically independent - meaning that a higher execution time on one task can be due to an external effect, that affects other tasks as well.
This eventually leads to a cascade of higher response times on all tasks that depend on, for example, some external input. Due to, that in the real world variables are not always independent.

5. RESPONSE-TIME ANALYSIS

Response-time analysis, in this case, involves the Deadline Monotonic feasibility test. It is based around the assumption that you know you critical instant (see section **??**) and from this point determines the worst-case response time of each task. Deadline monotonic is optimal for fixed task priority, and falls under the same assumptions as rate monotonic scheduling, with the one exception that deadlines can be less than the period.

5.1 Analysis

The analysis is based around the calculation of worst-case interference of a task. Interference of a task is defined as this:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \tag{3}$$

Meaning that the worst-case interference a task can experience is the response time of all previous tasks. Hence the total response time of the task becomes:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \tag{4}$$

I order to calculate this, we need to iterate through all the tasks with higher priority than the one we are currently examining, add up all the response times to the current task and record previously calculated response times.
Only the task with the lowest priority needs to be analysed in order to guarantee schedulability.

5.2 Implementation details

This response time analysis (deadline monotonic) is extended from an abstract analysis class, that has some properties general for all analysis's. This enables modularity and extensibility.
The implementation gives roughly the same textual output as the Very Simple Simulator, obviously without the svg timeline.

5.2.1 Comparison of RTA response times

The worst case response times are the same as the ones in the Very Simple Simulator when $T_i = D_i$. This is due to the fact, that the algorithms are very similar in effect. This hold specifically when $C_i = WCET_i$.

5.2.2 Comparison with Very Simple Simulator

In order to get the same numbers as with the VSS, you would need to run the simulation once with $C_i = WCET_i$, or infinite with random values, due to:

$$\lim_{n \to \infty} C_i = WCET_i \tag{5}$$

6. RESOURCES

In the tests studied so far, we have neglected to take into account the access to shared resources, and especially exclusive resources. If we were not able to provide secure and consistent access to resources, our deployment scenarios would very limited.
There exists a number of schedulability test that takes into account the usage of shared resources. These are based on the following protocols

- Priority inversion protocol
- Priority ceiling protocol
- Stack resource policy

The protocols are more or less extensions of each other from to to bottom. In this project, they will be implemented the same order, effectively giving them implementation priorities in the same order.

6.1 Priority inversion

6.2 Deadlock

7. CONCLUSION

APPENDIX

A. REFERENCES

[1] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *Scheduling in real-time systems*. John Wiley and Sons, octobre 2002.

[2] Java Platform Standard Ed. 6 Documentation. Documentation. Internet, 2010.