02223 Fundamental models for modern embedded systems E10

[Scheduling deterministic] [*]

Kim Rostgaard Christensen
s084283@student.dtu.dk

ABSTRACT

TODO - this section

1. INTRODUCTION

TODO - this

The application is built up of two approaches to schedule verification: simulation and analysis.

the concept of validation is to make a schedule that always meets all deadlines and utilizes maximum cpu

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.

3. SCHEDULABILITY

3.1 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

3.2 Schedulabilty test

To determine schedulability of a task set, a utilization test is applied. The formal version of the test is determined by Liu and Leyland and explained in [?] as follows:

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

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.

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

Futhermore, it holds that:

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

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. SIMULATION
In order to determine schedulabilty of a task set, a simulation can provide some additional information that a test cannot. For example; an execution timeline.

4.1 The model
The input model to the simulator is a set of tasks. A task has the attributes Name, BCET, WCET, Period, Deadline and Priority. The tasks are strictly independent and does not share any resources in this version.

4.2 Implementation
The simulator is implemented in a near-operating system fashion. It has a ready queue, a scheduler and a dispatcher. The dispatcher is built-in to the ready queue and is merely implemented as a method for getting the highest priority job, based on the current scheduling policy. The simulator simulates a scheduler by increasing time, one timeslice at the time, fetching the highest priority job from the set of ready tasks, and tries to execute them.

The simulator stores a history, in order to produce traceability and to produces a graphical output. This is in the form of a timeline in svg format. An example is shown i figure 1. This output is suboptimal, as development of it has not been emphasised very much during the project.

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 $P_i = \frac{LCM}{T_i}$, where P_i is the priority of the taks 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
Schedulability is determined by asserting $D_i \geq WCRT_i$ for each task:

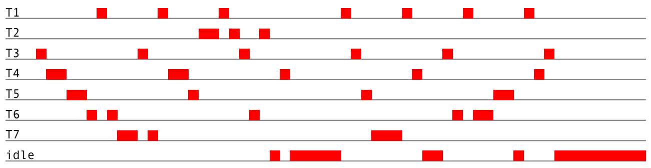

Figure 1: Example timeline

4.5 Implementation details
4.5.1 *GraphML*
The task model is stored in GraphML, which is an xml based modeling language.

4.5.2 *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[1]

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.

4.5.3 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.

4.5.4 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:

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

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:

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

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:

$$\lim_{n\to\infty} C_i = WCET_i \qquad (5)$$

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. Furthermore, resource access protocols are needed in order to prevent phenomenons such as priority inversion and deadlocks.

6.0.3 Priority inversion

6.0.4 Deadlock

In order to prevent this, the following protocols can be implemented.

- Priority inheritance protocol
- Priority ceiling protocol
- Stack resource policy

This report will focus on test and simulation of the priority inheritance protocol.

6.1 Schedulability test

The schedulability test of a task set using shared resources is actually quite simple. It merely extends the conventional response time analysis (4) with a blocking time (B_i).

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \qquad (6)$$

The blocking time is not so easily captured though. Using the priority inheritance protocol, every resource is provided with a ceiling. This ceiling is defined as the priority of the highest priority task that is using the resource.

The blocking factor B_i for a task can be calculated by the following steps,originally described in [?].

1. Find every critical section that can block the task τ_i
2. Find each semaphore that can block the task τ_i
3. Sum up the durations of the longest critical sections

6.2 Extending the simulator

To extend the simulator to include resource constraints, additional objects has to be introduced.

6.2.1 Resource mapping

We introduce two new entities; resource and usage. The resource entity is used to model an exclusive in an application and the usage entity is used to map them together with tasks.

6.2.2 Implementation

The two new entities resource and usage is modelled in GraphML as well as the tasks, and introduce two new files in order to provide the additional input information. One to model the resources, and one to map them. The directory structure is explained in section 7.

7. OVERVIEW AND USAGE

tasks must be stored in a directory along with its resources and usages. The input to the application is the directory containing the model. In order to extend the simulator to

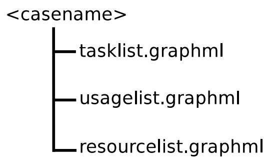

Figure 2: Directory structure

have a notion of resources, a few changes were needed.

With the new resource object, we needed to store references to the tasks using it in order to determine the priority of highest-priority task using the resource - the ceiling.

The resource is implemented as a binary semaphore. It provides a wait an signal primitive in the same way a normal operating system does. The resource also has a flag to indicate whether priority inheritance is enabled.

7.0.3 Testing

Testing is carried out by manually checking a known case. Three test models are provided: resource_case1, pathfinder_good and pathfinde_bad. The resource_case1 model is an example originating from [?], and models an application with usages of multiple resources. The two pathfinder studies model the application used in the Mars Pathfinder, that suffered from priority inversion and experienced watchdog resets as an effect.

The model suffixed by _good is the case where everything went well without priority inheritance protocol, and the model suffixed by _bad is the one that caused watchdog resets due to deadline misses.

Verification is simple when using the pathfinder models. The good one should schedule both with and without priority inversion enabled, and the bad should only schedule with priority inversion enabled. Furthermore, manual inspection of the execution time line has been done and verified to match the one from [?]. Some minor errors has been detected in the verification data, though.

7.0.4 Closing thoughts

The input model is a bit primitive in the sense that it assumes that a task aquires all resource at the start of its execution, and frees it only at the end end of its execution. A possible extension would be to include critical ranges instead of an absolute critical duration.

The model passes the pathfinder test, and still passes the tests that do not have resources usages, so we are fairly certain that nothing broke in the process of extending the simulator.

7.1 Simulation

In order provide additional verification the results of schedulability test, the project was extended to also include a simulation of the priority inheritance protocol.
The model should be reusable as-is from the implementation of the priority inheritance protocol, so the extension is only in the implementation.

7.1.1 Implementation

As the simulator uses a ready queue and dispatcher, the protocol can be implemented much in the same way it is in a real operating system. Making it more concrete.

The simulation cycle consists of some housekeeping, accounting and an instruction to execute the highest priority job in the ready queue. If the job blocks, a new one should be fetched from the ready queue. The following java while shows the implementation in the simulator:

```java
while (!currentJob.timeTick(cycle)) {
    currentJob =
        readyList.getHighestPriorityJob();
}
```

The example is a bit simplified in relation to the real implementation. But this extension allows us to move the execution of the job to the job itself. The job can now try to acquire a resource, if resource constraints are enabled, or just execute normally. This leaves the original functionality of the simulator intact.
If the job tries to acquire a resource, the resource is responsible for blocking the job and removing it from the ready queue, or giving the resource to the job.
Furthermore, it can do this assignment on the basis of policies. At the moment, priority inheritance is implemented. This is done by extending the normal wait and signal semaphore primitives with the following:

- **Wait:** When a job is blocked on a semaphore, it transmits its priority to the job currently holding the semaphore, leaving the holder with an elevated priority.
- **Signal:** When a job leaves the critical section, its priority is restored to its normal un-elevated priority.

7.1.2 Final thoughts

7.2 Discussion

Is priority inheritance protocol the new sliced bread? - In short, no. The protocol suffers from some drawbacks that makes it very unattractive - especially in hard real-time and safety-critical systems.

The primary problem of priority inheritance protocol is that it does not prevent deadlocking

8. USING THE TOOL

The application is built as a command line tool, that takes an analysis case as input and outputs the worst case response time for each task in the following form:
<name> <Worst-case response time> newline.
An example is shown below:

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

8.0.1 Documentation

The documentation is in the form of Javadoc and supplied with the code in the folder javadoc. To view the documentation, navigate to the folder and open the index.html file in a web browser.

9. CONCLUSION

TODO - this

10. REFERENCES

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