Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Felix Wermelinger

Implementation and Evaluation of Mixed Criticality Scheduling Approaches

Semester Thesis SA-2013-01
February 2013 to June 2013

Tutor: Prof. Dr. Lothar Thiele
Supervisors: Pengcheng Huang, Georgia Giannopoulou

Acknowledgments

I would like to thank Prof. Dr. Lothar Thiele and his research group at the Computer Engineering and Network Lab of the Swiss Federal Institute of Technology, for the environment that they created, within which I was able to write this thesis.
My supervisors Pengcheng Huang and Georgia Giannopoulou have been supportive in their role and have introduced me to scientific methods and ideas, as was needed for my work. Their constructive feedback has helped greatly in creating this thesis and I would like to thank them for that.

Abstract

This thesis studies the implementation of real-time mixed criticality scheduling algorithms. It explains which mechanisms a simulation environment has to support in order to allow mixed criticality scheduling, namely run-time monitoring and dynamically scheduling, preempting and canceling tasks. The implementation of the “Mixed Criticality Scheduling Framework” is documented, to show how such a system can be created. The thesis proceeds to test this framework, by implementing two mixed criticality scheduling algorithms, as well as explaining their underlying functionalities. The first algorithm “Earliest Deadline First with Virtual Deadlines” is implemented in two alternative ways, each having its own traits. The “Mixed Criticality Resource Flow” algorithm, having an entirely different idea behind it, is then used to test the framework and evaluate whether it is suitable for different scheduling approaches. For each of these implementations there were extensive simulation runs executed. The results support the claims that were made in the theoretical part.

Contents

1 Introduction **13**
- 1.1 Introduction . . . 13
- 1.2 Motivation . . . 13
- 1.3 Related Work . . . 14
- 1.4 Contributions . . . 14
- 1.5 Outline . . . 14

2 Mixed Criticality Scheduling **15**
- 2.1 Problem . . . 15
- 2.2 Example . . . 15
- 2.3 Formal description . . . 15
- 2.4 Scheduling algorithms . . . 16
  - 2.4.1 Earliest Deadline First with Virtual Deadlines . . . 16
  - 2.4.2 Mixed Criticality Resource Flow . . . 17
- 2.5 Summary . . . 19

3 Mixed Criticality Scheduling Framework **21**
- 3.1 Hierarchical Scheduling Framework . . . 21
- 3.2 Scope of the Implementation and Notation . . . 23
- 3.3 Extensions . . . 23
  - 3.3.1 Randomized Task Execution Times . . . 23
  - 3.3.2 Monitoring . . . 23
  - 3.3.3 Dynamic scheduling . . . 24
  - 3.3.4 Congestion control . . . 24
- 3.4 Implemented scheduling algorithms . . . 24
  - 3.4.1 Earliest Deadline First with Virtual Deadlines . . . 24
  - 3.4.2 Mixed Criticality Resource Flowing . . . 25
- 3.5 Summary . . . 26

4 Experimental Evaluation **27**
- 4.1 Overheads . . . 27
- 4.2 Measurement method . . . 27
- 4.3 Test set-up . . . 28
- 4.4 Earliest Deadline First with Virtual Deadlines Experiments . . . 28
  - 4.4.1 Task set . . . 28
  - 4.4.2 Results . . . 29
- 4.5 Mixed Criticality Resource Flow Experiments . . . 32
  - 4.5.1 Task set . . . 32
  - 4.5.2 Results . . . 33
- 4.6 Summary . . . 34

5 Conclusion and Future Work **35**
- 5.1 Conclusion . . . 35
- 5.2 Outlook . . . 35

A Mixed Criticality Scheduling Framework manual 37
A.1 Setup 37
A.2 Structure 38
A.3 Functions 38
A.4 Creating a scheduler 39

B Original Project Assignment 41

C Presentation Slides 47

List of Figures

2.1 An example of an EDF-VD schedule with two high critical tasks (tasks 1 and 2) and two low critical tasks (task 3 and 4). The two high critical tasks always get scheduled first, to guarantee meeting their deadlines and as soon as one of them overruns the low critical tasks get canceled. As soon as the CPU is idle the scheduler will schedule all tasks which arrive in the future 17
2.2 An example MCRF scheduling. The resource flows are depicted on the right side. The left side shows when each task is scheduled. The “server” illustrates the time that is actually allocated to task 1, even though some of it is then further distributed to tasks 4 and 5 because task 1’s execution is already guaranteed . . 19

3.1 Design of the HSF-framework (source: [2]) 21
3.2 Execution order example of a task instance of an EDF simulation using the hierarchical scheduling framework. 22
3.3 Example of a resource flowing graph using leaky bucket servers. 26

4.1 Scatter plot, where each ’o’ signifies, that for the respective probability/utilization bound, at least one experiment succeeded, while a ’x’ shows, that at least one failed 29
4.2 Average overheads of EDF-VD with 1 queue (left) and 2 queues (right), with utilization 70% 30
4.3 Average overheads of EDF-VD with 1 queue (left) and 2 queues (right), with probability of overrunning 20% 31
4.4 Aggregated graph of the relative overhead ($\frac{\text{absolute overhead}}{\text{total run time}}$) and the switching overhead of the two EDF-VD implementations. 32
4.5 Average relative overhead of MCRF as function of the probability of overrunning for each critical task. 33
4.6 Average relative overhead of MCRF as function of the Utilization Bound. 33

List of Tables

4.1 Table of Simulation Setup parameters 28

5.1 Comparison between the three implementations Mixed Criticality Resource Flow (MCRF), Earliest Deadline First with Virtual Deadlines using 2 queues (EDF-VD2) and Earliest Deadline First with Virtual Deadlines using 1 queue (EDF-VD1) . . . 35

Listings

2.1 Pseudo code of EDF-VD for n criticality Levels 16
2.2 Pseudo code of a MCRF Server 18
3.1 Example description of a task of criticality level 2 with 10% probability of running 40 ms and 90% of running only 20 ms 23
A.1 Content of the README file for HSF and MCSF 37

Chapter 1

Introduction

1.1 Introduction

This thesis is situated in the research field of real-time mixed criticality scheduling. Mixed criticality scheduling is a rather recent topic, which concerns itself with safety critical systems. Multiple tasks in such a system are crucial for the safety of the system itself and its surroundings, which often includes the safety of human beings. Tasks in such a system have varying execution times, unknown prior to execution. Normally, a system designer for a real-time system with varying execution times would before implementing the system, calculate a worst case execution time (WCET) and create a schedule which can schedule all tasks, such that every task meets its deadline, even if every task exhibits its WCET.

However, if the real-time system has safety-critical tasks, then it must be certified by one or more certification authorities (CA). CAs will grade different tasks in the system into criticality levels, where the highest criticality level is assigned to tasks, which are most important for safety, whereas lower critical tasks are less safety-critical. These CAs are very pessimistic about the execution times of tasks and assume higher WCETs than the WCETs that were assumed by the system designer. Designing a system the same as the mentioned system designer would design one, but with the assumptions of the CA would lead to a system, which is vastly over-dimensioned and uses most of its time only a small fraction of its resources.

The idea of mixed criticality scheduling is now to incorporate the criticality levels into the scheduling. We start from the system, which was designed by a system designer with his own assumptions. Should it actually happen that a task exceeds the WCET assumed by the system designer, which means that the original schedule is not admissible, then the scheduler may start a new schedule by canceling tasks of lower criticality level in order to ensure the safe execution of safety-critical tasks.

So, as long as the assumptions made by the system designer hold, all tasks will be scheduled and meet their deadlines. Simultaneously, we can also satisfy the requirements imposed by the CA, because there is a schedule which can be used in case the system designer assumptions fail. This schedule is guaranteed to be admissible as long as the assumptions made by the CA hold.

1.2 Motivation

The motivation to write this thesis was to create a framework which will faithfully enforce real time execution of tasks, which allows the implementation of mixed criticality schedulers. Then some algorithms, that only exist in theory so far, can be implemented to test the framework. This way one can test out the algorithms with real examples on real platforms and take measurements of the performance. Also one can test various implementation methods and compare their efficiency.

1.3 Related Work

The paper “Mixed-Criticality Scheduling of Sporadic Task Systems” [1] describes the functionality of the mixed criticality scheduling algorithm “Earliest Deadlines First with Virtual Deadlines”. An alternate algorithm “Mixed Criticality Resource Flow” is described in a paper of the same name [3]. These algorithms have been implemented in this thesis. The implementation of this thesis is based on an extension of the Hierarchical Scheduling Framework [2], which allows the simulation of real-time schedulers, with a focus on hierarchical scheduling structures.

1.4 Contributions

The main contributions can be summarized as follows:

- Extending the Hierarchical Scheduling Framework [2] to support Mixed Criticality Scheduling.
- Implementing the algorithms “Earliest Deadline First with Virtual Deadlines” and “Mixed Criticality Resource Flow”.
- Suggesting a performance metric (overhead) and comparing the implemented algorithms with respect to this metric. Showing that simulations using the implemented algorithms exhibit the properties the theory suggests.

1.5 Outline

In Chapter 2 the research field of mixed criticality will be explained in detail, with a further investigation of two mixed criticality scheduling algorithms. In Chapter 3 we will then look into the implementation of a mixed critical system simulation environment and explain the necessary mechanisms of such a system as well as the implementation of the scheduling algorithms. Chapter 4 presents a metric of performance and compares different implementations and algorithms based on this metric. In the end, Section 5.1 will summarize the thesis and Section 5.2 will suggest future uses of this thesis.

Chapter 2

Mixed Criticality Scheduling

2.1 Problem

In a simplified case of Mixed Criticality setting, there are only two criticality levels: high critical and low critical. High critical tasks have to be always guaranteed to meet their deadlines, whereas low critical tasks should be guaranteed, as long as they do not interfere with the high critical ones (i.e. low critical tasks do not cause a high critical task to miss its deadline). All tasks have a worst case execution time (WCET) assigned for the low critical case. This WCET is equivalent to the one, that a system designer would assume (see Section 1.1). As long as all tasks never overrun these WCETs, the scheduler has to guarantee the execution of all tasks.
As soon as one of these tasks overruns its assumed WCET, the scheduler will assume that all high critical tasks run at the WCET proposed by the CA (see Section 1.1). The low critical tasks can - but do not have to - be discarded. The scheduler has to guarantee the meeting of all deadlines of high critical tasks in respect to the WCET proposed by the CA.
This problem can be generalized for more than 2 criticality levels. Assuming different levels of criticality means, that a gradation is made in between these levels. So instead of just classifying tasks as low critical or critical, we can have a smooth gradation from low critical over medium-critical up to high critical tasks. With increasing criticality level, tasks have a stronger guarantee, where the strongest guarantee is unconditional. Each task has a WCET assigned for each criticality level lower or equal to the one he is assigned to. These WCET increase with increasing criticality level. A task may never overrun the WCET of its own criticality level.

2.2 Example

A typical example of a mixed criticality system is an airborne software system. For simplicity we mention only two tasks in this system: The autopilot and the “nearest airport” task, which is a task to calculate where the nearest airport is. A developer of this system (the system designer), who wants to sell the system is interested in ensuring its customers all parts of the system at all times. Because the autopilot is a safety-critical task, a Certification Authority (CA) has to certify the system. By dynamically allowing the system to suspend the “nearest airport” task should the assumptions of the developers be violated, it can be assured that the CA requirements are always fulfilled. This is because security is still ensured if the “nearest airport” task will be scheduled in most, but not all, cases, since the output of this task only changes in small ways. At the same time, the developer can assure that all tasks are always scheduled, given his own realistic assumptions hold.

2.3 Formal description

Formally, a mixed critical system is defined using a task set, where each task is defined by a 4-tuple of parameters: $J_i = (A_i, D_i, \chi_i, C_i)$:

- $A_i \in \mathbb{R}^+$ is the release time.

- $D_i \in \mathbb{R}^+$ is the deadline. Generally $D_i \geq A_i$, but we will assume $D_i = A_i$.
- $\chi_i \in \mathbb{N}^+$ denotes the criticality of the job, where larger values denote higher criticality.
- $C_i : \mathbb{N}^+ \rightarrow \mathbb{R}^L$ specifies the WCET of J_i for each criticality level, where L is the number of criticality levels.

2.4 Scheduling algorithms

This section describes the algorithms implemented in this thesis. For the exact implementation refer to section 3.4.

2.4.1 Earliest Deadline First with Virtual Deadlines

Earliest Deadline First with Virtual Deadlines [1] (EDF-VD) implements an EDF-schedule [5] for each criticality level. The EDF-schedule of level i will schedule all tasks above and including criticality level i. Tasks of lower criticality level than i will not be scheduled.
Suppose a higher critical task being scheduled due to its normal WCET. In the worst case we would be notified of its overrunning very shortly before its deadline. At this point there might not be enough time left to meet its deadline, because overrunning has increased WCET to the WCET of the next higher criticality level. This means that an overrunning notification must always appear such that there is enough time to finish the task using its WCET of the uppermost criticality level.
Thus, we introduce the virtual deadlines. Each task has a virtual deadline for each criticality level, which will be earlier than the normal deadline. This assures, that even when we get modified of an overrun just before the virtual deadline, there is enough time left to schedule the task with its higher criticality WCET before its actual deadline.
At the beginning the scheduler runs the EDF-schedule for the lowest criticality level, which will run all tasks. As soon as a task overruns, the scheduler switches to the EDF-schedule of one criticality higher and cancels all tasks of lower criticality levels.
A pseudo-code example of the scheduling algorithm, as executed by a scheduler can be seen in Listing 2.1. It shows the three main functions that can be called on the scheduler:
NewTask will insert the new task into the deadline-ordered queue and change the currently running task, if necessary.
TaskFinished will delete a finished task from the queue and update the currently running task. If the CPU is idle it will also reset the criticality level.
TaskOverrun registers an overrunning task and will cancel this task if it has a too low criticality level. Otherwise it will delete all tasks of the current criticality level and increase the criticality level.
In the implementation these functions have to be synchronized properly to allow concurrent calls to these functions. For simplicity this has been omitted in the pseudo-code.

Listing 2.1: Pseudo code of EDF-VD for n criticality Levels

```
NewTask(task){
  if(task.CriticalityLevel<currentLevel){
    return
  }
  queue.insert(task)
  if queue.first!=currentlyRunning{
    currentlyRunning.preempt()
    currentlyRunning=queue.first
  }
}

TaskFinished(task){
  queue.delete(task)
  currentlyRunning=queue.first
  if queue.first==null
```

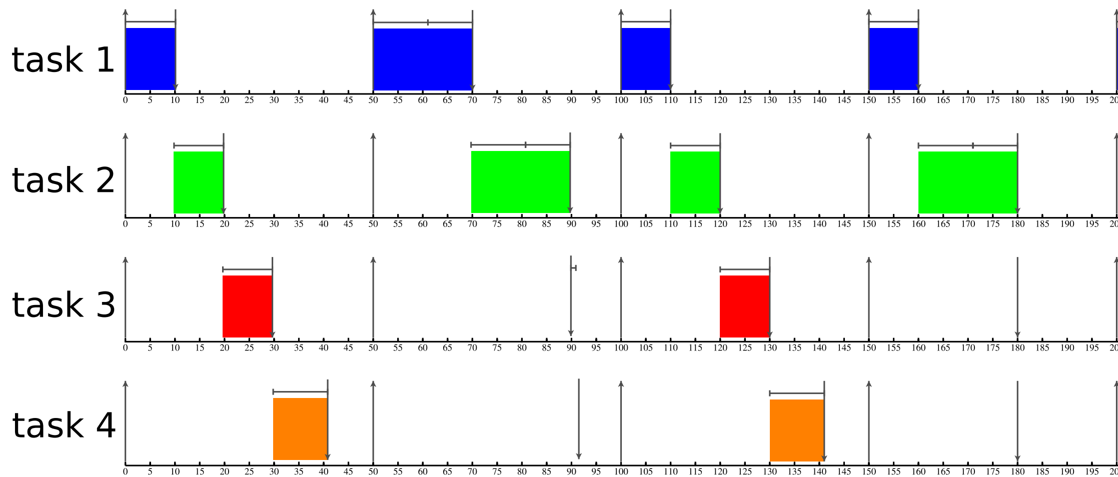


Figure 2.1: An example of an EDF-VD schedule with two high critical tasks (tasks 1 and 2) and two low critical tasks (task 3 and 4). The two high critical tasks always get scheduled first, to guarantee meeting their deadlines and as soon as one of them overruns the low critical tasks get canceled. As soon as the CPU is idle the scheduler will schedule all tasks which arrive in the future

```
    currentLevel=0
}

TaskOverrun(task){
  if(task.CriticalityLevel <=currentLevel){
    task.cancel()
    return
  }
  queue.CancelAndDeleteAllTasks(currentLevel)
  currentLevel+=1
}
```

2.4.2 Mixed Criticality Resource Flow

The Mixed Criticality Resource Flow (MCRF) algorithm has the advantage of a decreased number of canceled low critical tasks in a mixed criticality scheduling, while still remaining efficient in terms of scheduling overhead. This means, that a certain task overrunning will not cancel all tasks of lower criticality level. In contrast EDF-VD cancels all tasks of lower criticality level.

We assume each task gets assigned time slices of execution time, which guarantee that it will meet its deadline, even when its execution time is equal to its worst case execution time of its own level (which is the absolute worst case). Each task -called *master task*- has been assigned tasks of lower criticality level, called *slave tasks*, to which it will flow its remaining resources, i.e. the remaining time slices, after the *master task* is guaranteed. A resource flow (the amount of time slices, that are given to *slave tasks*) has to be guaranteed, if the *master task* does not overrun its own WCET of its own level. The overall system is called *schedulable* if the resources flowed to a task of level i guarantee that said task meets its deadline, given no task it gets resources from overruns the WCET of level i. This way it is guaranteed, that the *slave task* is scheduled, when the *master task* does not overrun its WCET of the criticality level of the *slave task*. When the *master task* overruns, it will not flow its resources, ensuring the meeting of its own deadline, and possibly making the meeting of deadlines for the *slave tasks* impossible.

The resources originate from the CPU, which distributes its time among various tasks. Each task has then unconditional resources (resource flows from the CPU) and/or conditional resources (resources flows from other tasks, which will distribute their slack time). Which task flows its

freed execution time to which other task is determined when designing the system and is a necessary information for the schedulability test. The only restriction for these resource flows is that the criticality level of the *master task* has to be strictly larger than the one of the *slave tasks*. The amount of resource flows is usually deliberately kept low in order to decrease scheduler complexity. This is because every resource flow will, if used, be another reallocation of resources, which causes overhead and also might trigger additional preemptions.

One can easily argue, that in general MCRF will cancel much less tasks than EDF-VD, because the overrunning of one task, which would cancel all tasks of lower criticality level in EDF-VD will only affect the *slave tasks* of the overrunning task. Not to mention the possibility, that each *slave task* has a chance to get enough resources from other tasks to still meet its deadline. However, one can also argue, that the scheduling overhead is most likely increased, due to the large number of preemptions. Also the schedulability analysis and designing of the system is quite sophisticated.

A pseudo-code example can be found in Listing 2.2. This code utilizes a server, which will be scheduled in place of its own *master task*. The server is responsible for scheduling its *master task* and all the *slave tasks* (which in turn might be replaced by servers responsible for said tasks).
The main function is *schedule*, which uses a round robin schedule among the *slave tasks*, but gives priority to an overrunning *master task*. Scheduling is performed by activating and deactivating the other entities. In turn the server itself can be preempted by a call to the *deactivate* method which will halt the *schedule* function and can be reactivated using *activate*.
In the implementation the *activate* and *deactivate* functions have to be synchronized properly to allow concurrent calls to these functions. For simplicity this has been omitted in the pseudocode.
An example can be seen in Figure 2.2, which shows how a server for task 1 will redistribute its execution time among its master and slave tasks as described in the pseudo code.

Listing 2.2: Pseudo code of a MCRF Server

```
schedule(){
 while(true){
  MasterTask.activate()
  waitFor(MasterTaskTimeSlice)
  if(MasterTask.hasOverrun){
   wait(UntilMasterTaskEnds)
  }
  MasterTask.deactivate()
  for each slave{
   slave.activate()
   waitFor(slave.TimeSlice)
   slave.deactivate()
  }
 }
}

deactivate(){
 schedule().Halt()
 currentlyRunningTask.deactivate()
}


activate(){
 schedule().Resume()
 currentlyRunningTask.activate()
}
```

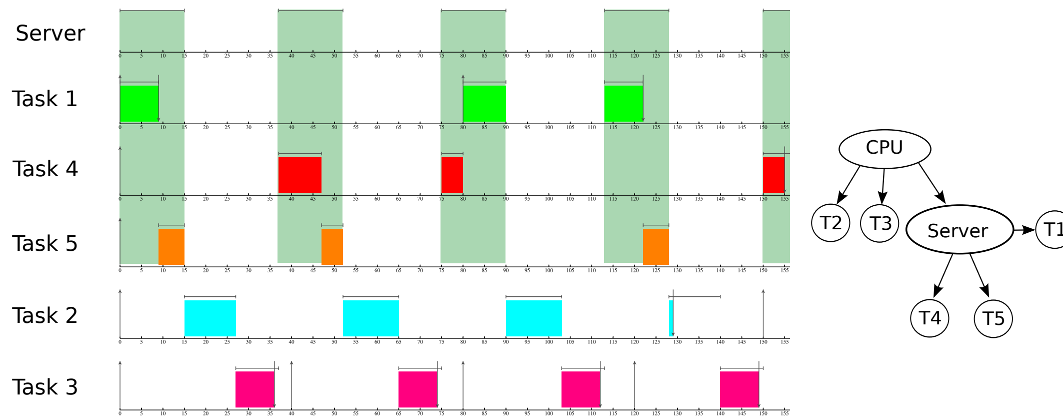


Figure 2.2: An example MCRF scheduling. The resource flows are depicted on the right side. The left side shows when each task is scheduled. The “server” illustrates the time that is actually allocated to task 1, even though some of it is then further distributed to tasks 4 and 5 because task 1’s execution is already guaranteed

2.5 Summary

This section explained the fundamentals of mixed criticality scheduling, including the formal problem description. It showed what the general idea of mixed criticality scheduling is and demonstrated the execution of said idea with two example algorithms.

Chapter 3

Mixed Criticality Scheduling Framework

3.1 Hierarchical Scheduling Framework

The software design is an extension of the Hierarchical Scheduling Framework (HSF) [2]. This framework was designed to allow hierarchical scheduling simulation and implemented classical scheduling algorithms such as “Earliest Deadline First”, “Time Division Multiple Access” and “Fixed Priority”.

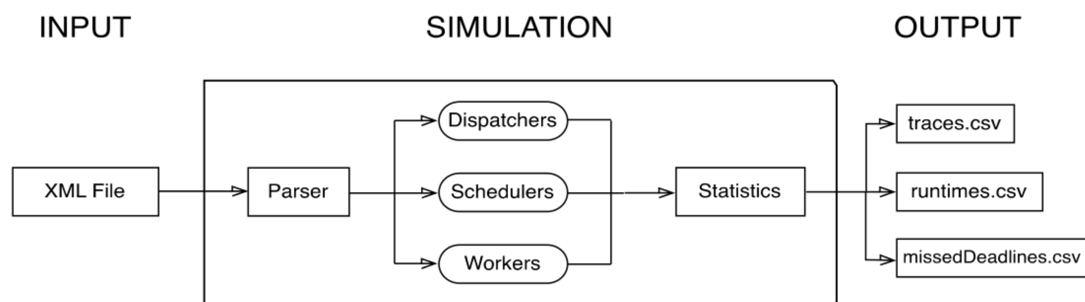


Figure 3.1: Design of the HSF-framework (source: [2])

Figure 3.1 shows the design of the HSF-framework. We briefly explain the components:

XML File This file holds the information of the simulation that will be carried out. This is information about the scheduling algorithm used and its parameters. The task set that will be handled is also defined in this file.

Parser The Parser will extract the information from the XML File, create and initialize all the components as described in the xml file.

Dispatcher For each task there exists a dispatcher, which will dispatch each instance of this task. Usually interesting are periodic dispatchers which will dispatch a new instance after a fixed time interval. But the framework also allows aperiodic dispatching.

Worker For each task there is exactly one worker responsible. As soon as the respective dispatcher has created an instance of said task, the worker will request execution time from the scheduler and start executing whenever the worker is granted any. The worker will also inform the scheduler when an instance has finished or has to be aborted because it overran its deadline.

Scheduler The scheduler will handle requests of workers who would like to execute their task instances. It will activate and deactivate these tasks according to the chosen scheduling policy. Schedulers may also be used hierarchically, where one scheduler is a subordinate of another and also has to request execution time, the same as a worker.

Statistics The statistics entity will collect data about the simulation to calculate different metrics after the simulation.

It is important to realize that this framework is designed as an application in the user space of a Linux system. This means, that there is an underlying Linux kernel with a priority based scheduler. Even though one should try to have no other tasks running when executing the simulation, there are precautions to minimize the effect of other tasks.

All the components of HSF are realized as threads. These threads have so called “real-time priorities” assigned, which makes them preempt any lower priority threads in the system. This way it is ensured, that there are no other tasks blocking the execution of our simulation, with the exception of other real-time tasks. The idea is that while being active, threads have a priority specific to their role, while being inactive they have the *inactive-priority*. If a task has *inactive-priority*, it will always be preempted by the idle thread, as it never sleeps and has a bigger priority, the *idle-priority*. However the idle thread is only scheduled, when no other thread is active, as they would then have a higher priority.

Figure 3.2 shows how the threads interact normally with each other. In the figure one can see, how a dispatcher is triggered at a certain point in time, to dispatch a new task instance, register it with the worker and request execution privilege for the worker. As soon as the scheduler grants these privileges, the worker will start the execution. When the task instance execution is finished, the worker will register the event with the scheduler.

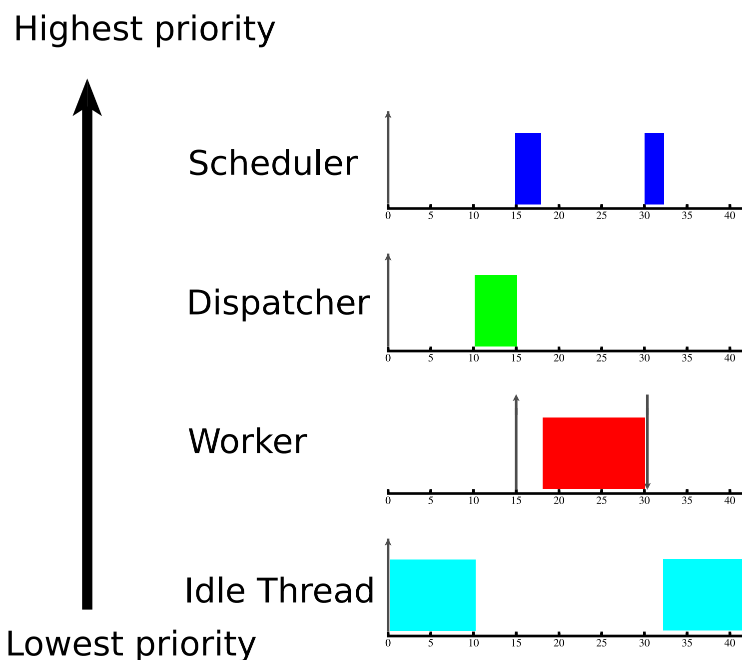


Figure 3.2: Execution order example of a task instance of an EDF simulation using the hierarchical scheduling framework.

3.2 Scope of the Implementation and Notation

Generally a task in mixed criticality scheduling has a separate worst case execution time (WCET) for each criticality level, which increases with increased criticality level. For the sake of simplicity, we will only consider systems with two criticality levels and thus each task has only up to two WCETs. We call the WCET of the low criticality level "Typical Case Execution Time" (TCET) and the high criticality level is still called "Worst Case Execution Time" (WCET).
The criticality level is represented by a value of "2" for high critical tasks and "1" for low critical tasks.

3.3 Extensions

As this project does not just want to implement a classic deadline or priority-based scheduler, some extensions to the existing HSF framework were required. This chapter will discuss why each extension is needed and explains the implementation.

3.3.1 Randomized Task Execution Times

Thus far the framework only allowed static task sets (i.e. one task would always run for the same amount of time). Mixed criticality scheduling is however only of interest if we have unknown task execution times.
The actual execution time will then be calculated as a function of the two parameters WCET and TCET and the third parameter *Distribution*, which may be a uniform (between 0 and WCET), binary (either WCET or TCET), normal (normally distributed) or fixed distribution (always WCET). For our simulation, the actual execution time is assigned whenever a new task instance has started and remains private knowledge of the task. The task will then run as usual for this execution time unless preempted. For the binary distribution, there exists an additional parameter "Probability". The actual execution time will be equal to WCET with this parametrized probability and otherwise it will be equal to TCET.

Listing 3.1: Example description of a task of criticality level 2 with 10% probability of running 40 ms and 90% of running only 20 ms

```
<runnable type="worker" periodicity="periodic" task="busy_wait"
monitoring="true">
    <period value="50"  units="ms" />
    <wcet   value="40" units="ms" />
    <tcet   value="20" units="ms" />
    <distribution value="binary" Probability="10" />
    <criticality_level   value="2" />
    <relative_deadline value="50" units="ms" />
    <virtual_deadline value="25" units="ms" />
</runnable>
```

3.3.2 Monitoring

Due to the randomized task execution times, one does not know when a task is going to finish prior to its finishing. Thus there has to be a monitoring mechanism which will observe the execution and inform the scheduler in case the task has overrun its typical case execution time.
In order to do this, each worker, which has in its xml-declaration *monitoring*="true" set will create an *OverrunChecker*-thread in parallel to its own worker-thread. The worker-thread will execute the task, which will only locally know what the actual execution time is going to be. The *OverrunChecker*-thread will run with higher priority than the task itself (with respect to the Linux-scheduler) and check periodically whether the task has overrun its TCET. As soon as this happens the *OverrunChecker*-thread will call the *OverrunJob*-function on the scheduler, to inform it about the overrun.

It should be noted, that monitoring is only supported for high critical tasks. Low critical tasks would be aborted by the scheduler anyway in the event of an overrun, so we emulate this behavior by defining that low critical tasks may never overrun their TCET in the first place.

3.3.3 Dynamic scheduling

When tasks of a high criticality level overrun, then the scheduler will need to cancel tasks of a lower criticality level. This means, that the worker has to be able to cancel its task at any time and free its resources to be ready for a new instance of the same task.
The scheduler may call the blocking function *cancel* on a worker, which will force the scheduling of the corresponding worker thread and set a canceling-flag for this worker. The worker thread, which will be busy executing the actual task, checks this canceling flag periodically and it will abort execution, should the flag be set. The *OverrunChecker*-thread should do the same. The worker will then have to properly finish the task, freeing its resources and as usual, register the finished task with the scheduler. After this is done, *cancel* will stop forcing scheduling the worker thread and return.

3.3.4 Congestion control

As a scheduler might set the scheduling to a high critical case for a long time and deny execution of low critical tasks, the workers will accumulate old task instances, which are dispatched. These tasks would all have to be handled, when the scheduler decides to reschedule low critical tasks, which might cause a delay. Thus, a dispatcher will always *cancel* the currently running task on the worker, when dispatching a new one. This will ensure, that each worker has only one task waiting for execution, which limits the delay to the same one we have during execution of the low critical scheduling anyway.

3.4 Implemented scheduling algorithms

Using the tools provided by the HSF-framework and the extension, the actual scheduling algorithms could be implemented. While this section discusses the implementation, the algorithm itself is described in Section 2.4.

3.4.1 Earliest Deadline First with Virtual Deadlines

The implementation of EDF-VD in this work only supports two criticality levels. The tasks which are on these levels are from now on called high critical and low critical tasks respectively. EDF-VD is a scheduling algorithm, which has just two possible schedules. Either it is in low critical mode and all tasks will be scheduled, or it is in high critical mode and only high critical tasks will be scheduled. Switching from low critical to high critical mode happens if any high critical task overruns its WCET of the low critical level. Switching from high critical to low critical mode is a difficult topic in itself. In the original paper [1] it was assumed, that normally no task overruns and if it should happen nonetheless, scheduling would never switch back to low criticality mode. In the case of this project, the implementation was simply done by just switching back if the queue of tasks to be scheduled is empty. This implementation is not optimal. One can even construct specific examples, where this policy would switch back a lot later than an optimal policy could. Assume a system with a high critical task with a very large execution time and period in comparison to the other tasks. For such a task set it would be optimal to precalculate the utilization over the future and see if the additional utilization, that would be required for a low critical task is available.
The implementations of EDF-VD are adaptions of a normal Earliest Deadline First (EDF) algorithm, which existed in the original HSF, in order to keep the extensions consistent with the framework. This implementation is event driven and will mostly operate around a *DeadlineQueue*. This queue will maintain tasks sorted by their deadlines. For our purpose it has been modified and it now supports high critical and low critical mode. In the former mode it will work the same as EDF and order tasks according to their relative deadlines. In the latter mode

it will instead use virtual relative deadlines. These deadlines are calculated prior to execution as explained in Section 2.4.1 and are written down in the xml-file.
The mode can not be switched on an existing queue, but has to be declared initially, when creating the queue. So instead of switching the mode one has to delete the old one and create a new one using the new mode. By using this simplified implementation method, there is barely any additional overhead, since the effor for rereading the deadlineas and reordering is necessary anyway.
Due to the complex nature of the problem there have been two different versions of EDF-VD implemented. One will construct a new queue when the criticality level is switched, whereas the other one will precalculate a second queue for the other criticality level. None of these two surpasses the other in every respect. Each is better suited for different scenarios, as explained below (for further comparison refer to Section 5.1).

EDF-VD1

This first implementation utilizes one *DeadlineQueue*. In low critical mode it will insert all tasks into the queue and schedule tasks like in standard EDF. Should a high critical task overrun and call the *OverrunJob*-function, it will switch to high critical mode. It then has to create a new *DeadlineQueue*, which runs in high critical mode and insert all tasks from the other *DeadlineQueue*. The new *DeadlineQueue* will have to be re-sorted, as the tasks are ordered by their virtual deadlines in low critical mode, whereas they are ordered by their deadlines in high critical mode.When started in *high critical mode*, a *DeadlineQueue* will automatically read the deadline of low critical tasks as infinitely large. Thus after all the high critical tasks have finished execution, the low critical tasks will also be scheduled and canceled using the *cancel* function. It is unwise to cancel tasks right when the mode is switched to high critical mode, as this will impose additional overhead at a moment, where there is already a lot of overhead, whereas with the method described, we will be canceling the tasks when the CPU would be idle anyway.
This implementation will have little average overhead, as in any of the two modes the overhead should be the same as a standard EDF overhead, because there is no additional calculation necessary when stable in one mode. When the mode is switched however, we expect a delay, as the scheduler has to create the new queue and insert all tasks into it. This delay might be quite large, because it would have to insert all tasks in a queue at once, whereas normally, we only insert one at a time, yielding increased overhead.

EDF-VD2

This implementation uses two *DeadlineQueues*. A low critical mode *DeadlineQueue*, which will have all tasks inserted(high critical tasks sorted by their virtual, low critical tasks by their actual deadlines) and a high critical mode *DeadlineQueue*, which will only have high critical tasks inserted, sorted by their actual deadlines. When switching to high critical mode, the scheduler can directly access the already existing high critical *DeadlineQueue*, where all the high critical tasks are already ordered correctly. As soon as the high critical Queue is emptied one can switch back to the low critical Queue, cancel and finish the execution of the remaining low critical tasks in this queue.
This implementation will have larger overhead during low critical mode, as there are always two queues maintained for inserting, scheduling or finishing high critical tasks. However, this implementation has a relatively small worst case overhead, as there is no additional overhead for switching the mode, because the alternate *DeadlineQueue* is already prepared.

3.4.2 Mixed Criticality Resource Flowing

The implementation is a modification of a classical Time Division Multiple Access server, which was already implemented. It uses a hierarchical approach, where multiple servers are linked to each other.
For each task, there is a dedicated *leaky-bucket-server*. This server will schedule the task it is dedicated to, the *master task*, and a set of tasks, we call the *slave tasks*, using a round robin schedule, with different time slice sizes for each task. The *master task* will always get the first time slice, if it is active. The time slices, which are given to *slave tasks* are equivalent

to the resource flows of the theoretical model explained in Section 2.4.2. The time slices are calculated in such a way, that under the assumptions of the system designer all tasks finish by their deadline, taking into account, that a task may get resource flows from multiple sources. Should the *master task* overrun, then the *leaky-bucket-server* will stop flowing resources to the *slave tasks* and only schedule the *master task* until it finishes.
Slave tasks can also be replaced by *leaky-bucket-servers* which will have their own master and slave tasks of lower criticality levels. Tasks of the highest criticality level may get their resources only from the CPU, which operates like a *leaky-bucket-server*, but is not dedicated to any *master task*, thus ensuring its resource flows unconditionally, as specified.
Tasks are not explicitly canceled by the scheduler in MCRF. However, they may get too few resources to meet their deadline. As a late execution is considered worthless, a worker missing the deadline will cancel his current task instance himself.
Due to the modularity of this algorithm, the implementation is theoretically capable of operating at an arbitrary count of criticality levels. However, as the framework does only support 2 levels, this could never be tested properly.

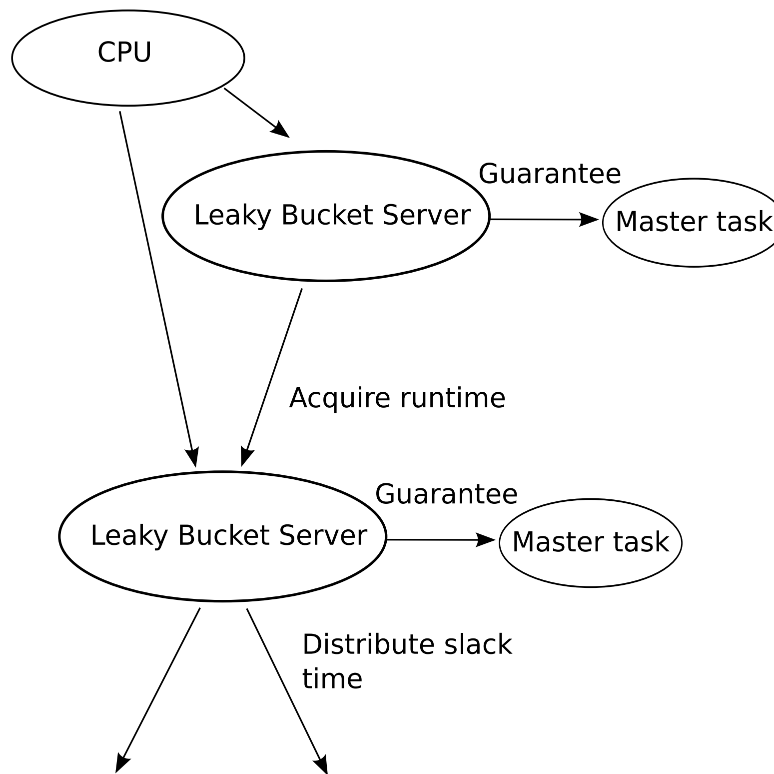


Figure 3.3: Example of a resource flowing graph using leaky bucket servers.

3.5 Summary

This section introduced the Hierarchical Scheduling Framework and the modifications made to it to create the Mixed Criticality Scheduling Framework (MCSF). Modifications include monitoring and dynamic scheduling of overrunning tasks. The use of the framework is then demonstrated by implementing two Mixed Criticality Schedulers: “Earliest Deadline First with Virtual Deadlines” and “Mixed Criticality Resource Flow”.

Chapter 4

Experimental Evaluation

4.1 Overheads

An interesting measure in real-time scheduling is always the overhead of a scheduler. In the given system however, we have several time measures which we have to distinguish in order to get meaningful results:

- t_{tasks}: The time workers spend actually executing a task
- t_{idle}: The time, no component of the system is active and the idle thread is running
- t_{Linux}: The time used by the underlying priority based Linux scheduler
- $t_{\text{framework}}$: The time used by the framework. This means all components of HSF, that are not concerned with scheduling, which includes the Dispatchers, Workers and OverrunCheckers.
- $t_{\text{scheduling}}$: The time used by the scheduler. EDF-VD uses time for handling queue-updates (insert/remove/resort), MCRF uses it for context switch when switching time slices
- $t_{\text{system tasks}}$: The time used by other tasks, that are not part of our Simulation. Should there be other real-time priority tasks in the system, then these would be able to preempt our execution.
- $t_{\text{total overhead}}$: The total overhead of the system

How these separate measures fit together can be seen in equation 4.1.

$$t_{\text{tasks}} + t_{\text{idle}} + \underbrace{t_{\text{Linux}} + t_{\text{framework}} + t_{\text{scheduling}}}_{t_{\text{total overhead}}} + t_{\text{system tasks}} = t_{\text{total run-time}} \tag{4.1}$$

We are primarily interested in $t_{\text{scheduling}}$, as we want to compare various different algorithms, however the total overhead shown in equation 4.2 might also be of interest.

$$t_{\text{total overhead}} = t_{\text{Linux}} + t_{\text{framework}} + t_{\text{scheduling}} \tag{4.2}$$

4.2 Measurement method

$t_{\text{scheduling}}$ is supposed to be a meaningful measure for the complexity of the scheduling algorithm used. It is defined as all time spent executing tasks, which are used for the algorithm. It is important to note, that $t_{\text{scheduling}}$ is not simply equivalent to the execution time of the scheduling thread. Some operations, which are part of the algorithms, such as inserting new jobs, are executed by other threads. Also there are operations, that the scheduling thread performs on Workers, which are not part of the algorithm itself (e.g. canceling task instances including saving the statistics). Thus, $t_{\text{scheduling}}$ had to be measured separately. This measurement utilizes stopwatches, which support *start* and *stop* functions, which will measure the time difference between the calls to

start and *stop* and add it up. To get deeper insight, there are several separate stopwatches used for different parts of the algorithm.
For calculating $t_{\text{total overhead}}$ we reformulate equation 4.1 into equation 4.3.

$$t_{\text{total overhead}} = t_{\text{total run-time}} - t_{\text{tasks}} - t_{\text{idle}} - t_{\text{system tasks}} \tag{4.3}$$

t_{tasks} is easily measurable, since this time is monitored to ensure that deadlines are met anyway. $t_{\text{total run-time}}$ is known and $t_{\text{system tasks}}$ can be neglected, as the system tasks will run on the other CPU cores. t_{idle} is equivalent to the time that the idle thread has run during simulation, which is the total run-time of the idle thread and the time the thread needed for initialization and freeing its resources. If we just assume t_{idle} is equal to the idle thread run-time, we will get a slightly too small value. When we use this to calculate $t_{\text{total overhead}}$ according to equation 4.3 this will be a pessimistic measure, since we chose the negative summand t_{idle} too small.

4.3 Test set-up

To have objective results, the generation of task sets was done according to the randomized algorithm presented in [4]. Task sets were generated with fixed utilization bounds in steps of 10% in the range of 10% up to 100%.
The utilization bound is defined as $U_{\text{bound}} = \max\left(U_{LO}^{LO} + U_{HI}^{LO}, U_{HI}^{HI}\right)$ where U_x^y represents the utilization of all tasks of criticality level x using exactly their WCET of level y and the levels HI, LO stand for “High criticality” and “Low criticality” respectively.
For the execution times of critical tasks, we have chosen a binary distribution, i.e. a task will either exhibit exactly its worst case execution time (WCET) or its typical case execution time (TCET). The probability with which it chooses the WCET can be configured. As in this setup the setting of this probability is equivalent to setting the probability of overrunning for each task execution, we will run each example multiple times for different probability settings. In the standard case, we will use 10% steps from 0% to 100%, but this can be configured freely.
For one scheduling algorithm each task set is run for each probability setting for a user configurable time, where just a few seconds are usually sufficient, as execution times are quite small (around 1ms).

	EDF-VD	MCRF
Number of task sets	1000	500
Utilization range of task sets	$10 \cdots 100\%$ in steps of 10%	$10 \cdots 100\%$ in steps of 10%
Probability tested for each task set	$0\% \cdots 100\%$ in steps of 10%	$0\% \cdots 100\%$ in steps of 10%
Execution time for 1 task	10μs $\cdots 1$ms	10ms $\cdots 1$ s
Time used for 1 simulation run	2 s	30 s

Table 4.1: Table of Simulation Setup parameters

4.4 Earliest Deadline First with Virtual Deadlines Experiments

4.4.1 Task set

We randomly generated 1000 task sets to be scheduled, 100 for each utilization bound value from 10% in 10% steps to 100% (For details, see Table 4.1) and each of these 1000 task sets was simulated 11 times with probabilities of overrunning of critical tasks from 0% in 10% steps to 100%. However, since the task sets were generated using tight bounds (i.e. the overhead is assumed to be zero), some of these task sets cannot always be properly executed by a real scheduler, which needs slack time for overhead. One recognizes a failure, when a critical task has missed its deadline, which is in theory impossible, as meeting deadlines of critical tasks is

guaranteed. However, the probability of overrunning has also an effect, because the advertised utilization bound might never be reached, when tasks simply never overrun. These results are summarized in Figure 4.1, which shows, that indeed only for large probabilities and Utilization, tests fail.

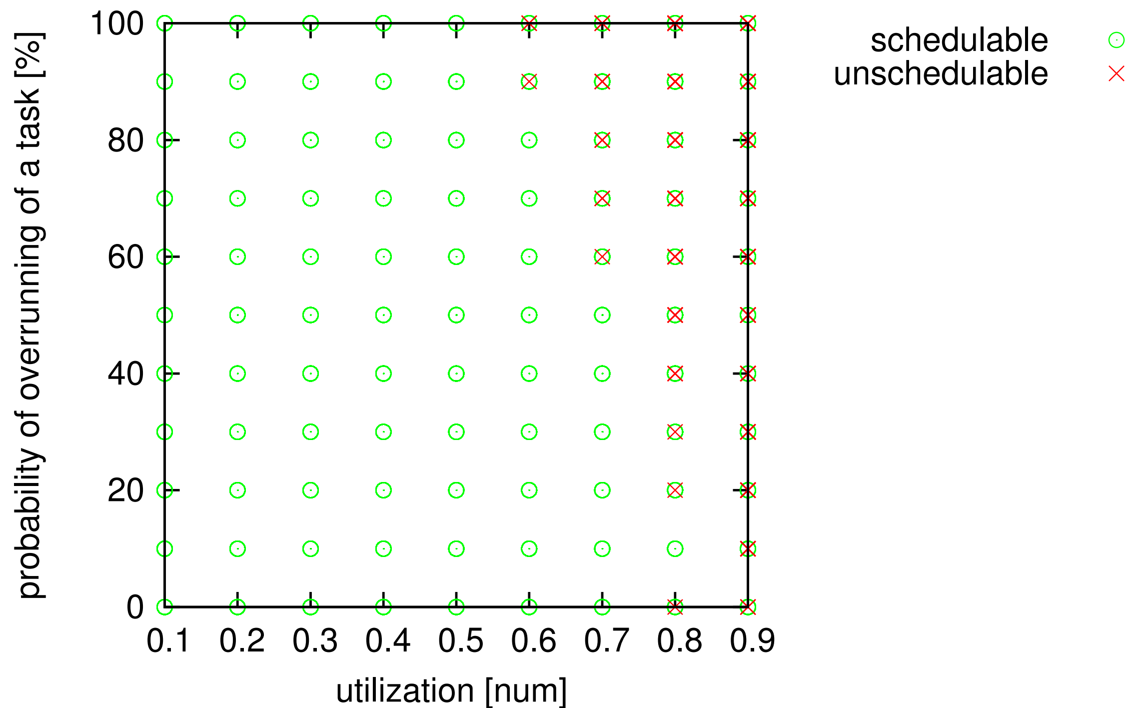
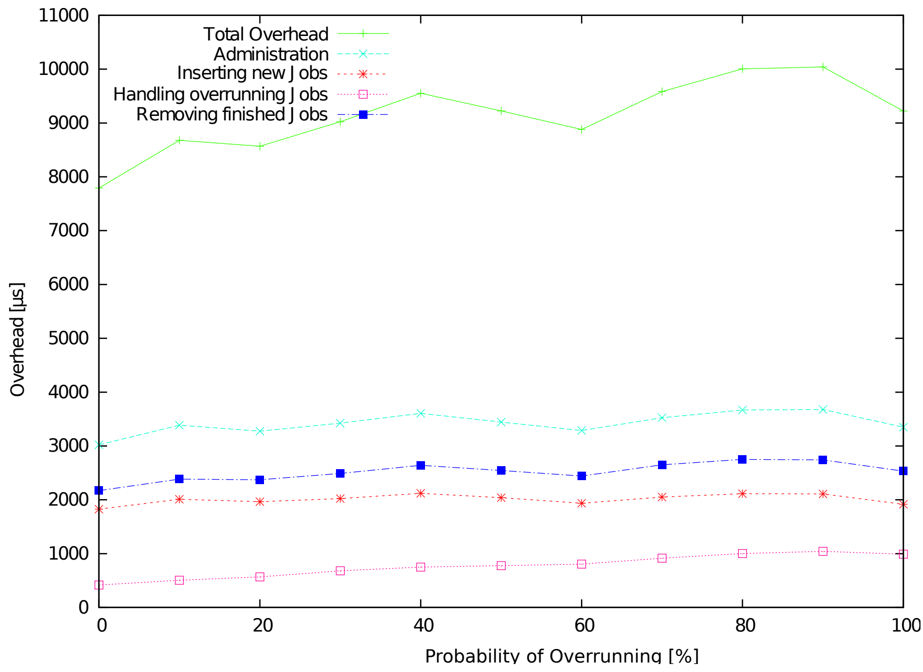


Figure 4.1: Scatter plot, where each 'o' signifies, that for the respective probability/utilization bound, at least one experiment succeeded, while a 'x' shows, that at least one failed

4.4.2 Results

The results from these test runs can be seen in Figures 4.2 and 4.3. The framework measures various separate overhead components of $t_{\text{scheduling}}$ during simulation. The shown data includes the time taken for handling new, finishing and overrunning jobs and also the additional administrative time taken for preempting currently running workers and rescheduling. The sum of all overheads is also included. Some small additional components are not shown, in order to keep the graphs simple.



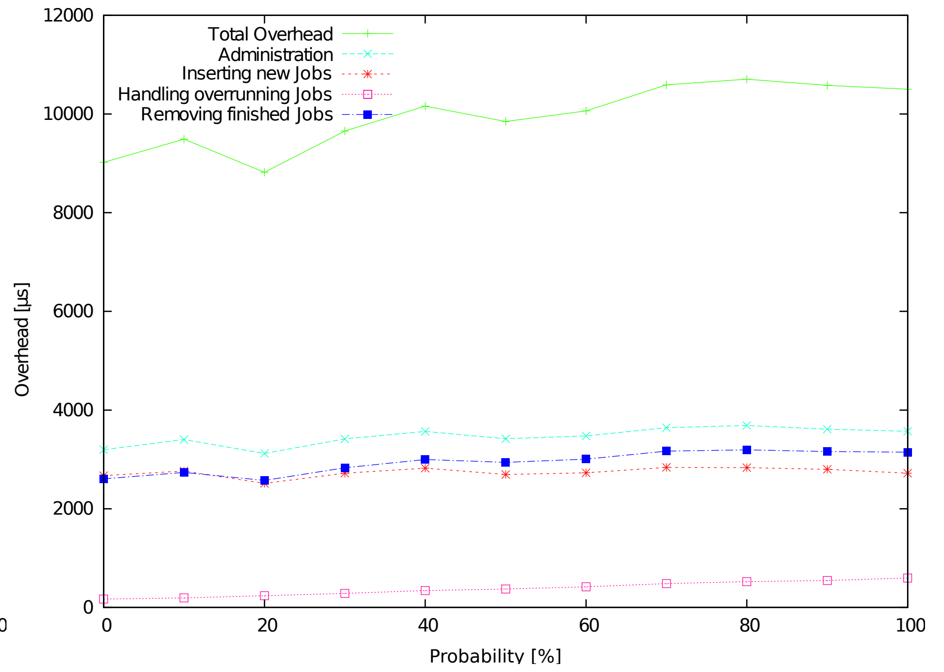


Figure 4.2: Average overheads of EDF-VD with 1 queue (left) and 2 queues (right), with utilization 70%

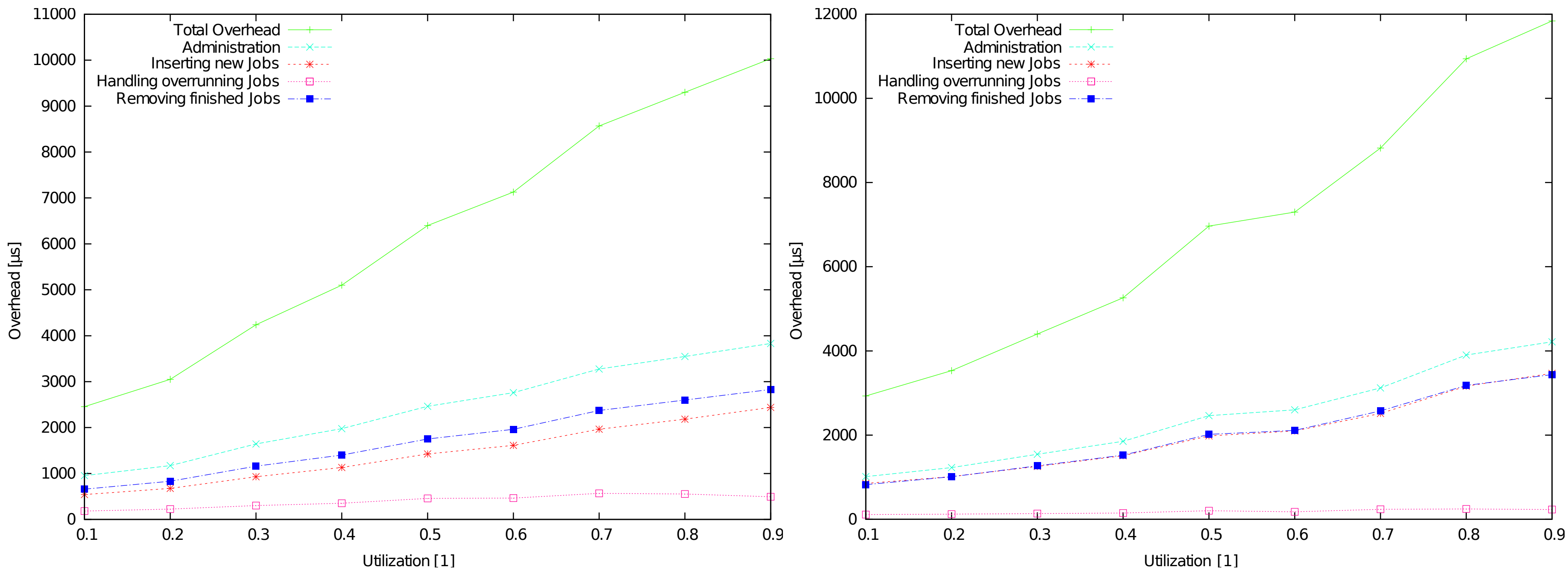


Figure 4.3: Average overheads of EDF-VD with 1 queue (left) and 2 queues (right), with probability of overrunning 20%

These results are consistent with the argumentation presented in Section 3.4 . I.e. the implementation with 2 queues (EDF-VD2) has generally a higher overhead of about 10% than the implementation using only 1 queue (EDF-VD1). We can explain the increase in overhead when increasing the utilization, because there are more tasks in the queue at once. This will increase the overhead for inserting task into the queue, because a task has to be inserted at the right space into the queue, such that the queue is sorted by the deadlines. The increase in overhead with probability is also because overrunning tasks will stay longer in the queues and make the operations on the queue more complicated.

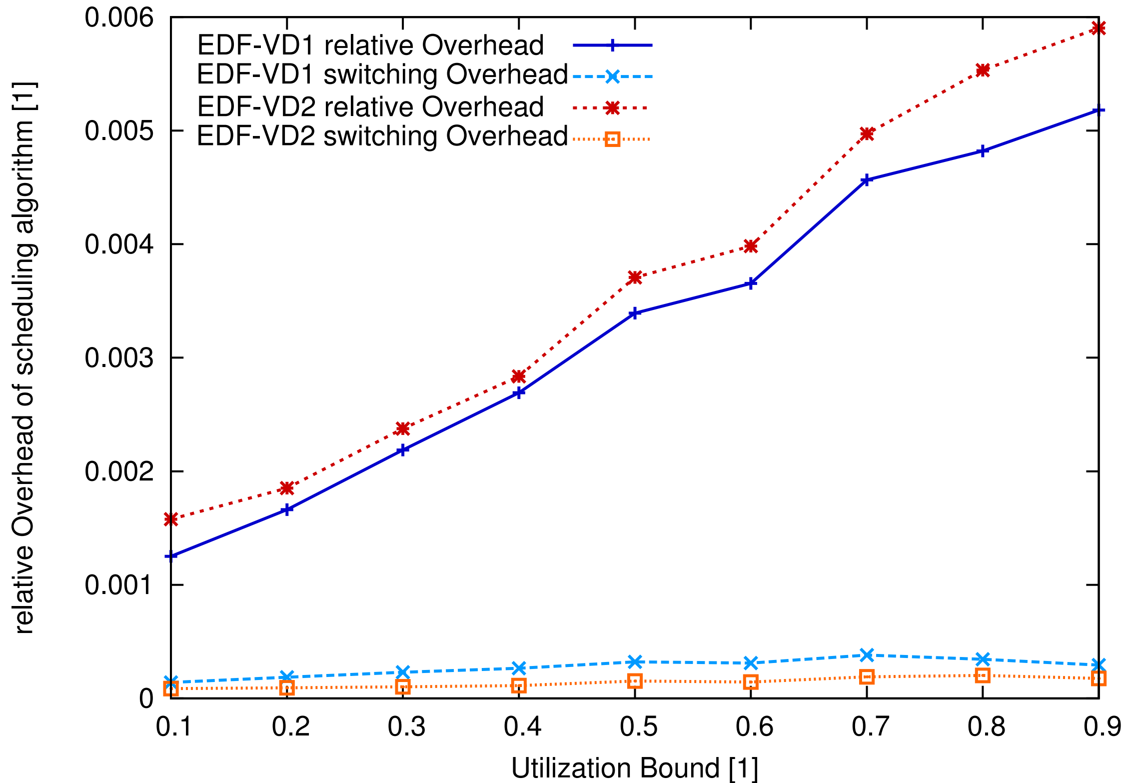


Figure 4.4: Aggregated graph of the relative overhead ($\frac{\text{absolute overhead}}{\text{total run time}}$) and the switching overhead of the two EDF-VD implementations.

The claim, that EDF-VD2 has a smaller overhead, when switching criticality level can however not be seen directly in the total overhead. One might think, that higher probability of overrun implies more switching, which would favor EDF-VD2 in the overall overhead. However, the algorithm switches to the critical level, as soon as only one task overruns and waits for the CPU being idle, before switching back. This means that the scheduler often spends most of its time in critical mode and switches rarely, even with low overrun probabilities. However, the figure shows also the average time used for handling overruns. Whenever an overrun occurs in low critical mode, the algorithm will have to switch to the critical mode. So the increased values of EDF-VD1 in this respect, show the claim that EDF-VD1 has a larger overhead when switching criticality levels. An aggregated view of the comparative results can be seen in Figure 4.4.

4.5 Mixed Criticality Resource Flow Experiments

4.5.1 Task set

There were 500 task sets to be scheduled, 50 for each utilization bound value from 10% in 10% steps to 100% (For details, see Table 4.1) and each of these 1000 task sets was simulated 11 times with probabilities of overrunning of critical tasks from 0% in 10% steps to 100%.
It should be noted, that the algorithm used for the MCRF samples uses very small time slices, relative to task execution times. The parameters had to be scaled up a lot (see Table 4.1) in

order to run the simuations. Even in this configuration timeslots are still in the range of 1μs $\cdots 1$ms, which yields a lot of preemptions and thus additional overhead.

4.5.2 Results

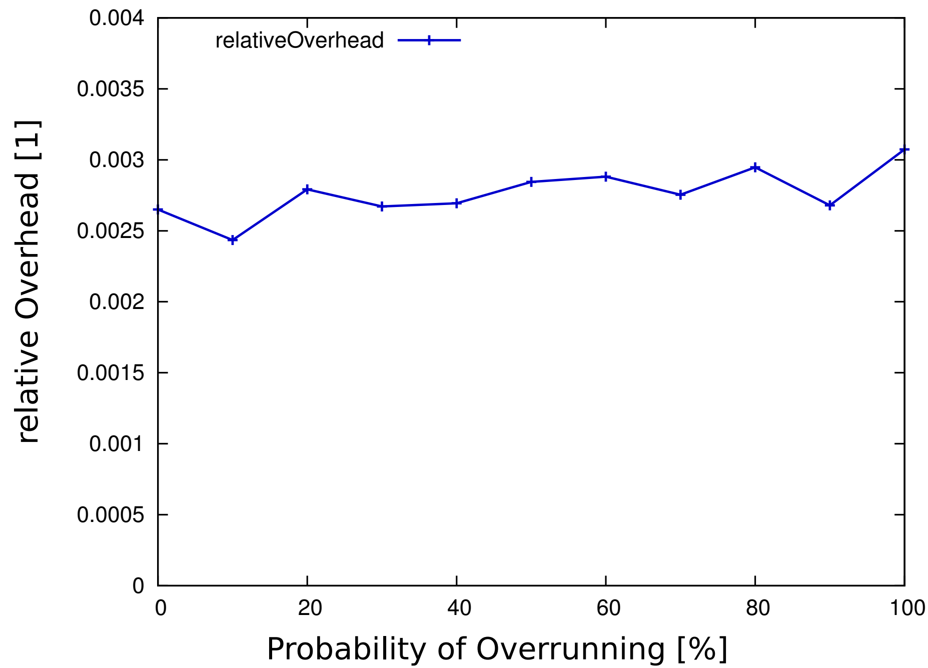


Figure 4.5: Average relative overhead of MCRF as function of the probability of overrunning for each critical task.

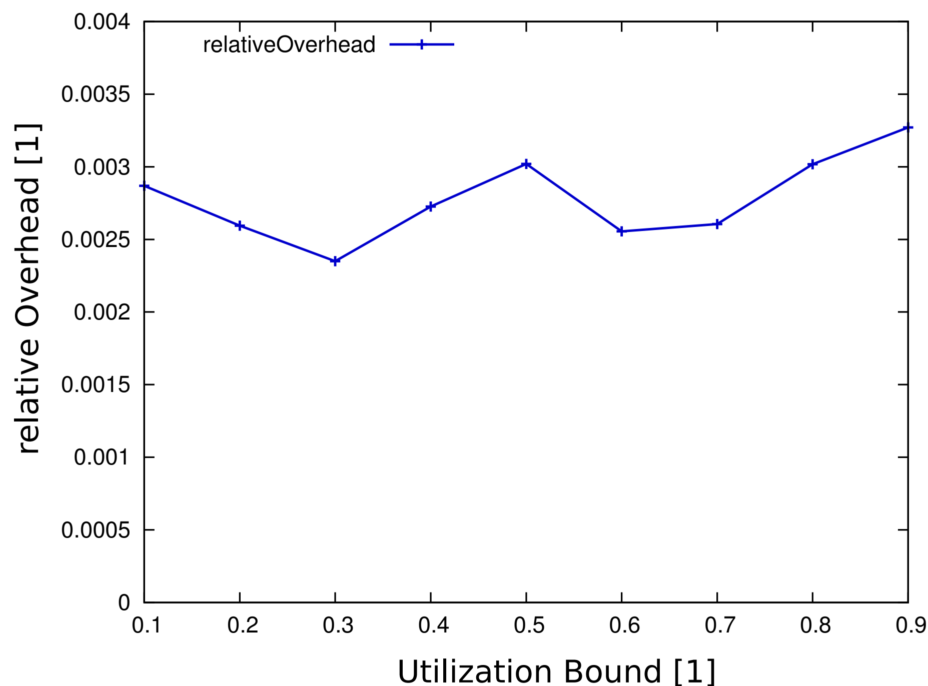


Figure 4.6: Average relative overhead of MCRF as function of the Utilization Bound.

The results about MCRF in Figures 4.5 and 4.6 show, that the overhead of MCRF is mostly

independent of overrunning probability and utilization, contrary to the others, where there is additional overhead for high utilizations and overrun probabilities. With high utilizations there is already very little slack time to begin with, so additional overhead in such cases are not desirable.
It should be noted, that one cannot directly compare the results of the two algorithms in this section. This is due to the large scale of the MCRF examples explained in Section 4.3. MCRF tasks are due to their small time slices much more often preempted and have thus a much larger overhead, when comparing it to a scaled up version of EDF-VD.

4.6 Summary

In this section the various overheads, that appear in the Mixed Criticality Scheduling Framework are explained and the ones, which are interesting metrics to analyze scheduling algorithms are highlighted. Both implemented algorithms were evaluated with respect to their overhead through extensive simulations. The simulation environment and the results of said simulations are shown. The simulation results are used to validate for the theory of Sections 2.1 and 3. There is also an explanation of why it is not fair to directly compare the results from the two algorithms with each other using these settings.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In conclusion, we have seen, that none of the implementations is in all respects superior to another. One should rather take the implementation, that fits best. Table 5.1 summarizes the characteristics of the implementations. Also we have seen, that the created framework is indeed capable of simulating different kinds of schedulers, using a modular and generic structure.

<table>
<tr><th>MCRF</th><th>EDF-VD2</th><th>EDF-VD1</th></tr>
<tr><td>Implementation is similar to a standard TDMA and also a Round-Robin Server. Such implementations can be used as template</td><td colspan="2">Implementation is a modification of standard EDF schedule</td></tr>
<tr><td>Same implementation for n criticality levels</td><td colspan="2">Implementation a lot more complicated for more than 2 criticality levels</td></tr>
<tr><td colspan="2">Large average overhead</td><td>Low average overhead</td></tr>
<tr><td>Overhead does not depend on probability of tasks overrunning</td><td colspan="2">Overhead increases with increased overrunning probability</td></tr>
<tr><td>-</td><td>No additional overhead when switching in between criticality levels</td><td>High overhead, when switching criticality level</td></tr>
<tr><td>Complex schedulability test</td><td colspan="2">Simple schedulability test, which increases in complexity for multiple criticality levels</td></tr>
<tr><td>Cancels only few tasks</td><td colspan="2">Cancels all tasks of lower criticality levels</td></tr>
</table>

Table 5.1: Comparison between the three implementations Mixed Criticality Resource Flow (MCRF), Earliest Deadline First with Virtual Deadlines using 2 queues (EDF-VD2) and Earliest Deadline First with Virtual Deadlines using 1 queue (EDF-VD1)

5.2 Outlook

The newly presented extensions to the Hierarchical Scheduling Framework (HSF) are ready to be used for other mixed criticality scheduling policies. Also one could even add more extensions, allowing more than 2 criticality levels, or additionally randomizing arrival times of tasks. The explanations within this thesis may also help, when implementing one of the algorithms in another system. The results may, then, help in choosing an appropriate algorithm. Another feature to

add would be multi-core support, as there are also mixed criticality schedulings proposed to handle multi-core environments.

Appendix A

Mixed Criticality Scheduling Framework manual

This section explains how the Mixed Criticality Scheduling Framework (MCSF) can be used.

A.1 Setup

Setup of the Mixed Criticality Scheduling Framework is the same as the setup for the Hierarchical Scheduling Framework [2]. Instructions can be taken from the README file found in the top HSF folder. The file is also shown in Listing A.1.

Listing A.1: Content of the README file for HSF and MCSF

```
REQUIREMENTS:

   HSF requires a *NIX kernel with standard libraries. To compile
    all sources, these packages are needed:

      g++
      make
      octave
      php
      libmgl-dev
      libX11-dev

  Mathgl library should be compiled, and libmgl.so.7.0.0 should be
  placed in /usr/local/lib (otherwise the MATHGL variable in the makefile
  should be changed to the appropriate location. For more information
  on Mathgl, please visit:

      http://mathgl.sourceforge.net/

CONFIGURATION:

1. If you HSF folder is not located in ~/git, then please change
   line 3 of hsf_paths.sh to the path of your HSF folder

2. In the terminal, type:

        source hsf_paths.sh

   This will set a new $HSF variable, and add it to your $PATH variable.
   You can also add it to your ~/.bashrc file, to have it load automatically
```

```
   You need root privileges to execute hsf. On some older systems, you might have
   to add the following line to your bash profile in order to inherit you PATH
   variable when using 'sudo':

         alias sudo='sudo env PATH=$PATH $@'

3. Then type:

         ./install.sh

4. Run HSF!

   You can now type in you terminal the following commands:

     sudo hsf [filename(.xml)]
     simulate [filename(.xml)]
     calculate [metric] [filename]
     show [metric] [filename]
     simfig [filename]
     publish [filename]

     [metric] can be one (or more) of the following:

       exe|exec      -> Execution Times
       resp          -> Response Times
       throughput    -> Throughput
       util          -> Utilization
       alloc         -> Resource allocation costs
       sys           -> System allocation costs
       worker        -> Worker costs
       missed        -> Missed deadlines
```

A.2 Structure

There are 4 important sections in the folder structure:

Folder name	Description
src	Source Files for the core simulation
bin	Binary Files and executable bash files for execution
scripts	Additional scripts and source files for things that are not used during normal simulation
examples	All .xml files, which describe task sets are saved in here

A.3 Functions

Here the most important functions provided by the framework will be shortly explained. Only the functionalities used for mixed criticality simulations are mentioned. For core hierarchical scheduling functions, refer to [2].

When not adding any functions to the system path, all these function are supposed to be called directly from the top directory.

Function call	Description
`getDuration.exe example.xml [newDuration]`	
	If newDuration is omitted, it extracts the specified time for the simulation duration from the xml file. Otherwise the xml file will be changed to newDuration.
`runProtected.sh exampleName [newDuration]`	
	Runs the core simulation using the exampleName.xml example from the examples directory. The newDuration option lets one set a specific new duration for the simulation, using the getDuration.exe script
`bin/setProbability Example.xml probability`	
	Sets the probability of overrunning of all critical tasks in Example.xml to probability (value ranging from 0 to 100)
`sweepProbability.sh ExampleName duration`	
	The file examples/ExampleName.xml will be simulated using runProtected once for each probability from 0% to 100% in 10% steps. duration has to be set.
`parseTaskSet.exe fileName SchedulerName`	
	Will parse the task set specified in fileName using the Scheduler SchedulerName and save all files under the name fileNameSchedulerNameNumber.xml
`parseAndSweep.sh fileName SchedulerName duration`	
	Will run 'parseTaskSet.exe fileName SchedulerName' and then run 'sweepProbability.sh exampleName duration' for each exampleName, which was created when parsing
`graph.exe inputFile.csv OutputFile.eps xAxis [yaxis...]`	
	Will save graph to OutputFile.eps constructed from the data in inputFile.csv (e.g. tasksEDF_VD1_Overhead.csv). Will plot all arguments in yaxis as functions of xaxis. xaxis and yaxis arguments must match exact header expressions in inputFile.csv. '–average' will only display average. '–fixed xAxis value' will only display rows of data, where argument xAxis is exactly equal value.

A.4 Creating a scheduler

This subsection will explain what needs to be done, should one want to implement a new Mixed Criticality Scheduler:

- Create a new class MyScheduler, which extends the abstract class “Scheduler”.
- Implement the functions defined in “Scheduler”. We will only mention the most important ones here:

Function	Functionality to implement
schedule	This is the main thread, it should perform all synchronized tasks
activate/deactivate	suspend the main thread by setting it to idle priority
newJob	Handle the arriving of a new job
finishedJob	Handle, that a job has finished its execution
overrunJob	Handle a job which has overrun its WCET of the low criticality level

Pay attention to synchronize these functions properly using semaphores.

- Extend the parser. Create a new entry in the Parser::parseScheduler function and add a function “parseMyScheduler”, where you create your own Scheduler. Your parser function will most likely the also contain other parseScheduler and parseWorker functions to parse the xml-children of your scheduler (e.g. the workers who will execute the tasks that you schedule).
- Write an example file, which utilizes your scheduler and execute it, with the commands explained in the previous section.

Appendix B

Original Project Assignment

Semester Thesis at the
Department of Information Technology and
Electrical Engineering

for

Felix Wermelinger

Safe Software for Safe Flights - Implementation and Evaluation of Mixed-Criticality Scheduling Approaches

Advisors: Pengcheng Huang
Georgia Giannopoulou

Professor: Prof. Dr. Lothar Thiele

Handout Date: 18.02.2013
Due Date: 25.05.2013

1 Project Description

Complex embedded systems typically involve functionalities of different importances (criticalities). As an example, the airplane software applications can be usually categorized as flight critical or mission critical. For flight critical applications, like the autopilot, failures (e.g. pilot commands not being transmitted in time) could result in an airplane crash, while for mission critical applications, like the radio communication or the passengers' video entertainment, the consequences of failures (e.g. loss of communication or wrongly decoded videos) are not severe. On the other hand, various unexpected situations may happen during the operation of an airplane, since neither the hardware nor the software we build for airplanes are perfect. How should the system react to such unexpected situations? And which properties should/can we guarantee in such dynamic and mixed-criticality environments? To answer those questions, smart online scheduling algorithms that can react to unexpected scenarios need to be developed.
Recently, in our group, we have developed a resource flowing scheme that can adaptively distribute resources to processes depending on the revealed situations when the system is running. It is essential to evaluate the runtime overhead of this scheduling technique in order to demonstrate its applicability to real-life applications. Hence, an implementation of the proposed scheduling technique needs to be conducted on a real platform. To achieve this, the Hierarchical Scheduling Framework (HSF) [4] is selected as a general scheduling framework and this project aims at integrating the new scheduler into this framework.

2 Project Goals

The goal of this semester thesis is to implement and evaluate the runtime behavior of our resource flowing scheme by implementing it on a real platform. The student needs to specify good metrics to quantify the performance and overheads of our resource flowing scheme, and compare it with other existing approaches. Specifically, some new features that need to be supported by HSF include:

- mechanism for runtime task monitoring;
- feedback control based on information monitored;
- a new scheduler which works on the basis of task monitoring and feedback control and specifically targets mixed-criticality scheduling.

Depending on the progress of this project, the goals of this project may further include:

- implement several other new schedulers (e.g. [5, 3]) in the HSF framework;
- propose metrics to quantify and compare the performances of different mixed-criticality schedulers.

3 Tasks

The project will be split up into several subtasks, as described below:

3.1 Familiarization with HSF

In the beginning of the project, the focus is on getting acquainted with the HSF scheduling framework, namely the mechanism that implements the hierarchical schedulers, the utilization of Posix threads [2] for this purpose, etc. A tutorial on the HSF framework is given in [4]. All the necessary resources for this part of the project will be made available as soon as the project starts. At the end of this project phase, it should be clear which parts of the HSF framework need to be adapted for implementing the new schedulers and how the runtime monitoring and feedback control can be supported by the HSF framework.

3.2 Implementation of Schedulers

In this second phase of the project, extensions to the HSF framework need to be first done to support task monitoring and feedback control. Based on the new features that will be developed, mixed-criticality schedulers will be integrated into the HSF framework. By actually running the implemented schedulers, quantitative measures on their performances need to be presented.

3.3 Thesis Report and Final Presentation

Finally, a thesis report is written that covers all aspects of the project. In addition, the final presentation has to be prepared.

4 Project Organization

4.1 Weekly Meeting

There will be a weekly meeting to discuss the project's progress based on a schedule defined at the beginning of the project. A revision of the working document should be provided at least 24 hours before the meeting.

4.2 Semester Thesis Report

Two hard-copies of the report are to be turned in. All copies remain the property of the Computer Engineering and Networks Laboratory. A copy of the developed software needs to be handed in on CD or DVD at the end of the project.

4.3 Initial and Final Presentation

In the first month of the project, the topic of the thesis will be presented in a short presentation during the group meeting of the Computer Engineering Lab. The duration of the talk is limited to three minutes. At the end of the project, the

outcome of the thesis will be presented in a 15 minutes task, again during the group meeting of the Computer Engineering Lab.

4.4 Work Environment

The work will be carried out in the framework of the European CERTAINTY [1] project, to which the Computer Engineering Lab is contributing in terms of scheduling techniques, performance analysis and multi-core mapping optimization. Concretely, this means that the results of this work can be used by the involved project partners if the project goals are met.

References

[1] Certification of Real Time Applications designed for mixed criticality (Certainty). http://www.certainty-project.eu/.

[2] POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/.

[3] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *Proceedings of the 19th European conference on Algorithms*, ESA'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] A. Gomez. Hierarchical Scheduling Framework - A Programmer's Manual. 2013.

[5] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

Zurich, February, 2013

Appendix C

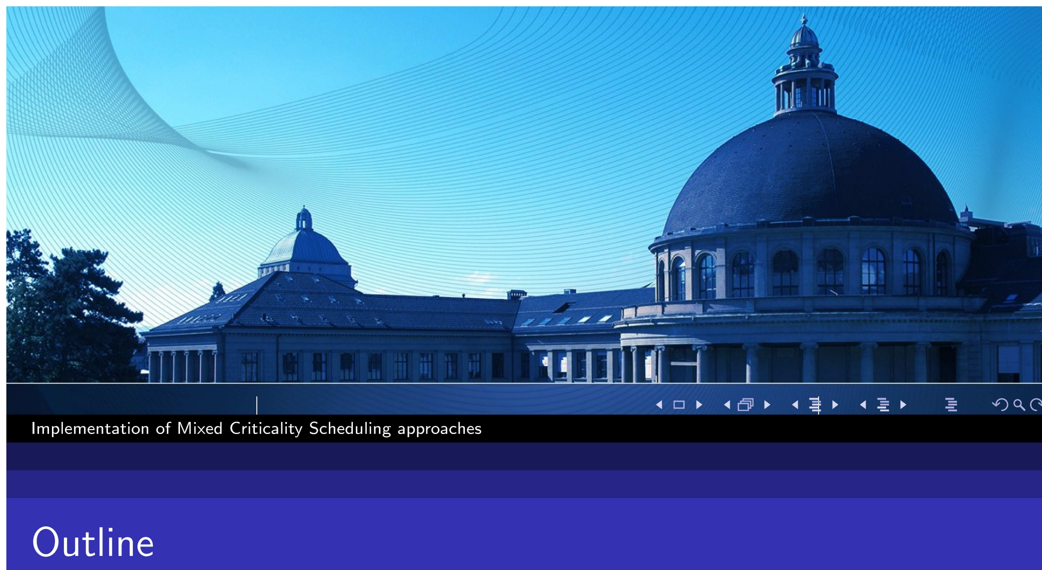
Presentation Slides

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK
Computer Engineering and
Networks Laboratory

Implementation and Evaluation of Mixed-Criticality Scheduling Approaches

Semester Thesis of Felix Wermelinger
Supervised by Pengcheng Huang & Georgia Giannopoulou



1. Introduction
2. Mixed Criticality Scheduling Framework
3. Earliest Deadline First with Virtual Deadlines
4. Mixed Criticality Resource Flow
5. Conclusion

Outline

1 Introduction
- Problem
- Motivation
- Contribution

2 Mixed Criticality Scheduling Framework

3 Earliest Deadline First with Virtual Deadlines

4 Mixed Criticality Resource Flow

5 Conclusion

Problem

- Designer for a hardware/software system for airplanes
  - Guarantee all functionalities to client
- System must be certified by Certification Authority
  - Assign criticality level to task
  - Pessimistic Performance Estimation
- Idea: dynamically schedule different task sets
  - Guarantee execution of critical tasks
  - Schedule low critical tasks iff higher ones are guaranteed

Motivation & Contribution

Motivation

- Few implementations of mixed criticality schedulers exist[1]
- Goal: real-time framework for implementing such schedulers

Contribution

- Implementation of the Mixed Criticality Scheduling Framework
- Implementation of 2 Mixed Criticality Scheduling Algorithms
  - Earliest Deadline First with Virtual Deadlines
  - Mixed Criticality Resource Flow
- Evaluation of implemented Algorithms
  - scheduling overhead

[1] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg, Multicore Operating-System Support for Mixed Criticality

Outline

1 Introduction

2 Mixed Criticality Scheduling Framework
- Hierarchical Scheduling Framework
- Mixed Criticality Scheduling Framework

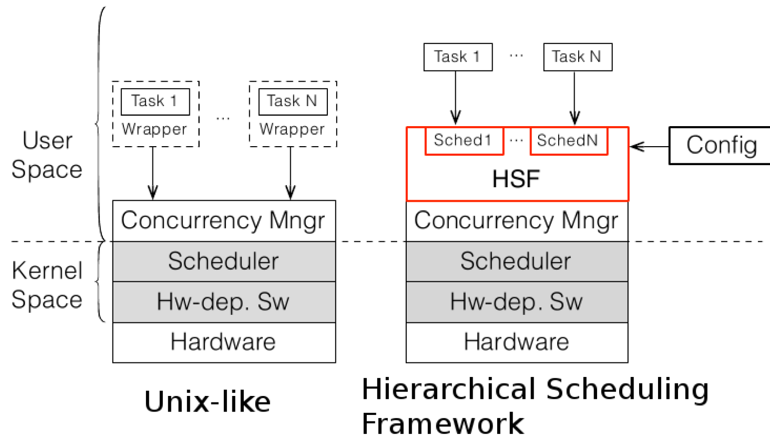
3 Earliest Deadline First with Virtual Deadlines

4 Mixed Criticality Resource Flow

5 Conclusion

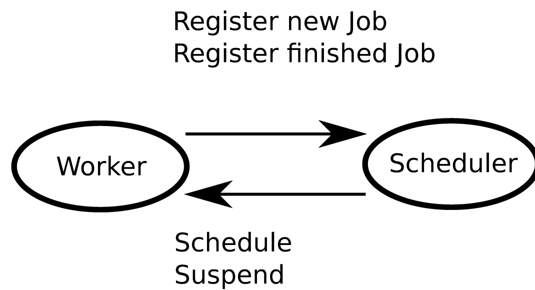
Hierarchical Scheduling Framework[2](HSF)

- Scheduling structure of Hierarchical Scheduling Framework

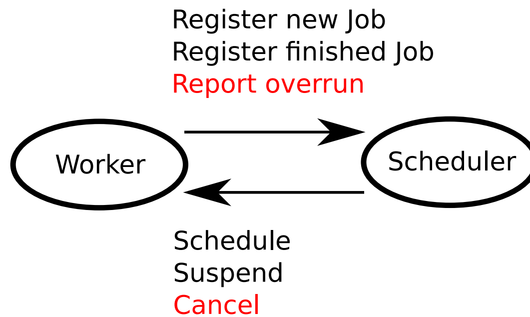


[2]A. Gomez, Hierarchical Scheduling Framework - A Programmer's Manual, 2013

Hierarchical Scheduling Framework



Mixed Criticality Scheduling Framework (MCSF)



Added Features to allow Mixed Criticality scheduling:

- Randomized execution Times
- Monitoring (Reporting Overruns)
- Dynamic Scheduling (Canceling Tasks during execution)
- Overhead measuring and calculation

Outline

1 Introduction

2 Mixed Criticality Scheduling Framework

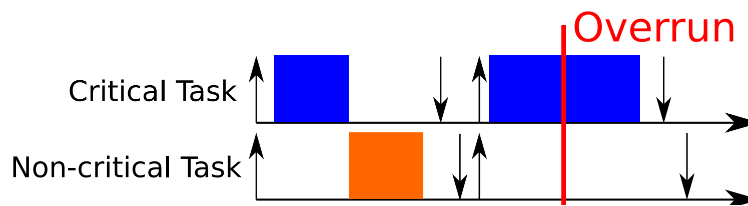
3 Earliest Deadline First with Virtual Deadlines
- Theory
- Implementation
- Simulation setup
- Results

4 Mixed Criticality Resource Flow

5 Conclusion

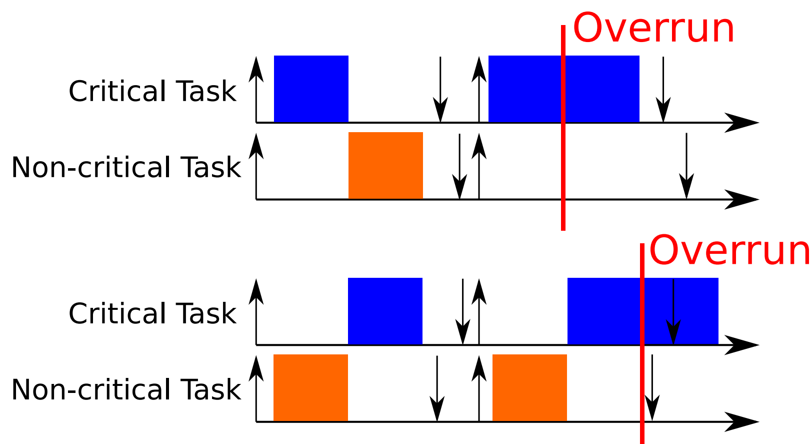
Earliest Deadlines First with Virtual Deadlines[3](EDF-VD)

- Queue of all tasks sorted by Deadline
- Two modes:
  - Low critical mode: assume typical Case Execution Times, run all tasks.
  - High critical mode: assume Worst Case Execution Times, only run critical tasks.



[3]Sanjoy K. Baruah , Vincenzo Bonifaci , Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster and Leen Stougie, Mixed-Criticality Scheduling of Sporadic Task Systems

Virtual Deadlines



Deadlines might get missed. Thus introduce Virtual Deadlines

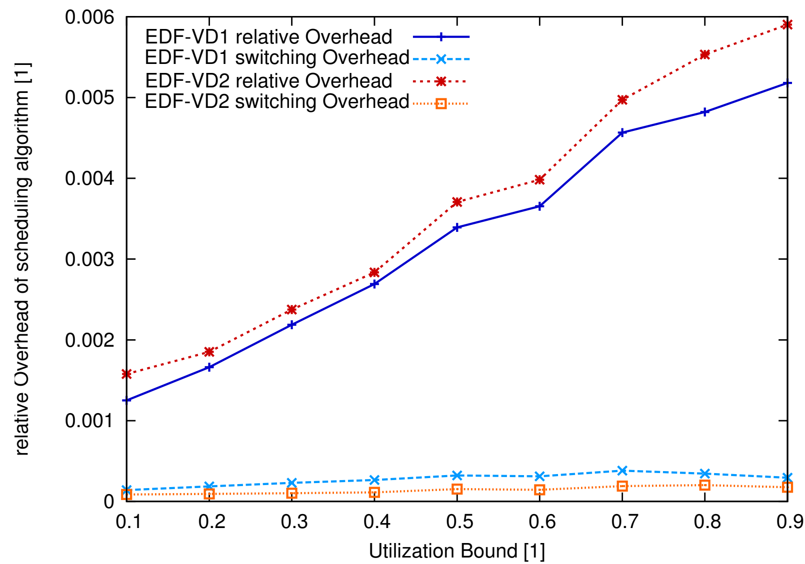
Implementation

- Implementation using 1 queue (EDF-VD1)
  - small average overhead, large switching overhead
- Implementation using 2 queues (EDF-VD2)
  - large average overhead, small switching overhead
- Implementations switch back from Critical to non-critical Mode when CPU is idle

Simulation Setup

- Randomly generated Task sets
  - System utilization uniform 10% to 100%
  - Probability of overrunning uniform 0% to 100%
- 10.000 Simulations for 2 seconds each
- Run time of a single task $10\mu s \cdots 1ms$

Results of Simulation Run



Outline

1 Introduction

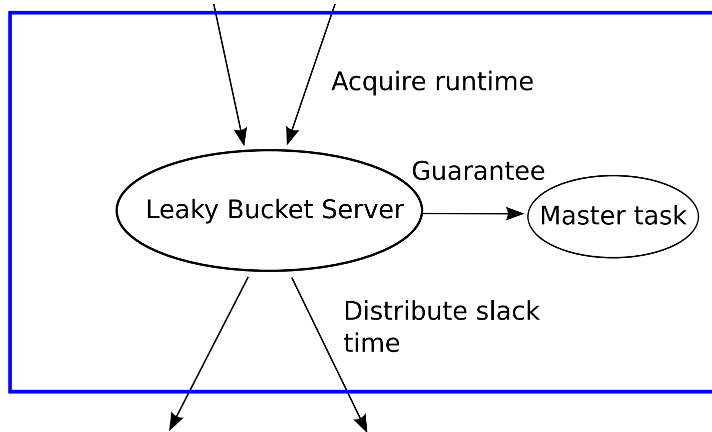
2 Mixed Criticality Scheduling Framework

3 Earliest Deadline First with Virtual Deadlines

4 Mixed Criticality Resource Flow
- Theory/Implementation
- Simulation setup
- Results

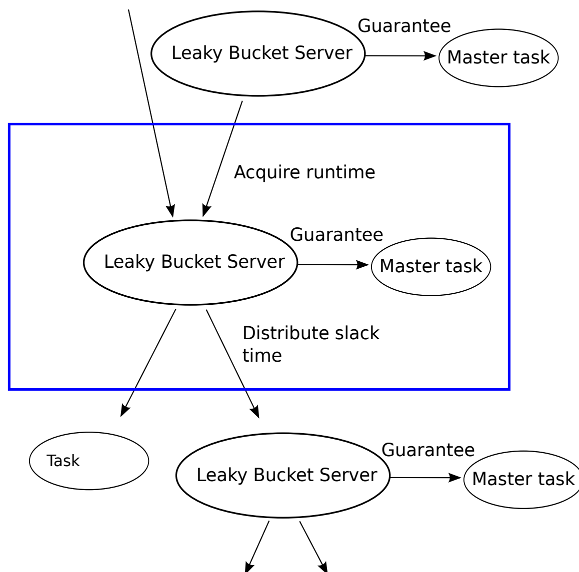
5 Conclusion

Leaky Bucket Server[4]

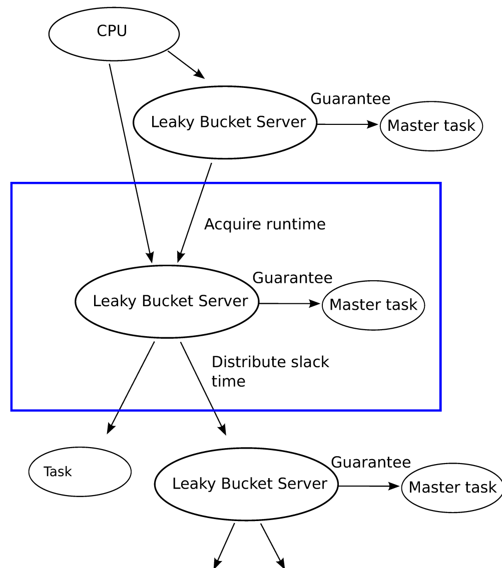


[4] P. Huang, Efficient Resource Flowing in Interference Constrained Mixed-Criticality Systems, 2013

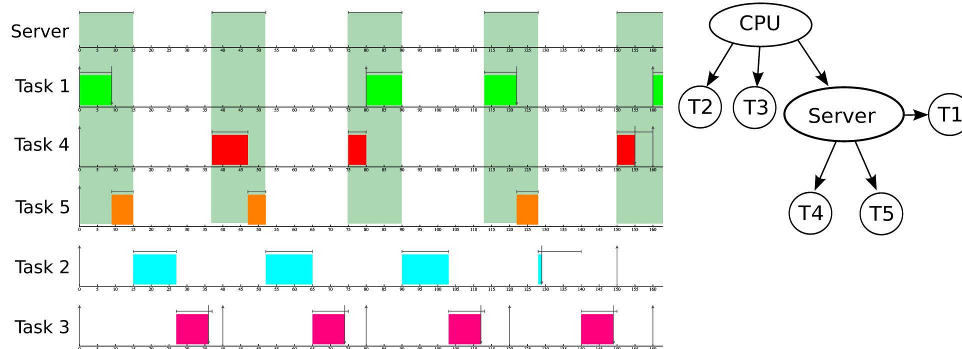
Leaky Bucket Server



Mixed Criticality Resource Flow Scheduling



Mixed Criticality Resource Flow (MCRF) Implementation



- Slack time is distributed in a round robin fashion

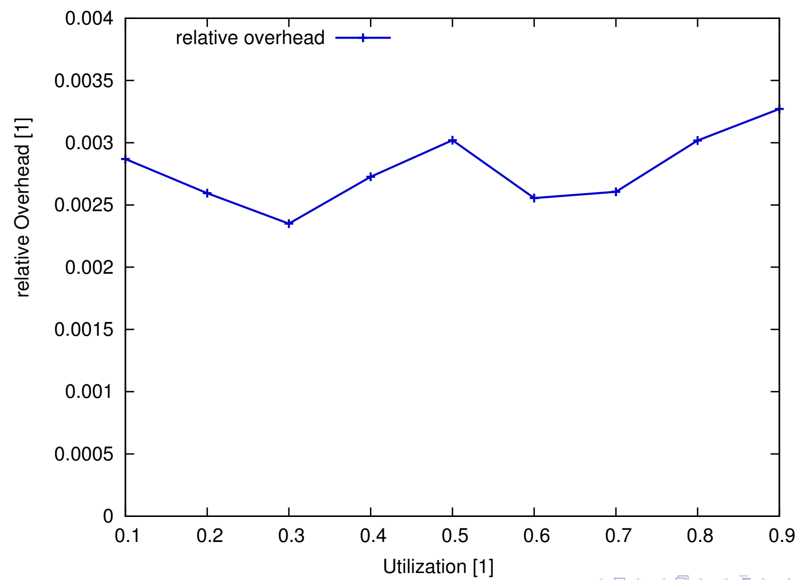
Implementation of Mixed Criticality Scheduling approaches
Mixed Criticality Resource Flow
Simulation setup

Simulation Setup

- Randomly generated Task sets
  - System utilization uniform 10% to 100%
  - Probability of overrunning uniform 0% to 100%
- 5.000 simulations for 30 seconds each
- Run time of a single task 10ms $\cdots$ 1 s

Implementation of Mixed Criticality Scheduling approaches
Mixed Criticality Resource Flow
Results

Results of Simulation Run



Outline

1 Introduction

2 Mixed Criticality Scheduling Framework

3 Earliest Deadline First with Virtual Deadlines

4 Mixed Criticality Resource Flow

5 Conclusion

Conclusion

- Mixed Criticality Scheduling Framework is usable for different manners of schedulers and can be extended easily
- Earliest Deadline First Implementations have low overhead, but increases with increasing system utilization
- Mixed Criticality Resource Flow has low overhead independent of system utilization

Implementation of Mixed Criticality Scheduling approaches
└ Conclusion

Questions?

Bibliography

[1] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *Proceedings of the 19th European conference on Algorithms*, ESA'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] A. Gomez. Hierarchical scheduling framework - a programmer's manual, 2013.

[3] P. Huang. Efficient resource flowing in interference constrained mixed-criticality systems. unpublished article, 2013.

[4] H. Li and S. Baruah. Outstanding paper award: Global mixed-criticality scheduling on multiprocessors. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 166–175, 2012.

[5] J. Stankovic. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Real-time systems series. Kluwer Academic Publishers, 1998.