CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Integrating Fault-Tolerance in Real-Time Scheduling of Mixed-Criticality Systems on Multiprocessors

Master's thesis in Computer Systems and Networks

PHILIP STÅLHAMMAR

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Integrating Fault-Tolerance in Real-Time Scheduling of Mixed-Criticality Systems on Multiprocessors

PHILIP STÅLHAMMAR

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Integrating Fault-Tolerance in Real-Time Scheduling of
Mixed-Criticality Systems on Multiprocessors
PHILIP STÅLHAMMAR

Supervisor: RISAT PATHAN, Department of Computer Science and Engineering
Examiner: JAN JONSSON, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Integrating Fault-Tolerance in Real-Time Scheduling of
Mixed-Criticality Systems on Multiprocessors
PHILIP STÅLHAMMAR
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents a novel real-time global fixed-priority scheduling algorithm that integrates mixed-criticality and fault-tolerance. An important aspect of mixed-criticality is to ensure that the *temporal* correctness of safety-critical tasks is not impacted by non-safety critical tasks they share a platform with. This thesis proposes that safety-critical tasks are given fault-tolerance to also ensure *functional* correctness. By integrating fault-tolerance the algorithm can provide guarantees for the *temporal* and *functional* correctness of safety critical tasks. Fault-tolerance is given through the execution of backup tasks where these tasks may be simple a re-execution of the primary or a diverse implementation. In addition, the backup tasks can be scheduled as either passive or active backups. The thesis includes an analysis to derive a schedulability test for the proposed algorithm and also presents policies to determine how many backups should be active for each task assuming a certain fault model. Simulated tests show that mixed-criticality and fault-tolerance can be costly in terms of schedulability, but that assigning active backups according to the policies proposed by the thesis can provide an increase in schedulability over having just passive backups.

Keywords: Real-Time Systems, RTS, Mixed-Criticality, Fault-Tolerance, Global Scheduling, Fixed-Priorities.

Acknowledgements

Thank you to my supervisor Risat Pathan whose knowledge and insight have helped tremendously to complete this project.

Philip Stålhammar, Gothenburg, June 2018

Contents

1

Introduction

In real-time scheduling there are tasks that have *temporal* correctness requirements, *temporal* correctness is that the tasks must produce a result within a given time window, and results produced outside of this window are considered incorrect [26], for example, a value read from a sensor might only be valid for a short amount of time, so tasks working on that sensor data have to finish before the data is no longer valid. The end of such a time interval is called the deadline of the task and the goal of a real-time scheduler is to ensure that all real-time tasks complete their execution before their deadline. A task-set is said to be schedulable if all its are schedulable i.e. they meet all their deadlines.

Mixed-criticality is a concept of increasing importance where tasks of differing criticality levels execute on the same processing system [9]. Criticality levels of tasks can be given from their function in accordance with some standard like ASIL, which is part of the larger automotive standard ISO 26262 [17], functions that are critical to safety will have a high criticality level, and Certification Authorities (CA) requires greater guarantees that the high criticality tasks will function correctly and meet the standards of that safety/criticality level. These guarantees can come in the form of a more pessimistic, then what developers would normally use, estimation of a tasks worst-case execution time (wcet). Since correctly estimating the wcet of a task is not a clear-cut problem [21], having a more pessimistic estimation lowers the chances that it is wrong. One issue with integrating real-time scheduling and mixed-criticality is that criticality level is independent of the tasks real-time priorities, i.e. a low priority task from a real-time perspective may have a high criticality level and vice versa.

As mentioned previously real-time schedulers focus on providing *temporal* correctness to tasks, but it is also important, especially for safety critical task, to have *functional* correctness, i.e. the task produces the correct output. There are multiple reasons why a task might produce erroneous output, from software bugs, to design faults in hardware, to bit-flips in registers caused by background radiation and more [18]. The good news is that many of these faults can be handled using fault-tolerance, a common way to handle faults is have some form of fault detection and then re-execute a faulty task in the hope that the fault was temporary.

It should be obvious that having fault-tolerance via re-execution introduces new

challenges to real-time scheduling, with one being increased resource usage. This is where multicore processors have benefits over single-core processors, in that they offer increased performance for their size, and lower power consumption for their performance [21], both important attributes for embedded systems where real-time applications are common. Multicore processors also have a measure of built-in hardware redundancy with multiple cores that can work independently, depending on the fault, not all cores might be affected. Multicore processors are an increasingly more common type of hardware that has a number of benefits over single core processors.

This thesis, therefore, presents the design and evaluation of a novel real-time scheduling algorithm using global scheduling with fixed-priorities for a multiprocessor environment. In global scheduling all processors in a system share a ready queue, where tasks that are ready to execute are stored, it is the scheduler's job to assign these tasks to the processors for execution. Fixed-priorities are set offline and do not change during runtime. The proposed scheduling algorithm will handle task sets with mixed-criticality levels and also offers fault-tolerance for high criticality tasks in terms of re-execution in case a fault is detected in such tasks. Areas where mixed-criticality task-sets are used and fault-tolerance is required are for example the automotive and aeronautic industries [21]. The real-time scheduler that this thesis presents integrates the concepts of mixed-criticality with fault-tolerance to make the high criticality tasks more reliable and thus improve the safety and performance of the system they are part of.

1.1 Problem Statement

Design a real-time multiprocessor scheduling algorithm that can schedule mixed-criticality task-sets with fault-tolerance guarantees for high criticality tasks, the algorithm will use global scheduling with fixed-priorities. Since fixed-priorities are assigned offline and the fact that priorities affect the schedulability of tasks, a method to assign priorities needs to be found.

The fault-tolerance will consist of the execution of active and passive backup tasks, active backups are always released to the scheduler at the same time as the primary, even when no faults are detected, this means that they can execute in parallel. On the other hand, passive backups for a task are only released when a fault has been detected in the executing task and no other variant (primary or backup) of that task is currently executing or waiting to execute, in effect the passive backups are released sequentially. Given that active and passive backups have different properties there is a need to design policies that decide the number of active backups each high criticality task has. The inclusion of fault-tolerance will add extra work to the system which may affect the schedulability negatively.

A real-time task-set requires that the tasks are able to meet their deadlines i.e. be schedulable, and the schedulability of a task-set should be proven before it becomes part of a live system. A schedulability test will determine the schedulability of a task-

set for a given scheduling algorithm. Thus to test the schedulability of task-sets on the proposed scheduling algorithm a schedulability test needs to be derived. Deriving the schedulability test requires finding suitable task and fault models, the task model will determine what properties will represent a task, and the fault model determines properties like what type of faults can occur, their number and frequency, how accurately they are detected etc. The models together with the scheduling algorithm will be used to derive a schedulability test used to determine the schedulability of task-sets on the proposed scheduling algorithm.

The performance of the proposed scheduling algorithm will be measured by its ability to schedule task-sets if a task-set is schedulable is determined by the schedulability test. The schedulability test will be tested by generating a number of random task-sets and measuring the ratio of tasks that are schedulable. To test the schedulability test in a meaningful way it needs to be implemented in code from its mathematical equations, this process is not trivial since the equations can be computationally heavy.

1.2 Methodology

The scheduling algorithm will use global fixed-priority scheduling. The mixed-criticality is modeled with two criticality levels, low and high, and conversely, the scheduler has two modes corresponding to these levels. In low criticality mode, all tasks are allowed to execute, and it is assumed that the high criticality tasks have a lower or less pessimistic wcet. If during runtime any high criticality task executes for longer than its low wcet then the scheduler switches to high criticality mode where only high criticality tasks are allowed to execute, low criticality tasks are dropped to free up resources for the high criticality tasks. In high criticality mode, the high criticality tasks wcet may be higher, or more pessimistic than the one used in the low criticality mode.

High criticality tasks are also given fault-tolerance through the use of backup tasks, either passive or active. Active backups are always released together with the primary and they can execute in parallel with each other and the primary. Active backups are released even if no faults have been detected, this means that active backups may increase the workload of the system without providing a benefit in terms of correcting faults, but on the other hand, active backups can execute in parallel which means a correct result may be available earlier. A Passive backup is released only when the primary, all active backups, and previous passive backups have failed to produce a valid result, as a consequence of this release pattern passive backups execute sequentially. Since passive backups are released only when needed they provide a minimal increase in workload, but since they execute sequentially, the time it takes until a valid result is available might be longer than if the equal number of active backups was used instead. So in a simplified summary: Active backups trade an increase in workload for a reduction in time before a valid result,

and the passive backups trade increase in time before a valid result for a reduction in workload compared to having the backup as active.

The schedulability test consists of two parts, one for each criticality level. The schedulability test will investigate a job from each task in the task-set, these jobs will be made to suffer the maximal amount of interference from higher priority jobs, and if the job still meets its deadline then all other jobs from that task are also guaranteed to meet their deadlines. The general analysis each part of the test performs is the same, first find the number of jobs from each higher priority task that interfere with the investigated job. Secondly, the workload of these interfering jobs must be calculated, and since high criticality jobs have backups their workload can differ based on the number of faults they encounter, thus the fault distribution that produces the maximal workload must be found. Next, the workload of the investigated job will be calculated, this includes the workload of any active backups. The previous workloads can be done in parallel, but the workload of passive backups for the investigated job can't be done in parallel. Next, the fault distribution between the investigated job and the interfering jobs must be found, again to maximize the workload. Lastly if the time it takes to execute the total workload is within the deadline of the investigated job then that job, and the task it belongs to is schedulable. The main difference with the high criticality analysis is to find the point in time that produces the highest total workload, taking into account that after that point low criticality tasks are no longer allowed, so the interfering workload decreases, but on the other hand high criticality jobs have a larger workload.

1.3 Limitations

This thesis focuses on designing a scheduler that uses fixed-priorities so dynamic-priorities will not be considered. The analysis of the mixed-criticality functionality will be limited to two criticality levels, with no function to return to a low criticality state once it has gone to high criticality. The schedulability analysis will be done with the fault model presented later in section 3.1, but there are other equally valid models. It is assumed that all task errors will be detected at the end of task execution at the latest, a more probabilistic analysis of error detection is outside the scope of this thesis.

2

Background

This section will present some of the theory behind real-time scheduling and will focus on the areas that this paper concerns namely multiprocessor scheduling, mixed-criticality scheduling and fault tolerance.

2.1 Multiprocessor Scheduling

Real-time multiprocessor scheduling comes in two main shapes global and partitioned scheduling [26]. In partitioned scheduling tasks are assigned offline to a core from which they can't migrate, i.e. a task assigned to a given core can only ever execute on that core. The benefit of this approach is that uniprocessor scheduling theory can be used to schedule the tasks on each individual core. The main drawbacks are inflexibility, and wasted resources, i.e. one core can remain idle while there are tasks in waiting on other cores.

Global scheduling, on the other hand, means that the scheduler assigns tasks to cores at runtime, and they are free to migrate to different cores during their execution. The migration process can be costly in terms of overhead, but in return tasks can execute on any free core, helping to prevent idle cores.

Global scheduling has a property that makes it harder to design schedulability test namely the lack of a predictable critical instant. A critical instant [19] is the moment in time where a task suffers the most interference from higher priority tasks, i.e its execution is delayed the most. The critical instant for a task set is when the lowest priority task faces the most interference, in uniprocessor fixed priority scheduling the critical instant happens when the lowest priority task is released for execution at the same time as all higher priority tasks. And if that task is able to meet its deadline in that situation than it will meet its deadline in any other situation. In global scheduling, on the other hand, the critical instant is not guaranteed to happen when all tasks are released at the same time, which makes it more challenging to design a schedulability test for global scheduling.

2.2 Mixed-Criticality

Mixed-criticality is a concept that mixes tasks of different criticality levels on the same processor. The criticality levels are given by certain standards like ASIL from the ISO 26262 standard. Functionality/tasks that concern safety critical properties are given a high criticality level, and CAs place their focus on the high criticality tasks working correctly from both a temporal and functional point of view. Usually, mixed-criticality in real-time scheduling is handled by giving the high criticality tasks a more pessimistic estimation of their worst-case execution time (wcet) than what the designers would give them if they were of a lower criticality. This pessimism means that fewer task sets can be scheduled and it is likely that there is slack, i.e unused processor time, in the schedule that lower criticality tasks can make use of [21]. Another variation is to assume two (or more) different wcets for high criticality tasks [9], [22]: The first one is larger and more pessimistic and will satisfy the CAs requirement that the real wcet will not exceed the estimated one. And the second wcet is estimated to be smaller and more realistic according to the designers. The designers then schedule the task set with the smaller wcet, but if during runtime a high criticality task overruns the lower estimated wcet then the system will shift mode and for example remove low criticality tasks from execution, thereby freeing up resources to make sure that the high criticality tasks meet their deadlines.

2.3 Fault-Tolerance

Fault-tolerance, in general, is the ability of a service to detect and handle (or tolerate) faults in hardware and software. Faults come in three main types [18]:
Permanent faults are faults that won't go away by themselves, like a processor that failed and won't function again.
Transient faults are faults that are temporary and the affected component will revert back to a normal state after some time, like a bit flip in a memory cell caused by background radiation.
Intermittent faults where the component will switch between a correct and faulty state like a loose connection.

The most common way to handle faults is by using redundancy, the two main categories of redundancy are space and time redundancy. Space redundancy means having extra components that do perform the same function so that in case one component fails another can take over, this is the main way to tolerate permanent faults. For example, a computer system might have multiple processors not to increase computational performance but to protect against broken processors. Bugs in software and hardware are a type of permanent fault that can't be tolerated by having more copies of the same component, diverse components that fulfill the same functionality are required, for example, software that needs to perform sorting in a fault tolerant way might include multiple different sorting algorithms with varying

levels of efficiency and complexity.

Time redundancy is to perform the same task again in the hope that the fault was temporary, which makes it the main approach in dealing with transient faults, in software it would involve re-executing the failed task. For time redundancy to work the functionality must have some leeway time-wise so that re-execution has time to complete.

Active backups are always executed, this makes it easier to determine if the task will meet its deadline even with faults, but the downside of active backups is that they are executed even in the absence of faults. *Passive* backups, on the other hand, are only executed once their primary task experience a fault, this makes them trickier to schedule since there must be sufficient amount of slack after the fault in the primary was detected to allow the passive backup time to finish executing before the deadline. The positive aspect of passive backups is that they only run when a fault is detected thereby saving resources. So in a situation with a relatively low amount of faults passive backups should be preferred, certain tasks might always require active backups to have fault-tolerance. An example of such a situation is if the execution time of the primary and the backup together would exceed the deadline then the backup can't be run in passive mode but must run in active mode.

2.4 Related Work

There is previous work in the areas of global scheduling, mixed-criticality and fault-tolerance. In [2] Andersson et al. presents a global scheduling rate-monotonic scheduling algorithm that was an extension of the uniprocessor rate-monotonic scheduling algorithm. Much work has gone into improving the schedulability analysis of global real-time scheduling for both fixed- and dynamic-priorities, a selection of this work can be seen in [15], [8], [10], [4]. In [14] Davis and Burns shows that Audsleys OPA algorithm [3] for assigning task priorities work with multiprocessor scheduling algorithms.

In [6] Baruah et al. introduce and analyze a uniprocessor scheduler that can handle mixed-criticality task-sets, they also present a formal model for how mixed-criticality task-set can be represented. In [7] Baruah et al. extends and improves upon the previous analysis, this analysis is further extended by Burns and Davis in [11] where they show that by adding regions where tasks can't be preempted that schedulability can be improved. In [22] Pathan presents an analysis of a mixed-criticality scheduler for use on multiprocessors, this analysis is used in the schedulability analysis for the algorithm this thesis proposes. In a bid to make mixed-criticality scheduling more robust Bate et al. [7] present analyses for allowing low criticality tasks back after they have been dropped by a criticality switch. In [5] Baruah and Guo consider mixed-criticality scheduling on processors that have varying speeds, normal and impaired, with the goal of having all tasks be schedulable at normal, and at least have the high criticality task schedulable in the impaired state.

In [25] Pathan and Jonsson present the analysis of a multiprocessor fault tolerant scheduler, the fault-tolerance is achieved by executing passive backup tasks. The main difference with the proposed algorithm is once again the use of mixed-criticality, but also that the proposed algorithm uses both active and passive backup tasks. The analysis in [12] by Chen et al. considers fault-tolerance through task replication, where there are multiple independent copies of a task in the system, this method of fault-tolerance is comparable to the active backups the proposed algorithm uses. In [27] Salehi et al. propose a check-point system to handle fault-tolerance, with a focus on energy-efficiency. In [24] Pathan presents a multiprocessor scheduling algorithm with fault-tolerance using two different fault models, and achieving fault-tolerance through active and passive backup tasks, the fault models used in Pathans paper are more advanced than the single model used for the proposed algorithm, but in turn the proposed algorithm consider mixed-criticality which [24] does not.

In [20] Liu et al. combines fault-tolerance and mixed-criticality for partitioned scheduling, with a goal to minimize the number of task reallocations while still preserving the function of high criticality tasks. Huang et al. [16] shows that they can model a mixed-criticality system with fault-tolerance through re-executions as just a mixed-criticality problem. The authors of [1] examine the design of mixed-criticality systems that can tolerate permanent processor failures. In [28] Thekkilakattil et al. explore fault-tolerance on a mixed-criticality system in a distributed system, their method maximizes the amount of resources available for low criticality tasks while ensuring that the high criticality tasks function correctly. The real-time scheduling algorithm presented by Pathan in [23] handles the concepts of fault-tolerance and mixed-criticality in a real-time context on uniprocessors. One of the main differences between this scheduler and the proposed one is that that it only works on uniprocessors, and uniprocessor scheduling theory is not directly applicable on multiprocessors [26].

3

Scheduling Algorithm

This section presents the proposed scheduling algorithm in greater detail and also present the task and fault models that the algorithm is designed for.

3.1 Task and Fault Models

The paper will consider the following task model, an application will be modeled like a set $\Gamma = \{\tau_1 ... \tau_n\}$ of n sporadic tasks where each task has the following properties $\tau_i =< T_i, D_i, \lambda_i, \bar{B}_i^{LO}, \bar{B}_i^{HI}, h_i >$ where:

- T_i gives the tasks minimum inter arrival time, commonly referred to as the tasks period.
- D_i gives the tasks relative deadline, since this paper considers constrained deadlines $D_i \leq T_i$.
- λ_i indicates the tasks criticality level, LO for low criticality and HI for high.
- $\bar{B}_i^{LO}$ is a vector $< E_{i,LO}^0, E_{i,LO}^1, E_{i,LO}^2 ... >$ with $E_{i,LO}^0$ being the wcet, in low criticality mode, of the primary and $E_{i,LO}^1$ the wcet of the 1st backup etc.
- $\bar{B}_i^{HI}$ is a vector $< E_{i,HI}^0, E_{i,HI}^1, E_{i,HI}^2 ... >$ with $E_{i,HI}^0$ being the wcet, in high criticality mode, of the primary and $E_{i,HI}^1$ the wcet of the 1st backup etc.
- h_i gives the number of active backups that task τ_i has, i.e. versions of the task that are released together and can run in parallel.

A task τ_i will generate an infinite sequence of jobs with job J_i^k, $k > 0$, being the k:th job of task τ_i. The wcet in low criticality mode will be referenced as low wcet and high wcet in the case of high criticality mode. The backups of a task are given the same fixed priority as the primary relative to other tasks, but internally the primary has higher priority than the first active backup, that in turn has higher priority than the second active backup etc. Also, note that only tasks with high criticality level will have backups. For tasks that experience more faults then they have backups,

the missing backups will be re-executions of the primary.

The fault model will consider both transient faults and permanent hardware faults, for transient faults we assume that a maximum of f faults can occur in a given time interval D_{max} which is the largest relative deadline, also that a maximum of ρ cores can suffer a permanent fault and be put out of commission. Another assumption is that errors will at the latest be detected when jobs signal for completion, specifically how errors are detected is beyond the scope of this paper, but a TMR approach could be modeled with two active backups.

3.2 Scheduling Algorithm

The following points will provide a high-level overview of how the scheduling algorithm is designed to work.

- When a job of task τ_i is released, it and all of its active backups are placed in the ready queue, if there are idle cores then the scheduler will assign the job, giving priority to the primary job over the active backups.
- If there are no idle cores then the scheduler will look to see if there are any jobs from tasks with lower priority currently executing and if so it will preempt them, and they will be returned to the ready queue. If there are no idle cores and no preempt-able jobs then the released job will wait in the ready queue.
- If a high criticality job is detected as faulty and no other backup of that task is running then the next passive backup is released for execution, and new passive backups can be released as error are detected.
- When a job has signaled an error-free completion then no more backups are released and currently waiting or running backups are dropped.

If at any point a job of a high criticality task has not signaled for completion (or fault) before executing for more than the low criticality wcet then the scheduler will move into high criticality mode, and tasks with low criticality level will be dropped, giving the system more resources to execute the high criticality tasks.

Figure 3.1 provides a simple example of how some of the schedulers features work. It shows jobs from three different tasks, τ_1 and τ_2 are both high criticality tasks, while τ_3 is a low criticality task. The jobs of τ_1 are used to show the fault-tolerance aspects of the algorithm, thus τ_1 has been given one active backup denoted as $J_{1,A}$ in the figure, conversely $J_{1,P}$ denotes a passive backup. The red triangles indicate that a fault has been detected in that job, the green triangle means that a correct result has been obtained for a job that has experienced faults. We see that the active backup executes at the same time as the primary and that no passive backup is released when a fault occurs in the active backup, the passive is only released when the only

remaining job, in this case, the primary, also fails to produce a correct result. The second release of τ_1 experience no faults, so no passive backups are released. At the point in time marked by s and the yellow triangle, the scheduler switches into high criticality mode since the execution time of J_2^1 has exceeded the low wcet. After the switch to high criticality mode, all low criticality tasks are dropped this means that J_3^1 can not finish execution, the high criticality tasks also use the high wcet as seen by the second release of τ_1 that have a larger time block allotted.

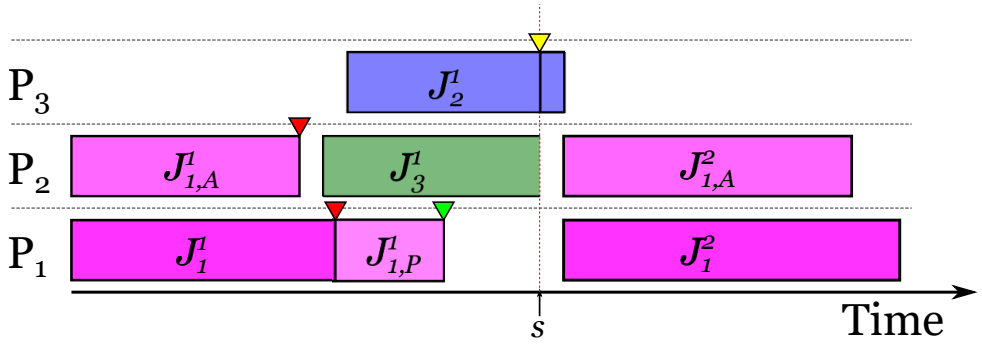


Figure 3.1: An example that shows the 2 main features of the scheduling algorithm, mixed-criticality, and fault-tolerance. The red triangles indicate that a fault has been detected in that job, the green triangle means that a correct result has been obtained for a job that has experienced faults, and the yellow triangle indicates that the execution time of a high criticality job has exceeded the low wcet. The red dotted line indicates the time when the scheduler switches from low to high criticality mode.

4

Schedulability Analysis

This section will cover the schedulability analysis that is used to derive a schedulability test. It is divided into subsections that cover the analysis of the low and high criticality modes, and general properties that are shared by both analyses.

4.1 General Properties

The schedulability of a task, τ_i, will be determined by finding the response time of a generic job J_i of the task, the response time is the time it takes from the release, r_i of J_i, to the successful completion, this is shown in figure 4.1. In order to find the largest response time, J_i will be made to suffer the maximum amount of interference, where interference is the time that J_i is available for execution but the processors are busy executing higher priority jobs. If it is still schedulable, i.e. the response time is smaller or equal to the deadline, then all other jobs of τ_i are schedulable and by extension task τ_i is schedulable. The scheduling window of job J_i is defined as the time interval that J_i is available for execution i.e. the interval $[r_i, d_i)$ with length D_i, where r_i is the release time of J_i and d_i the absolute deadline, see figure 4.1 for a visual explanation.

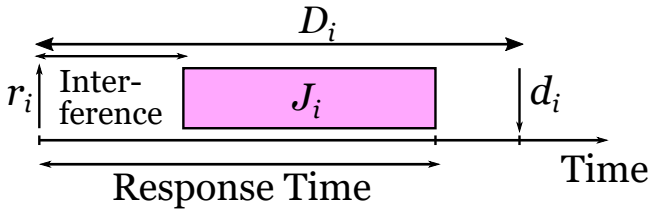


Figure 4.1: r_i denotes the point in time when J_i is released for execution, the deadline of J_i is given by d_i is the point in time at which J_i must have finished execution, D_i is the amount of time that J_i has to produce a result. The response time is the amount of time it takes, from r_i to the point in time that J_i finishes. Finally, interference is the amount of time that J_i is available for execution but is not executing.

Since this algorithm employs static priorities only jobs from higher priority tasks

will affect the schedulability, let $hp(i)$ denote the set of tasks with higher priority than task τ_i, $hpL(i)$ is the set of tasks with higher priority than τ_i but with low criticality, and finally $hpH(i)$ is the set of higher priority tasks than τ_i and with high criticality.

$$hp(i) = hpL(i) \cup hpH(i)$$

Let $C_{i,\lambda}^{f}$ be the total workload of task τ_i in criticality mode λ, suffering from f errors, and is given by the following equation:

$$C_{i,\lambda}^{f} = E_{i,\lambda}^{0} + E_{i,\lambda}^{1} + ...E_{i,\lambda}^{max\{h_i,\ f\}} \tag{4.1}$$

Note that only high criticality tasks have backups, so $C_{j,LO}^{f}$ for a low criticality tasks τ_j is $E_{j,LO}^{0}$. The primary and the active backups can run in parallel but the passive backups run sequentially so their workload must be separated, let $\widehat{C}_{i}^{f}$ denote the workload of the passive backups of task τ_i then

$$\widehat{C}_{i,\lambda}^{f} = C_{i,\lambda}^{f} - C_{i,\lambda}^{h_i} \tag{4.2}$$

If τ_i is a low criticality task then $\widehat{C}_{i,LO}^{f} = 0$ since low criticality task don't have backups.

Additionally, let $\widehat{M}$ denote the number of working cores where M is the total number of cores (working and faulty) on the platform, i.e. $\widehat{M} = M - \rho$, and as a reminder ρ is the number of faulty cores. Also let $e_i = (je_i + \rho)$ denote the total number of failures that occur in the scheduling window of job J_i, where je_i denotes the maximum number of task failures.

4.2 Low Criticality Mode

This section describes the main principle of the low criticality part of schedulability test and the schedulability analysis of the low criticality mode.

4.2.1 General Procedure Overview

The purpose of this part of the schedulability test is to determine the schedulability of all tasks in low criticality mode. The procedure to determine the schedulability of a job J_k is in general:

- Find the number of interfering jobs from each higher priority task.
- Calculate the workload these interfering jobs produce, additionally find the fault distribution among the jobs that gives the maximal workload.
- Calculate the workload of J_k, including active backups.

- If J_k is a high criticality task, find the fault distribution between J_k and the interfering jobs that give the maximal workload.
- Calculate the response time of J_k using the previously calculated workloads.
- Check the schedulability of J_k.

4.2.2 Low Criticality Analysis

In low criticality mode, all tasks are available for execution, and generic job J_k of task τ_k will suffer interference from jobs of the tasks in $hp(k)$. Determining the interference on J_k involves finding the sets HJ_K, HJN_k and HJH_k, which are the set of jobs belonging to tasks in $hp(k)$, $hpL(k)$ and $hpH(k)$ that are active in the scheduling window of J_k. Since it is not known exactly when the higher priority jobs are active, HJ_k will be an upper bound of the possible interfering jobs. The sporadic task model assumes that jobs from the same task must be separated by at least T_i time units.

If $D_k < (T_i - D_i)$ then only one job of τ_i can interfere so $J_i^1 \in HJ_K$, on the other hand if $D_k \geq (T_i - D_i)$ then an additional $\lceil \frac{D_k-(T_i-D_i)}{T_i} \rceil$ jobs can interfere, summing up these two contributions gives us the following equation for finding the number of interfering jobs from a higher priority task τ_1:

$$N(i) = \left\lceil \frac{max\{0,\ D_k - (T_i - D_i)\}}{T_i} \right\rceil + 1 \tag{4.3}$$

The first term in the max function covers the case where $D_k < (T_i - D_i)$ and the other term the reverse. So now that we know the number of interfering jobs, HJ_k is given by:

$$HJN_k = \bigcup_{\tau_i \in hpL(k)} \{J_i^1, J_i^2, ...J_i^{N(i)}\} \tag{4.4}$$

$$HJH_k = \bigcup_{\tau_i \in hpH(k)} \{J_i^1, J_i^2, ...J_i^{N(i)}\} \tag{4.5}$$

$$HJ_k = HJN_k \cup HJH_k \tag{4.6}$$

Next the workload of HJ_k needs to be calculated, let $W^c(HJ_k)$ denote the workload of HJ_k when suffering c faults, the tasks suffering faults will be chosen so to maximize the workload. Since the jobs in HJN_k don't have fault-tolerance any faults affecting these tasks won't increase the workload, so it is safe to assume zero faults affecting these tasks when calculating the workload of this set, equation (4.7) shows how the workload for the low criticality jobs is calculated.

$$W^0(HJN_k) = \sum_{J_i^p \in HJN_k} C_{i,LO}^0 = \sum_{J_i^p \in HJN_k} E_{i,LO}^0 \tag{4.7}$$

In the end, the workload of the low criticality jobs is just a summation of their wcet.

Calculating the workload for $W^c(HJH_k)$ is more complicated since the high criticality tasks has fault-tolerance that may increase the workload through having to execute passive backups, so finding which jobs increase the workload the most when hit by faults, in other words calculating $W^c(HJH_k)$ involves finding the fault distribution among the jobs that produce the maximal workload. Calculating $W^c(HJH_k)$ is done through recursion using equation (4.8) as a basic element, this equation basically gives the workload of a single job.

$$W^f(\{J_i^p\}) = C_{i,LO}^f \tag{4.8}$$

where $J_i^p \in HJH_k$, and assuming that f faults affect that job. Then the workload $W^c(HJH_k)$ can be calculated recursively as follows:

$$W^c(HJH_k) = \max_{f=0}^{c} \left\{ W^f(\{J_i^p\}) + W^{c-f}(\beta) \right\} \tag{4.9}$$

where $\beta = (HJH_k - \{j_i^p\},\ J_i^p \in HJH_k)$. The equation is given c number of faults that it can distribute among the jobs to maximize the workload, each step checks if it is better to apply faults to the selected job or if it is better to apply it to the rest if the jobs in β. We can now find the total interfering workload, $W^c(HJ_k)$, by simply taking the sum of the workloads produced by equations (4.7) and (4.9) as seen in the following equation:

$$W^c(HJ_k) = W^0(HJN_k) + W^c(HJH_k) \tag{4.10}$$

Next up the CPU time requirement for job J_k and its active backups is $\widehat{M} \cdot S_{k,\widehat{M}}$ where:

$$S_{k,\widehat{M}} = \max_{z=0}^{h_k} \left\{ E_{k,LO}^z + \frac{\sum_{b=0}^{z-1} E_{k,LO}^b}{\widehat{M}} \right\} \tag{4.11}$$

the first term in the max function covers the sequential execution, since a single job can't be executed in parallel, of the zth active backup and the second term covers that the primary and the rest of the active backups can execute in parallel. The time given by $\widehat{M} \cdot S_{k,\widehat{M}}$ is an upper bound on the CPU time it takes to complete the execution of J_k and all its active backups. We can show that $\widehat{M} \cdot S_{k,\widehat{M}}$ time units is enough to complete the execution of J_k and all its h_k active backups using the following equation for $z = 0,\ 1, ...h_k$:

$$\widehat{M} \cdot S_{k,\widehat{M}} \geq \widehat{M} \cdot E_{k,LO}^z + \sum_{b=0}^{z-1} E_{k,LO}^b \tag{4.12}$$

Under the proposed scheduler the primary (z = 0) has higher priority than the first active backup, $z = 1$, which in turn has higher priority than the second active backup, $z = 2$, etc. Given this the time required to complete the execution of the zth active backup includes the time required to execute the higher priority backups, $\sum_{b=0}^{z-1} E_{k,LO}^b$, in addition to its own execution time. After completing the workload of the higher priority backups there is $\widehat{M} \cdot E_{k,LO}^z$ time units left, spreading this time evenly over the working cores leaves the least amount of sequential time for the zth

backup to execute, even then $\frac{\widehat{M} \cdot E^z_{k,LO}}{\widehat{M}} = E^z_{k,LO}$ there is enough time left to complete the execution of the zth backup.

Assuming that the speed of the processors is 1, i.e a job with execution time 4 will take 4 time units to complete, then the total amount of CPU time that needs to be assigned to complete the execution of the higher priority jobs and J_ks primary and active backups is $(W^c(HJ_k) + \widehat{M} \cdot S_{k,\widehat{M}})$. Then $\lceil \frac{W^{e_k}(HJ_k) + \widehat{M} \cdot S_{k,\widehat{M}}}{\widehat{M}} \rceil$ gives us the time units required to execute the workload on a system with $\widehat{M}$ processors. Since the passive backups of J_k are only released after its primary and active backups have failed the time required to execute them is equal to their workload $\widehat{C}^f_{k,\lambda}$. By adding these two terms together we can find the response time for the low criticality mode, R^{LO}_k, of J_k, and if R^{LO}_k is smaller than the deadline D_k then J_k and by extension τ_k are schedulable. Now we have the required components to formulate schedulability conditions under the low criticality mode. The basic schedulability condition for low criticality tasks can be seen in equation (4.13).

$$R^{LO}_k = \left\lceil \frac{W^{e_k}(HJ_k) + \widehat{M} \cdot S_{k,\widehat{M}}}{\widehat{M}} \right\rceil + \widehat{C}^0_{k,LO} \leq D_k \tag{4.13}$$

Since fault-tolerance is not applicable to low criticality tasks equation (4.13) can be simplified further, $\widehat{C}^{(0)}_{k,LO}$ is always 0 for the low criticality tasks, and with no active backups $S_{k,\widehat{M}}$ just evaluates to $E^0_{k,LO}$. Applying these two changes gives us the final schedulability condition for the low criticality tasks in equation 4.14.

$$R^{LO}_k = \left\lceil \frac{W^{e_k}(HJ_k)}{\widehat{M}} + E^0_{k,LO} \right\rceil \leq D_k \tag{4.14}$$

For high criticality tasks the schedulability condition needs an additional element namely finding the final fault distribution between J_k and the jobs in HJ_k that produces the maximum response time for J_k. The fault distribution that maximizes the workload of HJ_k, given a certain number of faults, is already calculated in (4.9), so finding the final fault distribution just requires testing which combination of faults effecting J_k or HJ_k produce the maximal response time. For example with a total of 2 faults, the combinations to test would be: 2 faults effecting J_k and 0 HJ_k, 1 fault effecting each of then, and finally 0 faults effecting J_k and 2 HJ_k. With this in mind we can extend equation (4.13) into equation (4.15) where the max function finds the fault combination that gives the largest response time.

$$R^{LO}_k = \max_{c=0}^{e_k} \left\{ \left\lceil \frac{W^c(HJ_k) + \widehat{M} \cdot S_{k,\widehat{M}}}{\widehat{M}} \right\rceil + \widehat{C}^{(e_k - c)}_{k,LO} \right\} \leq D_k \tag{4.15}$$

which can be rewritten as

$$R^{LO}_k = \max_{c=0}^{e_k} \left\{ \left\lceil \frac{W^c(HJ_k)}{\widehat{M}} + S_{k,\widehat{M}} \right\rceil + \widehat{C}^{(e_k - c)}_{k,LO} \right\} \leq D_k \tag{4.16}$$

So the final schedulability conditions for the scheduler in low criticality mode are given in equations (4.14) and (4.16), for low and high criticality tasks respectively. Low criticality tasks that pass this part of the test are deemed schedulable but the high criticality tasks must also pass the high criticality part of the test to be deemed schedulable.

4.3 High Criticality Mode

This section contains the schedulability analysis to derive the high criticality part of the overall schedulability test.

4.3.1 General Procedure Overview

The purpose of this part of the schedulability test is to determine the schedulability of the high criticality tasks in high criticality mode. The procedure to determine the schedulability of a job J_k is mainly the same as the low criticality analysis but the main difference is that the main challenge is to find the point in time, s (see figure 3.1), that gives the largest response time for J_k. Before this point jobs from all tasks can interfere with J_k, while after only high criticality jobs can interfere but in return, these jobs may have a higher wcet and thereby produce more interference.

4.3.2 High Criticality Analysis

When the system switches to high criticality mode all tasks with low criticality level are thrown out, and will no longer be allowed to execute, while the high criticality tasks may have a higher wcet. When testing the schedulability of the high criticality mode there is no need to test the schedulability of low criticality task, therefore the high criticality analysis will only consider generic jobs from high criticality tasks. Let J_k be a generic job of the high criticality task τ_k.

Let s be the time relative to the release of high criticality job J_k, that the system switched to high criticality mode, i.e. at time $t > r_k + s$ the system is in high criticality mode. If s is larger than R_k^{LO} then J_k has already finished executing in low criticality mode and is thus not affected by the switch. Since we consider an integer time model, there are at most R_k^{LO} possible values of s, which leads to a schedulability test with pseudo-polynomial time complexity.

Since low criticality jobs are only active before time r_k+s they can only interfere with J_k during that portion of time instead of the whole scheduling window of J_k. Based on previous statements we have that $s \leq R_k^{LO} \leq D_k$ from this it follows that the length of the time intervals also follow the pattern of $[r_k, r_k + s) \leq [r_k, r_k + R_k^{LO}) \leq$

$[r_k, r_k + D_k)$. For the purpose of finding the number of interfering low criticality jobs the time that J_k is available to be interfered with can be reduced to s from D_k. By making this change we can change equation (4.3) that we used to find the total number of interfering faults into equation (4.17) that will give us the number of interfering low criticality jobs for a given s value.

$$NLC(i,\ s) = \left\lceil \frac{max\{0,\ s - (T_i - D_i)\}}{T_i} \right\rceil + 1 \tag{4.17}$$

This equation gives us an upper bound on the number of low criticality jobs that can interfere with J_k. Next, we need to find the number of interfering jobs from high criticality tasks. Since after point s the high criticality tasks may increase their wcet, and by extension increase their workload, we also need to find an upper limit to the number of jobs released after this point. With similar reasoning to finding the number of interfering low criticality jobs, the high criticality jobs with high wcet can also just effect J_k in a limited time interval, since they are only released after point s. The amount of time that these jobs can interfere with J_k is limited to $D_k - s$, now we can change equation (4.3) to take this reduced interval into account as seen in equation (4.18).

$$NH(i,\ s) = \left\lceil \frac{max\{0,\ (D_k - s) - (T_i - D_i)\}}{T_i} \right\rceil + 1 \tag{4.18}$$

This equation finds an upper limit to the number of high criticality jobs that are released in high criticality mode given a time s when the criticality mode switches.

There are still high criticality jobs unaccounted for, namely the jobs that released before point s. Since the switch to high criticality mode does not affect the release pattern of the high criticality tasks, the total number of interfering jobs from these tasks must be the same as the number calculated in equation (4.3) for the low criticality analysis. By removing the number of jobs found by equation (4.18) from the total number of jobs we can get the number of interfering high criticality jobs in low criticality mode, which gives us the following equation:

$$NL(i,\ s) = N(i) - NH(i,\ s) \tag{4.19}$$

This equation finds the number of high criticality jobs that are released in low criticality mode for a given point s.

To find which value of s gives the largest interfering workload we calculate equations (4.17), (4.19) and (4.18) for every value of s and add each value of s that gave a different number distribution to the set S. We then use these numbers to create the following sets of interfering jobs:

$$HJN_{k,s} = \bigcup_{\tau_i \in hpL(k)} \{J_i^1, J_i^2, ...J_i^{NLC(i,\ s)}\},\ s \in S \tag{4.20}$$

$$HJL_{k,s} = \bigcup_{\tau_i \in hpH(k)} \{J_i^1, J_i^2, ...J_i^{NL(i,\ s)}\},\ s \in S \tag{4.21}$$

$$HJH_{k,s} = \bigcup_{\tau_i \in hpH(k)} \{J_i^1, J_i^2, ...J_i^{NH(i,\ s)}\},\ s \in S \tag{4.22}$$

From these we can construct the following sets:

$$HJM_{k,s} = HJL_{k,s} \cup HJH_{k,s} \tag{4.23}$$

$$HJ_{k,s} = HJN_{k,s} \cup HJM_{k,s} \tag{4.24}$$

Since $HJL_{k,s}$ and $HJH_{k,s}$ contains jobs from the same tasks and their workload in regards to faults will be handled in the same way, baring different wcet, the sets $HJL_{k,s}$ and $HJH_{k,s}$ are merged into one set $HJM_{k,s}$ with jobs from $HJL_{k,s}$ being identified as $J^p_{i,LO}$ and jobs from $HJH_{k,s}$ with $J^p_{i,HI}$.

We will again calculate $W^c(HJ_{k,s})$, i.e. the workload of the interfering jobs suffering from c faults. The workload of the low criticality tasks, $W^0(HJN_{k,s})$, is calculated using equation (4.7) in the same way as in the low criticality analysis.

The procedure to calculate the workload of the high criticality jobs is largely the same as the low criticality analysis, just with the addition of handling high criticality jobs released in both modes. Calculating the workload for $W^c(HJM_{k,s})$ is done through recursion using equation (4.25) as a basic element.

$$W^f(\{J^p_{i,\lambda}\}) = C^f_{i,\lambda},\ \lambda \in \{LO,\ HI\} \tag{4.25}$$

This equation just gives the workload of a single job, where $J^p_{i,\lambda} \in HJM_{k,s}$, and assuming that f faults affect that job. Then the workload $W^c(HJM_{k,s})$ can be calculated recursively as follows:

$$W^c(HJM_{k,s}) = \max_{f=0}^{c} \left\{ W^f(\{J^p_{i,\lambda}\}) + W^{c-f}(\beta) \right\} \tag{4.26}$$

where $\beta = (HJM_{k,s} - \{j^p_{i,\lambda}\},\ J^p_{i,\lambda} \in HJM_{k,s})$. This equation also finds the fault distribution in $HJM_{k,s}$ that gives the maximal workload following the same reasoning as for equation (4.9).

Then we have that the workload $W^c(HJ_{k,s})$ is given by:

$$W^c(HJ_{k,s}) = \max_{s \in S} \left\{ W^0(HJN_{k,s}) + W^c(HJM_{k,s}) \right\} \tag{4.27}$$

This equation goes through the different sets of $HJ_{k,s}$, given by different s values, to find the one that has the largest workload.

Next up the CPU time requirement for job J_k and its active backups is $\widehat{M} \cdot S_{k,\widehat{M}}$ where:

$$S_{k,\widehat{M}} = \max_{z=0}^{h_k} \left\{ E^z_{k,HI} + \frac{\sum_{b=0}^{z-1} E^b_{k,HI}}{\widehat{M}} \right\} \tag{4.28}$$

The only difference to the low criticality analysis is that this equation assumes that J_k uses the high wcet, since we can not be sure what portion of the execution was done before s. This assumption also holds for J_k's passive backups.

Next we calculate the time needed to complete the execution of the previously mentioned workloads, following the same reasoning as for the low criticality mode.

Once we have that time we add in the time needed to complete the passive backups of J_k given by $\widehat{C}^f_{k,HI}$ to produce equation $\left\lceil \frac{W^c(HJ_{k,s})+\widehat{M}\cdot S_{k,\widehat{M}}}{\widehat{M}} \right\rceil + \widehat{C}^f_{k,HI}$. From this we can construct the high criticality mode schedulability condition as seen in equation (4.29).

$$R^{HI}_k = \max_{c=0}^{e_k} \left\{ \left\lceil \frac{W^c(HJ_{k,s})}{\widehat{M}} + S_{k,\widehat{M}} \right\rceil + \widehat{C}^{(e_k-c)}_{k,HI} \right\} \leq D_k \tag{4.29}$$

Following the same reasoning as for equation (4.15), equation (4.29) also finds the fault distribution between J_k and the jobs in $HJ_{k,s}$ that produce the longest response time R^{HI}_k. A high criticality task that passed the condition set in equation (4.29) is deemed to be schedulable under the proposed scheduling algorithm.

4.4 Priority and Active Backup Assignment

Since the proposed scheduling algorithm uses fixed priorities, task-sets that are to be scheduled on this algorithm needs to have their priorities assigned and the schedulability of the task-sets must be tested to ensure that all deadlines are met. Priority assignment will assign each task in a task-set a distinct priority, so a task-set with n tasks will have n different priority levels, where higher priority number means a lower real-time priority. In a task-set with n tasks there are $n!$ different priority assignments, and going through each combination in an attempt to find a priority assignment that is schedulable is clearly unfeasible for any but the smallest task-sets. Audsley's OPA approach [3] is a way to reduce the number of different priority assignments that need to be checked and also terminate the priority assignment early if no valid, i.e. schedulable, priority assignment exists. By following Audsley's OPA approach the number of different priority assignments for a task-set with n tasks can be reduced to the following: $n + (n-1) + ... + 1$ instead of $n!$. The following procedure follows Audsley's OPA approach and will try to assign priorities to a task-set:

1. Start with all tasks having no priority and belong to set $P_\emptyset$.
2. Pick a task $\tau_i \in P_\emptyset$ from $P_\emptyset$ that has not been picked for this priority and assign it the lowest available priority.
3. Perform the schedulability test, if it fails re-add τ_i to $P_\emptyset$ and go back to step 2.
4. If the test is successful, go back to step 2 but don't add τ_i back to $P_\emptyset$.
5. If no task can be successfully assigned a priority level then the task-set is not schedulable.

Since the $\widehat{M}$ highest priority tasks will be able to preempt at least one lower priority task, it is assumed that the $\widehat{M}$ highest priority tasks will not suffer any interference

from each other, effectively the $\widehat{M}$ highest priority tasks share the highest priority.

The use of active backups brings both positive and negative impacts on schedulability, the negative is that it introduces more work that needs to be done since active backups are always executed. But in return, the active backups can execute in parallel which can improve schedulability, and take advantage of multicore processors since passive backups must execute in sequence.

This paper proposes two main policies to assign active backups, *Minimal* and *Joint*. The latter of which comes in two variants *Joint-Min* and *Joint-Max*. For comparison's sake, the following basic policies are also included: *None*, and *Random*. The specifics of the policies used in this paper are as follows:

1) *None*: No active backups are assigned.

2) *Random*: The number of active backups is assigned randomly, that is h_i for each high criticality task is assigned a random number from $(0,\ f)$.

3) *Minimal*: Assign active backups to tasks that can't meet their deadline suffering f faults even at the highest priority. For each task τ_i where the following equation holds:

$$R_i^{HI} = \left\lceil S_{i,\widehat{M}} \right\rceil + \widehat{C}_{i,HI}^{f} > D_i \tag{4.30}$$

h_i is incremented until equation (4.30) is no longer fulfilled.

4) *Joint*: Active backups are assigned in conjunction with the priority assignment. When a high criticality task τ_j is not schedulable at a particular priority level instead of picking a new task, increment h_j by 1 and repeat the test, continue to do this until the test is successful or $h_j > f$. If the test is successful then an additional check to see if the previously assigned tasks are still schedulable is done, since the workload of the additional backups was not there when those tasks were scheduled. If this check fails or if $h_j > f$ then τ_j is deemed not schedulable at this level and h_j is reset to its previous value and the priority assignment process carries on as normal. Since the *Minimal* policy gives us the minimum number of active backups required to make the task-set theoretically schedulable this will be the starting point for the *Joint-Min* policy. To mitigate the issue of post-assignment interference increase another variant *Joint-Max* is presented. It assumes that all high criticality tasks that have not been assigned a priority, will have the maximum number, f, of active backups, when the tasks are assigned they will get the "correct" number of active backups. So, in summary, the *Joint* policy has the following two variants which govern the starting amount of active backups:

4a) *Joint-Min*: The starting number of active backups are set according to the *Minimal* policy.
4b) *Joint-Max*: The starting number of active backups are to f.

5

Simulated Testing

The scheduler and schedulability test will be simulated and tested with randomly generated task-sets. The main property that will be tested is acceptance ratio, i.e. the proportion of task sets that pass the schedulability test. Then a sensitivity analysis will be performed, where the effects of different values for system properties, like with varying utilization levels, the total number of tasks, the ratio of high criticality tasks, the number of failures and the effects of the different policies for active backup assignment. Let n be the number of tasks in a task-set, and f be the maximal amount of faults that can occur.

5.1 Task-set Generation

The task-sets are generated accordingly: First, the task-set is given a utilization, U, between (0, 1), this value is then scaled with the number of processors, M. Then U is passed to the UUnifastDiscard algorithm [13] together with the number of tasks, n, the algorithm then generates utilizations for each task. Next the tasks criticality will be generated by giving the wanted ratio of high criticality tasks, CR, to a simple algorithm that for each task generates a random number between (0, 1) and if this number is smaller than CR then the task is given high criticality otherwise it is given low criticality. This algorithm also ensures that the actual criticality ratio is close to the wanted CR, by checking that the number of generated high criticality tasks equals the theoretical number of tasks derived from the CR. If the wanted number of high criticality tasks is not an integer then the number of generated high criticality tasks is accepted if the number is either the $floor$ or *ceiling* of the wanted number, for example in a task-set with 5 tasks and $CR = 30\%$ the wanted number is $5 \cdot 0.3 = 1.5$, so generating either 1 or 2 high criticality tasks are accepted. Next, the tasks have to be given a period, by randomly selecting a number from (10, 10000). Now the wcet of each task is given by multiplying the tasks period with its utilization, this wcet is considered to be the low criticality one. The high criticality tasks also require a high wcet, this is achieved by adding some extra time to the low wcet, this *extra* time is given by picking a random number from (0, U_{extra}) where at the start U_{extra} is the sum of the utilization of the low criticality tasks. This number is then bounded to ensure that the task utilization does not exceed 1, U_{extra} is then

reduced by this number for future high criticality tasks. This procedure ensures that the utilization in high criticality mode does not exceed the utilization level specified in the generation. In summary, the task-set generation takes the following parameters: the total utilization scaled by the number of processors, the number of tasks in the task-set, and finally the criticality ratio.

5.2 Test Results

For each utilization level, 1000 task-sets are generated and each task-set is then tested by the schedulability test, once for each policy, then the acceptance ratio is given by dividing the number of successfully scheduled task-sets with the total number of task-sets. The utilization is varied from 0 to 1 in steps of 0.025. The results presented in this section are for the following system parameters: $M = 4$ processors and a CR of 30 % when testing the fault-tolerance. For these tests, the task's deadline equals their period, and backup tasks are copies of the primary.

5.2.1 Base Performance

The purpose of this round of tests is to see how the schedulability test performs under basic conditions, that is no fault-tolerance. Figure 5.1 shows the base performance of the proposed algorithm, without any high criticality tasks, i.e. CR of 0. We see that the acceptance ratio generally improves with an increased number of tasks in the task-sets, the exception to this is the task-set with 5 tasks, it has better schedulability than the other task-sets at a utilization of around 0.65 - 0.7, this is due to the fact that the number of tasks is very close to the number of cores, in this case only one task actually suffers interference.

Figure 5.2 shows the effect that different criticality ratios have on schedulability. We can see that the results seem counter-intuitive, the acceptance ratio for the larger criticality ratios (70 - 90 %) is in general closer to the case with $CR = 0\%$ and better than the smaller CRs. This is a side effect of how the high wcet is generated, with a low criticality ratio the "spare" utilization available for the *extra* time is greater, so the probability of more than $\widehat{M}$ high criticality task having a utilization of 1 is great, which means that the task-set is not schedulable. When looking at the effects of different task-sets sizes we see that as the size increases the acceptance ratios for the CRs become more "ordered", i.e. the higher the CR the better the schedulability. A reason for this is again the task-set generation, with larger task-sets the randomness is more spread out lowering the probability of having utilization heavy tasks.

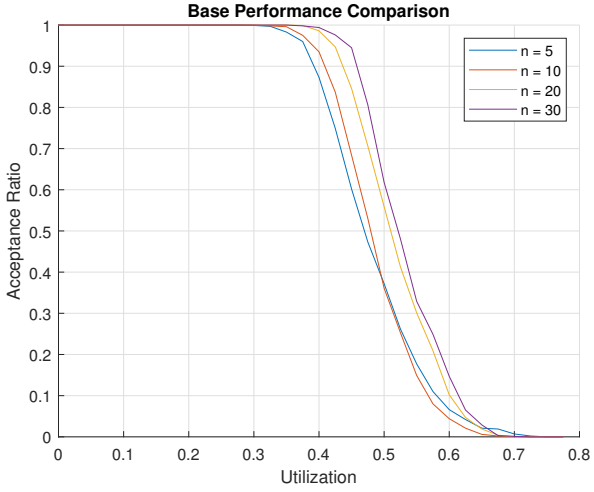
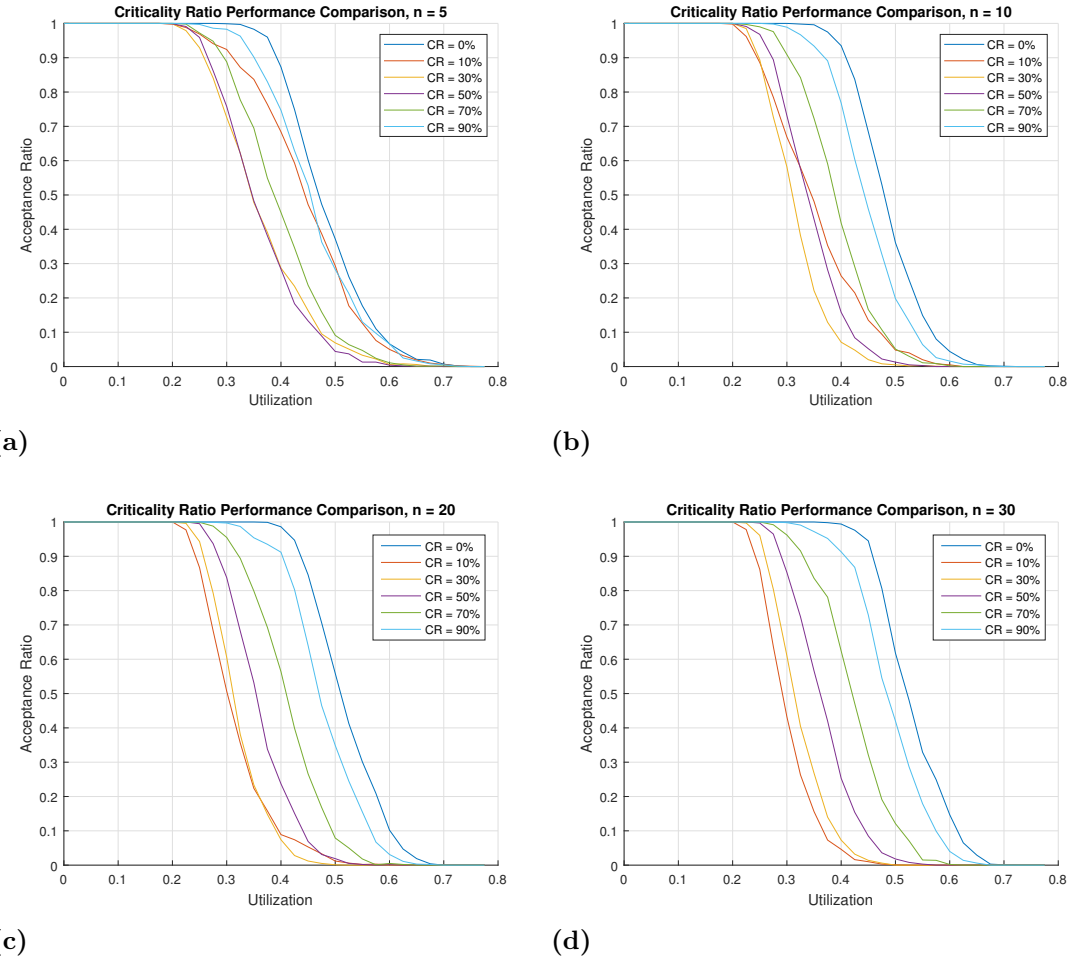


Figure 5.1: Acceptance ratios of the schedulability test in basic setting, no mixed-criticality and no fault-tolerance, for four differently sized task-sets.



(a) **(b)**

(c) **(d)**

Figure 5.2: The schedulability tests acceptance ratio for different criticality ratios.

5.2.2 Fault-Tolerance Performance

The purpose of these tests is to see how the fault-tolerance affects the schedulability. The left side of figure 5.3 shows the schedulability of task-sets with 5 tasks suffering from an increased number of faults. For these task-sets, we see that all policies that use active backups have better schedulability than the *None*-policy and that the main policies proposed (*Minimal* & *Joint*) are better than the *Random* policy. This is not continued for larger task-sets, with 5 tasks running on 4 cores there is only 1 task that can be interfered with, so this size of task-sets are more tolerating of increases in workload from the active backups. Among the proposed policies the *Joint-Min* is the best by a very small margin that increases slightly with the number of faults.

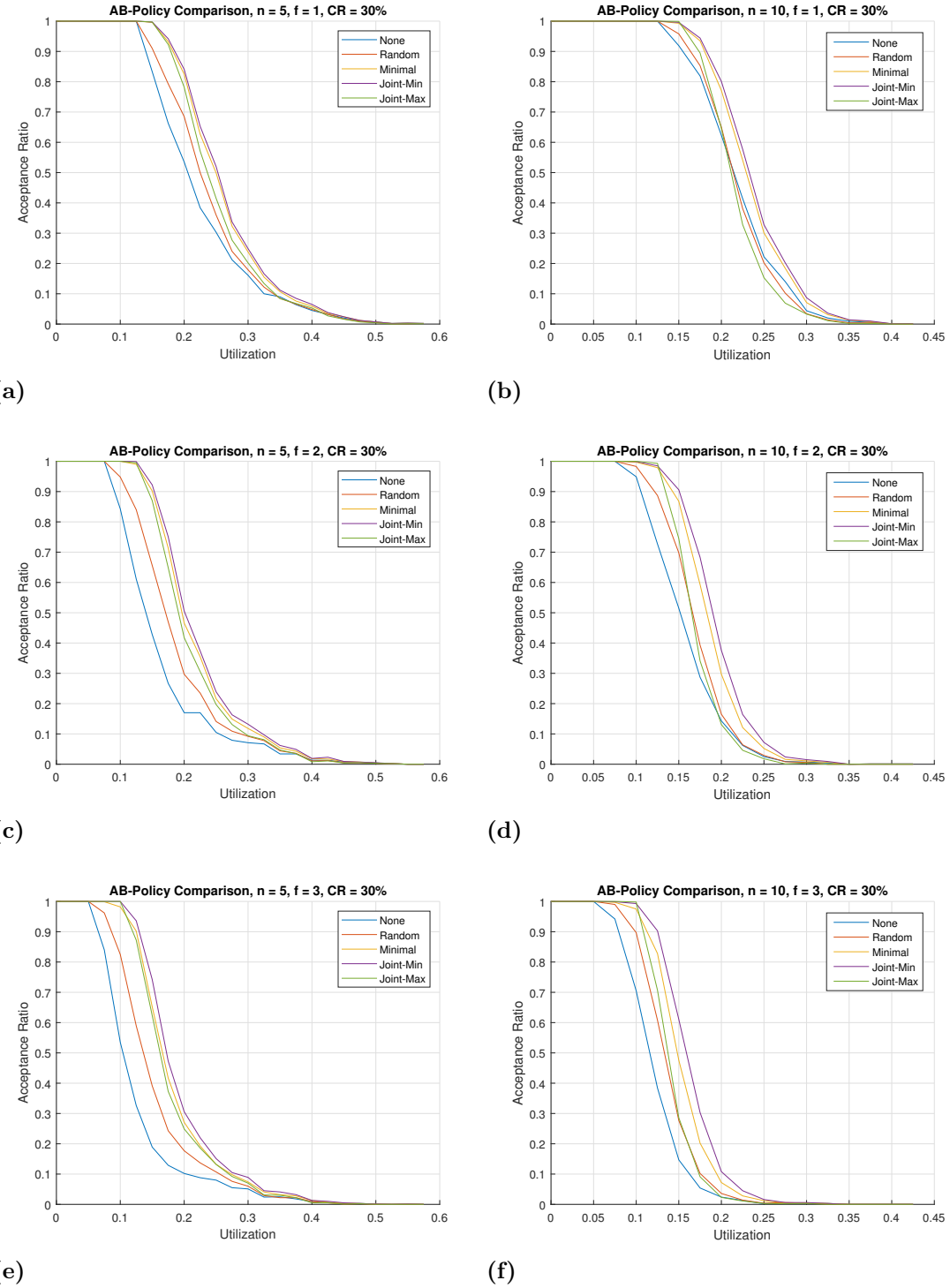
The right side of figure 5.3 shows the schedulability of task-sets with 10 tasks suffering from an increased number of faults. Here we start to see some of the negatives of active backups, with 1 or 2 faults (5.3 (**b**) & (**d**)) we see that that the *Joint-Max* and *Random* policies have worse schedulability than the *None* policy for the higher utilization levels. This is because we have an increased number of high criticality tasks, in comparison to the $n = 5$ task-sets, that $Joint - Max$ gives the maximal number of active backups to, leading to a big increase in workload that is detrimental to schedulability. With 3 faults, (5.3 (**f**)), the relative performance of the *Joint-Max* and *Random* policies improves as active backups becomes needed to handle faults in a timely manner. The *Joint-Min* policy is the best and its lead compared to the next best policy, the *Minimal*, increases as the number of faults increase.

For larger task-sets figure 5.4 shows us their schedulability. The left side shows task-sets with 20 tasks, and here we can see the trend that began in the 10 task task-sets, namely that assigning to many active backups or assigning them to the wrong tasks have a negative impact on schedulability. We see that the *Joint-Max* and *Random* policies are worse than the None policy after certain utilization levels, generally, the None policy is the worst at lower utilization levels while Joint-Max is worst for higher utilization levels. The *Random* policy is worse than the *None* policy for task-sets suffering 1 fault (5.4 (a)) but is largely comparable with the *None* policy for the larger fault numbers (5.4 (c) & (e)). The *Joint-Min* policy is also the best policy for task-sets with 20 tasks, with the *Minimal* following closely behind. Once again the gap between the *Joint-Min* and *Minimal* policies increase with the number of faults.

The final tests were done with task-sets with 30 tasks and the results can be seen on the right side of figure 5.4. These curves are very similar to the one for $n = 20$ task-sets, again the *Joint-Max* and *Random* policies are worse than the *None* policy for larger utilizations, they compare slightly better for higher fault numbers. Once again the *Joint-Min* is the better policy with the *Minimal* one following tightly, but falling behind with the increasing number of faults.

In contrast with the base performance (figure 5.1) where the schedulability improved with task-set size, the inverse is true when fault-tolerance is used. The main reason

is the increase in the number of high criticality tasks, many of whose utilization can be 1 for reasons previously discussed, this high utilization obviously makes it hard to handle any faults as there are no margins in the system.

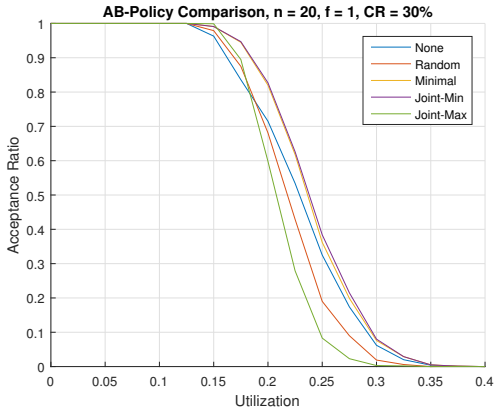


(a) **(b)**

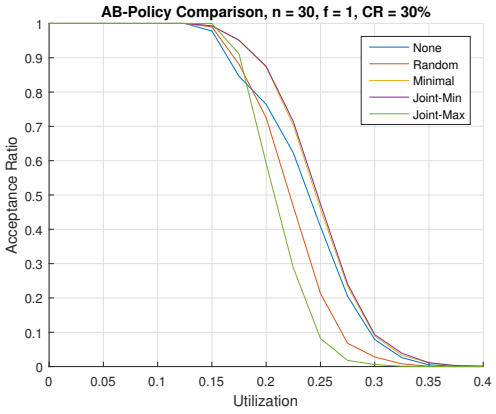
(c) **(d)**

(e) **(f)**

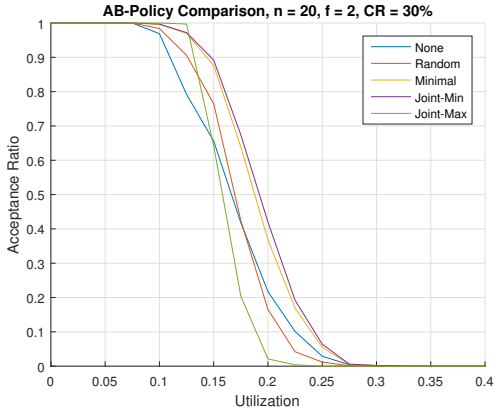
Figure 5.3: Comparison of the different active backup assignment policies. Left side shows task-sets with 5 tasks, the right task-sets with 10 tasks.



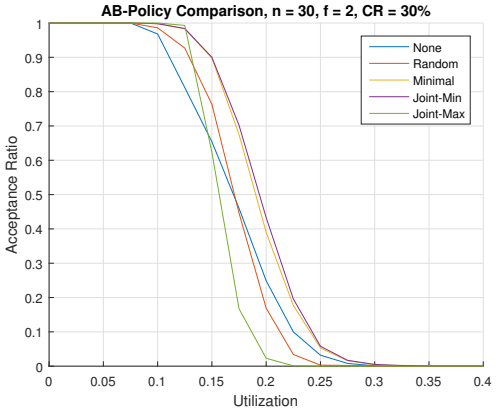
(a)



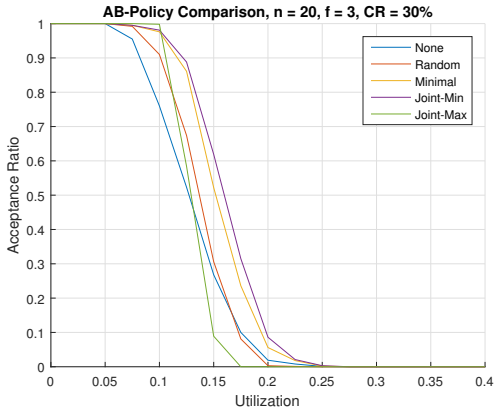
(b)



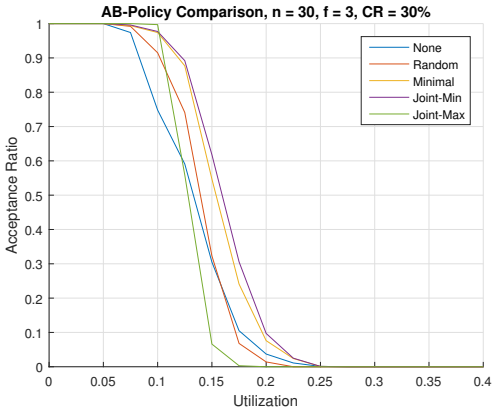
(c)



(d)



(e)



(f)

Figure 5.4: Comparison of the different active backup assignment policies. Left side shows task-sets with 20 tasks, the right task-sets with 30 tasks.

5.3 Summary of Results

We saw that the schedulability of the algorithm gets worse with larger task-sets, and in regards to the active backup assignment policies the tests found that:

- *None*: generally the worst policy for task-sets with less than 20 tasks, but having slightly better schedulability than the *Joint-Max* and *Random* policies for larger task-sets.
- *Random*: Better then the *None* policy for small task-sets ($n <= 10$), but negatively impacts schedulability for larger task-sets.
- *Minimal*: Is the second-best policy, always slightly behind the Joint-Min policy.
- *Joint-Min*: The policy that provides the best schedulability, and its performance relative to the *Minimal* policy improves as the number of faults increase.
- *Joint-Max*: Performs comparably to Minimal and Joint-Min for small task-sets ($n = 5$), but its performance worsens as the task-sets increase in size.

6

Conclusions

This thesis presents a new real-time global fixed-priority scheduling algorithm that integrates mixed-criticality with fault-tolerance. Using mixed-criticality in real-time scheduling is a way to mix safety critical and none safety-critical tasks on the same platform while ensuring that the *temporal* correctness of the safety-critical tasks is not compromised. The thesis chose to model mixed-criticality with 2 criticality level, low and high, non-safety critical tasks were assigned the low criticality level and the safety critical task the high criticality level. The high criticality tasks were also assumed to have 2 separate wcet, one that was less pessimistic and one that more pessimistic to reduce the probability that the estimated wcet is smaller than the actual wcet. Having the less pessimistic wcet allows for more tasks to be scheduled on the same platform, but in case this estimate is exceeded during runtime, the scheduler is designed to drop all low criticality tasks to allow more computational resources to be devoted to the high criticality tasks to ensure that they don't miss any deadlines.

Adding fault-tolerance to the safety critical/high criticality tasks enhances the *functional* correctness of these tasks, thus adding more robustness to these tasks to ensure that they operate correctly. The proposed scheduling algorithm handles fault-tolerance through the execution of backup tasks, that are either active or passive. The algorithm can handle transient software and hardware faults and permanent software faults in case some, or all, of the backup tasks, are diverse implementations rather than copies of the primary. The algorithm can also handle permanent core failures thanks in part to the use of global scheduling that allows core failures to be modeled as a task failure. Backup tasks can then be scheduled on the remaining cores since they have are not bound to specific cores, core failures during runtime means that the scheduler just has a reduced number of cores to assign tasks to, in theory, the scheduling algorithm can keep a task-set scheduled and functional even when reduced to a single working core.

The proposed scheduling algorithm is designed for a sporadic task model with explicit deadlines, having low and high criticality tasks, with the high criticality ones having additional backup tasks, these can be diverse implementations or copies of the primary. The model also allows certain backups to be designated as active, that is they are always released alongside the primary. This potentially allows handling faults affecting a task more quickly, the downside is that they are released even

when no faults have been detected leading to an increase in workload. Backup tasks not designated as active will instead be passive, i.e. they release only when a fault has been detected in a task and no other version of that task is currently executing/waiting. This method conserves workload but may, in turn, take a longer time to produce a correct result.

The fault model designed for the proposed algorithm specifies that the number of faults that can affect the system is limited to a specified number for a time interval that is equal to the largest deadline in the task-set. Inside this interval, the frequency at which the faults occur is not considered, the faulty can happen in quick succession or more slowly. The model assumes that faults that affect the result of tasks are detected, with a perfect success rate, at the end of a task's execution. In addition, the model also specifies the number of core failures that can occur, no consideration is taken to specifically when these faults occur, the worst case is that the cores fail as soon as the system is started, so that is the assumption that the model will make.

These Models are used to derive a schedulability test for the scheduling algorithm, this test guarantees that a task-set will meet all deadlines in the presence of a certain number of task faults and core failures. Finding a valid priority assignment for the fixed priorities is a computationally hard problem, therefore the schedulability test was designed to be OPA compatible, which means that Audsley's OPA approach can be used to assign the priorities in an efficient way.

To make use of the different properties between active and passive backups this thesis also designed a number of different policies that govern how many active backups should be assigned to the tasks with the aim to improve schedulability. Simulated tests showed that mixed-criticality and the fault-tolerance lead to noticeable drops in schedulability. The tests showed that having some form of active backups improved schedulability, but that it became more important to assign active backups carefully the larger the task-sets become. The *Joint-Min* active-backup policy was shown to be the policy that could schedule the most task-sets, with its performance improvement relative to the other policies as the number of faults grew.

6.1 Future Work

Following in the spirit of robustness and fault-tolerance it would be interesting to extend the mixed-criticality analysis to include the possibility to return to low criticality mode from high criticality mode when certain conditions are met. This could in effect allow the scheduler to switch to high criticality mode for a period of time, when there are a large number of faults experienced, and switch back to low criticality mode if the number of faults decreases. This would make the system more robust and able to function in a fluctuation environment.

Another interesting approach would be to do a schedulability analysis for a more advanced fault model. For example, instead of having all tasks be affected equally

by faults, some tasks might be more susceptible to faults while others are more robust. Having this differentiation would allow for a more precise assignment of active backups, for example, it would be sensible to assign active backups to tasks that regularly suffer faults while leaving tasks that rarely experience faults with just passive backups.

Having guarantees that a task-set can handle f number of faults in a time interval can leave much empty time in the schedule. Such as in the case where no faults are experienced, or the faults affected tasks that were not as workload intensive as the worst case estimation performed by the schedulability test. In these cases, it might be beneficial to extend the fault-tolerance to low criticality tasks in an "opportunistic" way, where they are only allowed backups in case the schedule is empty.

Bibliography

[1] Z. Al-bayati, B. H. Meyer, and H. Zeng. Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 57–62, Sept 2016.

[2] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, pages 193–202, Dec 2001.

[3] N.C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39 – 44, 2001.

[4] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 119–128, Dec 2007.

[5] S. Baruah and Z. Guo. Mixed-criticality scheduling upon varying-speed processors. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 68–77, Dec 2013.

[6] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 13–22, April 2010.

[7] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, Nov 2011.

[8] S. K. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970, Jul 2003.

[9] I. Bate, A. Burns, and R. I. Davis. A bailout protocol for mixed criticality systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 259–268, July 2015.

[10] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Par-*

allel and Distributed Systems, 20(4):553–566, April 2009.

[11] A. Burns and R. I. Davis. Adaptive mixed criticality scheduling with deferred preemption. In *2014 IEEE Real-Time Systems Symposium*, pages 21–30, Dec 2014.

[12] J. J. Chen, C. Y. Yang, T. W. Kuo, and S. Y. Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 249–258, April 2007.

[13] R. I. Davis and A. Burns. Robust priority assignment for fixed priority real-time systems. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 3–14, Dec 2007.

[14] R. I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *2009 30th IEEE Real-Time Systems Symposium*, pages 398–409, Dec 2009.

[15] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *2009 30th IEEE Real-Time Systems Symposium*, pages 387–397, Dec 2009.

[16] P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[17] International Organization for Standardization (ISO). *Road vehicles – Functional safety.* ISO 26262-2011. International Organization for Standardization (ISO), 11 2011.

[18] K. I. Krishna and C. Mani. *Fault Tolerant Systems.* Elsevier, 2007.

[19] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[20] G. Liu, Y. Lu, and S. Wang. An efficient fault recovery algorithm in multiprocessor mixed-criticality systems. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 2006–2013, Nov 2013.

[21] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1864–1871, June 2010.

[22] R. M. Pathan. Schedulability analysis of mixed-criticality systems on multi-

processors. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 309–320, July 2012.

[23] R. M. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4):509–547, Jul 2014.

[24] R. M. Pathan. Real-time scheduling algorithm for safety-critical systems on faulty multicore environments. *Real-Time Systems*, 53(1):45–81, Jan 2017.

[25] R. M. Pathan and J. Jonsson. Ftgs: Fault-tolerant fixed-priority scheduling on multiprocessors. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1164–1175, Nov 2011.

[26] M. Bertogna S. Baruah, G. Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[27] M. Salehi, M. Khavari Tavana, S. Rehman, M. Shafique, A. Ejlali, and J. Henkel. Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(7):2426–2437, July 2016.

[28] A. Thekkilakattil, R. Dobrin, and S. Punnekkat. Mixed criticality scheduling in fault-tolerant distributed real-time systems. In *2014 International Conference on Embedded Systems (ICES)*, pages 92–97, July 2014.