CHALMERS


# Partitioned Fixed-Priority Multiprocessor Scheduling for Mixed-Criticality Real-Time Systems

*Master of Science Thesis in Programme Computer Systems and Networks*

Aljoscha Lautenbach

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, September 2013

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Partitioned Fixed-Priority Multiprocessor Scheduling for Mixed-Criticality Real-Time Systems

Aljoscha Lautenbach

©Aljoscha Lautenbach, September 2013.

Examiner: Jan Jonsson
Supervisor: Risat Pathan

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2013

## Abstract


The scheduling of real-time systems has been the subject of research for many years since it has many implications for safety-critical embedded real-time systems. Within that field, there have been two recent developments which are the subject of this thesis. The first is the development of new scheduling theory for mixed-criticality systems, i.e. systems in which functions of differing importance are scheduled on the same processor. The second is the increasing production use of multiprocessor systems, even in the world of embedded real-time systems.

The purpose of this work is twofold. First a new schedulability test for fixed-priority mixed-criticality uniprocessor systems will be presented and evaluated. Secondly, that schedulability test will be used in an evaluation of different heuristics for partitioned multiprocessor scheduling of mixed-criticality systems.

In order to evaluate the performance of the new schedulability test and the different heuristics in terms of schedulability, a number of experiments were performed. For this purpose random task sets were generated, and each such task set was tested for schedulability.

It was found that the new uniprocessor schedulability test outperforms the previously known approaches for fixed-priority mixed-criticality task sets on pre-emptively scheduled uniprocessor systems. In terms of the heuristics that were evaluated for partitioning, it was determined that the combination of a slack-monotonic initial ordering with a best-fit allocation algorithm and deadline monotonic priority ordering yields the best schedulability. It seems surprising that using a deadline monotonic priority ordering outperforms Audsley's priority ordering approach. Furthermore, it was shown that a utilisation-based worst-fit task allocation algorithm is not a good heuristic for the kind of systems under assessment. Finally, the presented uniprocessor schedulability test seems to scale well with the number of processors, if the number of tasks per taskset is sufficiently high.

# Acknowledgements

I would especially like to thank my supervisor Risat Pathan for all the interesting discussions and for his patience, advise and support. Furthermore, I am grateful to my examiner Jan Jonsson for his support, and for sparking my interest in this topic through his real-time systems courses. I would also like to thank my family and friends for their general support, and especially my sister for regularly answering my questions regarding LaTeX issues.

Aljoscha Lautenbach, Göteborg, October 30, 2013

# Contents

**1 Introduction** **1**

**2 Background** **3**
2.1 Real-Time Systems . . . 3
2.2 Scheduling . . . 4
2.2.1 Schedulability Analysis . . . 5
2.2.2 Response Time Analysis . . . 7
2.3 Multiprocessor Scheduling . . . 8
2.3.1 Partitioned Scheduling . . . 8
2.4 Mixed-Criticality Systems . . . 9

**3 Related Work** **11**

**4 System Model and Notation** **16**

**5 Uniprocessor Schedulability Test - IAMC** **18**
5.1 Calculating $I_L(s)$ . . . 20
5.2 Calculating $I_H(s)$ . . . 20
5.3 Summary of IAMC . . . 25

**6 Evaluation of IAMC for Uniprocessors** **27**
6.1 Experimental Setup . . . 27
6.1.1 Taskset Generation . . . 27
6.1.2 Schedulability Tests . . . 29
6.2 Experiments . . . 29

**7 Evaluation of IAMC for Multiprocessors** **36**
7.1 Heuristics . . . 36
7.1.1 Processor Selection Heuristics . . . 36

7.1.2 Initial Orderings . . . . . . . . . . . . . . . . . . . . . . . . 37
7.2 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
7.3 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
7.3.1 Comparing The Heuristics . . . . . . . . . . . . . . . . . . . . 39
7.3.2 Scalability of IAMC-DM-SM-BF . . . . . . . . . . . . . . . . . 46

**8 Conclusions** **52**

**Bibliography** **56**

# List of abbreviations

**AMC** Adaptive Mixed-Criticality

**ASIL(s)** automotive safety integrity level(s)

**BF** best-fit

**COP** Compress-on-Overload Packing

**CSM** criticality slack monotonic

**CU** criticality utilisation

**DC** decreasing criticality

**DM** deadline-monotonic

**DU** decreasing utilisation

**ECU(s)** electronic control unit(s)

**EDF** earliest deadline first

**FF** first-fit

**FP** fixed-priority

**IAMC** Improved AMC

**LCT** Latest Completion Time

**MC** mixed-criticality

**OPA** optimal priority ordering

**OCBP** Own Criticality-Based Priorities

**PLRS** Priority List Reuse Scheduling

**RM** rate-monotonic

**RTA** response-time analysis

**SIL(s)** safety integrity level(s)

**SM** slack monotonic

**WCET** worst-case execution time

**WCRT** worst-case response time

**WF** worst-fit

**ZSRM** Zero-Slack Rate-Monotonic

# 1

# Introduction

During the last two decades, embedded systems have become highly pervasive in our society. There are few devices which operate without embedded processors, and many of their applications have real-time requirements. Modern cars for instance consist of somewhere between 50 and 100 electronic control unit(s) (ECU(s)) which execute a number of applications, ranging from safety-critical control systems to comfort systems. The timely reaction of those applications to user and sensor input is of paramount importance so that traffic safety can be guaranteed, e.g. when braking.

A real-time system comprises applications which have stringent timing requirements, e.g. restrictions on their finishing time (execution deadline). Most of these applications consists of a collection of small tasks that run concurrently on a processor. Scheduling programs in real-time systems has been an active area of research for a long time.

In order to reason about the design of these systems, their timing properties are abstracted into models. With the help of these models extensive theory for guaranteeing timeliness has been developed. A common model for real-time system is as a set of recurrent tasks which have a fixed frequency of execution and a deadline by which the execution must be finished. This type of application is often found in control systems.

Many real-time systems consist of applications of different criticality. Revisiting the example of ECU(s) in a car, some of them run safety-critical applications whereas others may only be running comfort functions. Currently, applications of different criticality are designed to run on different processors so that safety can be guaranteed. In the future it may be more cost-effective to be able to integrate safety-critical and non-safety-critical applications on a single processor. However, this integration may raise questions to the safety of such a system. In order to be

able to guarantee safety, a complete theory is necessary. Systems that integrate applications of differing criticality on a single processor are also known as mixed-criticality systems and research on this topic has recently gained traction.

Due to the stagnation of uniprocessor clock speeds, multiprocessor systems continue to spread in all areas. This recent trend also has strong repercussions in the real-time systems community. The scheduling issues for uniprocessor systems are generally well understood, but the theory for multiprocessor scheduling is still under very active development.

In this report we present a new response-time analysis based uniprocessor schedulability test for mixed-criticality systems which uses work-load analysis, an idea which is customarily applied to multiprocessor scheduling. We will show that for a specific run-time model, the new test performs better than all previously proposed uniprocessor tests for mixed-criticality systems. We will also investigate how it performs in partitioned multiprocessor environments compared to other tests, and we will investigate how different parameters influence the results.

The remaining structure of this report is as follows. In chapter 2 the necessary background and definitions for the rest of the thesis are developed. Chapter 3 gives an overview of the existing work regarding the scheduling of mixed-criticality systems. The system model that is used in the remainder of the report is described in chapter 4. The new uniprocessor test for mixed-criticality systems is presented in the 5th chapter. Subsequently the performance of the proposed test is evaluated for uniprocessor and for partitioned multiprocessor systems in chapters 6 and 7 respectively. Finally, our conclusions are presented in chapter 8.

# 2

# Background

THIS chapter will give a brief introduction to the field of real-time systems, especially to the topics which are relevant in the context of this thesis. We will also define a number of basic terms here.

## 2.1 Real-Time Systems

In general, *real-time systems* are systems in which the timing of a computation is important for the correctness of the system. A distinction can be made between *hard real-time* and *soft real-time* requirements. Systems with hard real-time requirements can not tolerate even a single violation of their timing requirements, whereas for soft real-time systems there is a gradual decline in utility for each occurring timing fault. This thesis is only concerned with hard real-time systems.

For the purpose of this thesis, a real-time system is modelled as a number of independent, recurrent tasks which need to be scheduled on one or more processors according to their timing properties.

A task $\tau_i$ has the following timing properties:

- **Release time $r_i$:** The time at which this task was first released.
- **Relative deadline $D_i$:** The time by which this task needs to be done.
- **Period $T_i$:** The (minimum) inter-arrival time between releases.
- **Worst-case execution time $C_i$:** The uninterrupted/undisturbed execution time of this task in the worst case.

These can also be written as a tuple ($r_i$, $C_i$, $T_i$, $D_i$). A *job* of a task is an *instance* of the task. A task can spawn an infinite number of jobs.

Another important property is the *task utilisation*, i.e. the ratio of the worst-case execution time (WCET) and the period ($U_i = C_i/T_i$). The utilisation provides a very simple scheduling check for any processor. *Processor utilisation* can be defined as the sum of the task utilisations of the tasks scheduled on that processor. If the processor utilisation is bigger than 1, the tasks can not all be scheduled successfully on that processor.

The periodicity of a task depends on the relation of the period and the release time. A *strictly periodic* task is released exactly every $T$ time units, i.e. $T$ specifies the exact inter-arrival time. A *sporadic* task has an inter-arrival time of at least $T$ time units, but the release may happen later. Finally, *aperiodic tasks* show no periodicity, they are released randomly. All task systems in this thesis consist of sporadic tasks.

If a task has a deadline which is lower or equal to its period ($D \leq T$), we call it a *constrained deadline* task system. If the deadline is equal to the period ($D = T$), we call it an *implicit deadline* task system. In any other case, we call it an *arbitrary deadline* task system. In this thesis we only consider task systems with constrained deadlines.

Tasks can have interdependencies due to resource constraints, for instance if several tasks need to communicate over a single bus. However, we will only consider independent tasks in this thesis.

A very simple task system with two tasks is depicted in table 2.1. If no arrival time is specified as in this case, it is assumed to be 0 for all tasks.

**Table 2.1:** Example task set

| | $C_i$ | $T_i$ | $D_i$ | $U_i$ |
|---|---|---|---|---|
| $\tau_1$ | 1 | 5 | 3 | 0.2 |
| $\tau_2$ | 2 | 10 | 5 | 0.2 |

## 2.2 Scheduling

The scheduling of real-time systems has been an active area of research for many decades, and a lot of theory has been developed as a result. Process scheduling is also a part of operating systems theory, and the terminology is partly the same.

In *preemptive* systems a running task can be suspended in favour of another task, even though the running task has not yet completed. A *non-preemptive* system is one in which tasks always run to completion before a new task can be

scheduled to run. There are advantages and disadvantages to both models, but we will only consider preemptive systems.

The run-time system consists of a *scheduler* and a *dispatcher*. The dispatcher is responsible for starting ready-to-execute tasks according to the priority determined by the scheduler. The scheduler implements a scheduling algorithm which decides the task execution order. The *scheduling algorithm* generates a schedule for a task system in a specific run-time system.

A *feasible schedule* is one which fulfils all timing constraints of a task system, and a task system is *schedulable* if at least one scheduling algorithm exists which generates a feasible schedule.

Pretty much all run-time systems have a concept of *task priority* which helps the scheduling algorithm to generate a feasible schedule. The task priority is also used by the dispatcher during run-time to determine which task should be executed next.

Task priorities can either be fixed or dynamic. In *fixed-priority (FP)* systems, the task priorities are determined offline and never change at run-time. An example of this is *rate-monotonic (RM)* scheduling, in which priorities are assigned offline according to task periods, i.e. the shorter the period the higher the priority of the task. In systems with *dynamic priorities* on the other hand, priorities can change at run-time based on some run-time properties. An example for this is *earliest deadline first (EDF)* scheduling, in which the task with the next upcoming absolute deadline has the highest priority. The absolute deadline of a job is the release time of the job plus its relative deadline. We will only consider FP systems.

### 2.2.1 Schedulability Analysis

Schedulability analysis is concerned with determining the schedulability of a given task system for a specific run-time scheduler.

It consists of two related sub-problems. First, the problem of finding a priority assignment which makes the system schedulable. Second, in order to actually test the priority assignment for schedulability, a schedulability test is needed. We will consider each problem in turn in the following subsections.

#### Priority Assignment

The *priority assignment* problem can be defined as follows: In a given task system, each task must be assigned a priority so that each task meets its deadline if it is scheduled in a priority-based run-time system.

A popular priority assignment policy for preemptive FP systems is *deadline-monotonic (DM)* scheduling [LW82]. Under this policy, the task priority is inversely proportional to the relative deadline of the task, i.e. the shorter the dead-

line, the higher the priority. For systems with implicit deadlines ($T = D$), DM is the same as RM. These are "classic" priority assignment policies and extensive theory has been developed regarding their applications. The interested reader is referred to any introductory real-time systems book, e.g. [BW09].

Another very popular approach to priority assignment is Audsley's optimal priority ordering (OPA) [Aud91], often simly called "Audsley's approach" in the literature. OPA works with a specific scheduling algorithm which needs to be chosen in advance. Initially the priority assignment algorithm assumes that all tasks have no assigned priority. All tasks without assigned priority will be handled as highest priority tasks by the schedulability test, i.e. they all interfere with tasks which have been assigned a priority. For each unassigned priority level, the OPA algorithm iterates over the entire task set and tries to assign this lowest unassigned priority to a task using the chosen schedulability algorithm. If the lowest priority could be assigned, the algorithm proceeds to the next unassigned priority level and again tries to assign it. If each task can be assigned a priority under the chosen schedulabilit test, the assignment succeeds. If at any point no task can be assigned the lowest unassigned priority level, the algorithm fails and the task set is determined to be unschedulable under the given schedulability test. Our implementation in C is shown in listing 2.1.

```
1  char OPA(int n, Task* tasks, analysis_t test)
2  {
3      int assigned = 0;
4
5      for (int i = 0; i < n; i++)
6          tasks[i].p = UNASSIGNED;
7
8      while(assign_lowest(n, tasks, assigned, test) != FALSE)
9          assigned++;
10
11     if (assigned == n)
12         return TRUE;
13     else
14         return FALSE;
15 }
```

**Listing 2.1:** OPA implementation in C

where assign_lowest is a function which tries to assign the lowest available priority to the current task, with the given schedulability test.

However, OPA is not always applicable to all systems and quite some research has been done regarding the properties which make a system OPA schedulable. We will show later that OPA is indeed applicable for the new uniprocessor test which has been developed in chapter 5.

**Schedulability Testing**

A *schedulability test* determines whether a given task set can be scheduled under a certain priority assignment with a specific scheduling algorithm such that all deadlines of the tasks will be met.

An *exact schedulability* test is both necessary and sufficient, which means that a positive result of the test confirms the schedulability of the task set, and a negative result falsifies it. An exact test is highly desirable, but not always computationally tractable.

A schedulability test which is only *sufficient* confirms the schedulability of a given task set, but a negative result yields no information. If a sufficient test gives a negative result, nothing can be deduced about the schedulability of the task set. Sufficient tests are usually the result of pessimism in the analysis. They reveal less information than an exact test, but they are also easier to derive.

The uniprocessor test which we will present in chapter 5 is a sufficient test.

### 2.2.2 Response Time Analysis

A schedulability test which has long been used is response time analysis[JP86]. The goal of response-time analysis (RTA) is to determine the worst-case response time (WCRT) of a specific task under a given priority assignment, taking the interference of higher priority tasks into account. It is important to note the difference between WCET and WCRT. The WCET is the *uninterrupted* execution time of a task, i.e. completely independent of any other tasks, whereas the WCRT does include interference due to higher priority tasks.

The basic fixed-point response time equation is

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.1}$$

where $hp(i)$ is the set of tasks with higher priority than task $i$, and $\sum_{j \in hp(i)} \lceil R_i/T_j \rceil C_j$ is the interference caused by higher priority tasks. The logical initial value for $R_i$ to solve this fixed point equation is its own WCET $C_i$.

In order to check a task for schedulability according to RTA, the above equation needs to be solved, and the resulting WCRT needs to be checked against the deadline of the task. If the WCRT is bigger than the deadline, the task is not schedulable. This can be repeated for every task in the task set to determine the schedulability of the entire task set. If a single task is not schedulable, the entire task set is not schedulable, assuming the concrete RTA is exact.

One nice property of this test is that it is highly extensible. Baruah et al. [BBD11] adapted RTA to mixed-criticality task sets, and we will adopt a similar approach for our new uniprocessor test.

## 2.3 Multiprocessor Scheduling

In this section we will discuss the additional complexities of multiprocessor scheduling. We will only consider homogeneous multiprocessor environments, i.e. where all processors are of the same type.

There are two main scheduling models for multiprocessors, global scheduling and partitioned scheduling. There are also hybrid models which combine global and partitioned scheduling techniques.

*Global scheduling* entails that there is a single ready-queue for the entire system. When a ready task arrives, it can be scheduled on any processor according to a global scheduler. So tasks can migrate at run-time from processor to processor. This requires a completely new scheduling theory, and a lot of research has been done on this in the last decade or so. It also holds the promise of high processor utilisation.

*Partitioned scheduling* on the other hand means that the tasks will be assigned offline to one specific processor and will never migrate. Each processor has a dedicated ready-queue accordingly. This means that a *task-to-processor* assignment must be found before the system is taken into operation. This task-to-processor assignment is essentially a *bin-packing* problem, and we will use these terms interchangeably. Bin-packing has been shown to be NP-hard.The big advantage of partitioned scheduling is that the well-known uniprocessor theory can be reused on a per-processor basis. The downside is that processor utilisation might not be optimal since ready tasks might have to wait on one processor while another one is idle.

This thesis only deals with partitioned scheduling.

### 2.3.1 Partitioned Scheduling

Partitioned scheduling is about dividing a task set $\Gamma$ such that each task $\tau \in \Gamma$ is assigned to exactly one of $m$ available processors. Let $\Gamma_i$ be the $i$th partition, where $i \in \{1..m\}$. The partitioning needs to be done in such a way that each partition $\Gamma_i$ is schedulable on a single processor.

There are several components in this process that have an impact on schedulability. The whole process can be broken down into three main steps.

1. An **initial ordering** $\Pi$ of the unassigned tasks should be chosen. Any parameter could be chosen to facilitate this ordering. Ordering the task set according to decreasing utilisation has been shown to be beneficial [LGDG03] for example.

2. For each task, the processors need to be tested if the task can be added without making the partition unschedulable under a certain schedulability

test. The processor order depends on some heuristic, since an exhaustive search would be computationally too expensive. If a task fails to be assigned to a processor, the partitioning process fails.

3. When testing whether a task can be added to a specific processor, a schedulability test with a specific priority ordering must be applied.

Since no backtracking is employed, each of these three steps can have a large impact on whether or not a specific task set is found to be schedulable. The complete algorithm is described in pseudo-code in algorithm box 1.

**Algorithm 1** Multiprocessor Partitioning

```
 1: function partitioning(Γ, pack_heuristic, sched_test, prio_ordering)
 2:     assigned_tasks := 0
 3:     while assigned_tasks < n do
 4:         for all τ_i ∈ Γ do
 5:             if (τ_i is UNASSIGNED) then
 6:                 assigned := proc_alloc(τ_i, pack_heuristic, sched_test, prio_ordering)
 7:                 if (assigned is TRUE) then
 8:                     assigned_tasks := assigned_tasks + 1
 9:                     break
10:                 else
11:                     return UNSCHEDULABLE
12:                 end if
13:             end if
14:         end for
15:     end while
16:     return SCHEDULABLE
17: end function
```

## 2.4 Mixed-Criticality Systems

A *mixed-criticality (MC) system* is a system in which tasks of different criticality levels run on the same processor.

Functional safety standards usually define a small number of criticality levels. For example, the general international standard IEC 61508 (“Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems”) defines 4 safety integrity level(s) (SIL(s)). Several adaptations to specialised industries exist, for instance the relatively new standard ISO26262, which defines 4 corresponding automotive safety integrity level(s) (ASIL(s)).

System designers have an interest in scheduling tasks of different criticality on the same processor to reduce costs, and also to be able to better utilise the processor. But many embedded systems, especially safety-critical systems, require certification by certification authorities which usually set different standards from the system designers.

When tasks of different criticality are put on the same processor, the authorities will demand that all tasks will be certified to the level of the task with the highest criticality. This introduces pessimism which diminishes the utility of this approach for the system designer. Much of the research work on MC systems has been done to try to reconcile these two conflicting interests.

One of the most popular approaches to achieve this consolidation is to use different WCET for each criticality level, i.e. to model the WCET as a function of the criticality level.

In MC systems, the task properties listed in section 2.1 are extended by the task criticality level $L_i$. It is important to note that task criticality is distinct from task priority. The criticality of a task gives an indication of the importance of the task functionality, whereas the task priority is a scheduling mechanism to improve system schedulability. There is no necessary correlation between the importance of a task and its priority, although it is of course possible to design it so.

# 3

# Related Work

MIXED-CRITICALITY SYSTEMS have become a popular research subject in recent years, which goes back to the seminal work done by Vestal [Ves07]. He extended the standard response time analysis to preemptive fixed-priority MC systems.

The main difference in the models of standard and mixed-criticality systems is that the WCET $C_i$ of a task $\tau_i$ is an $\mathcal{L}$-dimensional vector, where $\mathcal{L}$ is the number of criticality levels. $C_i(L)$ will denote the WCET of $\tau_i$ at criticality level $L$. Let $L_i$ denote the assigned criticality level of task $\tau_i$. Vestal's least fixed point equation for the response time $R_i$ of task $\tau_i$ is then

$$R_i = C_i(L_i) + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(L_i) \tag{3.1}$$

where $hp(i)$ denotes the set of tasks with higher priority than $\tau_i$. Note the implication that all lower criticality tasks need to be certified up to the highest criticality level, because for each task the response time depends only on its own criticality level $L_i$.

Baruah, Burns and Davis [BBD11] built on Vestal's work and developed a new scheme they call Adaptive Mixed-Criticality (AMC), because it adapts the criticality level of the system at run-time. It was developed for a dual-criticality system, and AMC requires run-time monitoring to work. They also formalised the response-time analysis for AMC. The basic idea is that the system starts out in `LO`-criticality mode, but switches to `HI`-criticality mode if any task runs over its WCET. Once the system is in `HI`-criticality mode, all `LO`-criticality tasks are dropped.

The schedulability verification process according to Baruah et al.'s RTA consists

of three parts:

1. Checking the schedulability of the `LO`-criticality mode
2. Checking the schedulability of the `HI`-criticality mode
3. Checking the schedulability of the moment of the criticality change

Verifying the `LO`- and `HI`-criticality modes is rather straight-forward:

$$R_i^{\mathtt{LO}} = C_i(\mathtt{LO}) + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{\mathtt{LO}}}{T_j} \right\rceil C_j(\mathtt{LO}) \tag{3.2}$$

$$R_i^{\mathtt{HI}} = C_i(\mathtt{HI}) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i^{\mathtt{HI}}}{T_j} \right\rceil C_j(\mathtt{HI}) \tag{3.3}$$

where $hpH(i)$ is the set of tasks with `HI`-criticality and higher priority than $\tau_i$. For each mode the respective recurrence equation needs to be solved for all tasks, and the response time of the tasks need to be checked against the task deadlines.

The analysis for the moment of the criticality change is more involved. If the criticality change happens at some arbitrary time $s$, the response time equation of task $\tau_i$ can be modelled as

$$R_i^s = C_i(\mathtt{HI}) + I_L(s) + I_H(s) \tag{3.4}$$

where $I_L(s)$ and $I_H(s)$ represent the interference from tasks of lower and higher or equal criticality than $\tau_i$, respectively.

$I_L(s)$ can be upper bounded by

$$I_L(s) = \sum_{j \in hpL(i)} \left( \left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(\mathtt{LO}) \tag{3.5}$$

where $hpL(i)$ is the set of tasks with `LO`-criticality and higher or equal priority than $\tau_i$.

In order to define $I_H(s)$ we need a term for the maximum number of releases of an interfering task $\tau_k$ after the criticality change s:

$$M(k,s,t) = min\left\{ \left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1, \left\lceil \frac{t}{T_k} \right\rceil \right\} \tag{3.6}$$

Using this term, an upper bound for $I_H(s)$ can be defined as

$$I_H(s) = \sum_{k \in hpH(i)} \left\{ \Big( M(k,s,t)\ C_k(\mathtt{HI}) \Big) + \left( \left\lceil \frac{t}{T_k} \right\rceil - M(k,s,t) \right) C_j(L_i) \right\} \tag{3.7}$$

where $hpH(i)$ is the set of tasks with `HI`-criticality and higher priority than $\tau_i$.

Finally, the response time of $\tau_i$ for the moment of the criticality change is given by

$$R_i^* = max(R_i^s)\forall s \tag{3.8}$$

Since $R_i^s$ can only increase at points when a `LO`-criticality task is released, only those values for $s$ need to be checked.

For details on the derivation of these equations we point to the original paper [BBD11]. It should be noted that their analysis allows `LO`-criticality tasks to run to completion after the criticality level of the system is raised. The run-time model which we will use for our analysis assumes that `LO`-criticality tasks can be aborted immediately, so they do not complete their runs.

Kelly, Aydin and Zhao [KAZ11] did a comparative study of different task allocation (bin-packing) and priority assignment algorithms for partitioned fixed-priority mixed-criticality task sets on multiprocessors, using Vestal's RTA [Ves07]. They compared first-fit (FF), best-fit (BF) and worst-fit (WF) for the task allocation algorithms in conjunction with decreasing utilisation (DU) and decreasing criticality (DC) as initial orderings of the task set. For DU they used the *nominal utilisation* $U_i(L_i)$, i.e. the utilisation at the criticality level of the task. For DC they also used DU in case of tie-breaks. In terms of priority assignment they only considered RM and OPA. This also implies that their experiments were limited to implicit deadline task sets. Kelly et al. concluded that choosing the right priority assignment is more important than choosing a good task allocation heuristic. FF and BF had comparable performance while WF performed worst. They also observed that OPA clearly outperformed RM and that the combination of DC-OPA performed best. Our multiprocessor evaluation section extends their set of experiments.

Pathan [Pat12] introduced a new RTA-based schedulability test for global fixed-priority multiprocessor systems. It works for arbitrary criticality levels, and the applicability of Audsley's priority assignment algorithm is also shown. This work is mostly interesting because our uniprocessor test is based on similar ideas. Both approaches use workload analysis to come up with tight response time tests. A more detailed description of this approach is deferred to chapter 5 which explains our new uniprocessor test.

In [BLS10] Baruah, Li and Stougie adapted Audsley's OPA [Aud91] to MC system instances which consist only of non-recurrent jobs. They call this adapted version "Own Criticality-Based Priorities (OCBP)", because when a job is considered for the lowest priority only the parameters for the criticality level of that job are considered in the analysis. They also show that OCBP is able to find a complete priority ordering in polynomial time, if such an ordering exists. Systems for which such an ordering exists are called OCBP-schedulable. Some parts of this

work have also been published in [BBD$^+$10], but most importantly Baruah et al. proved in [BBD$^+$10] that the scheduling of an MC system is NP-hard.

In [LB10] Li and Baruah generalised the OCBP algorithm to sporadic task systems. They showed that under certain conditions the priority ordering needs to be recomputed, and they devised an algorithm which does this in pseudo-polynomial time. Building on this work, Guan, Ekberg, Stigge and Yi [GESY11] developed an algorithm called Priority List Reuse Scheduling (PLRS) which can do the priority re-computation in polynomial time instead.

A more high-level view of an MC system is taken by Mollison et al. [MEA$^+$10]. They developed an architecture for MC systems with heavier emphasis on real-world requirements. Standards such as the RCTA standard DO-178B have a limited number of criticality levels, usually around 5, so they came up with a 5-level architecture. Their architecture relies on container or server based separation (see e.g. Lehoczky et al. [Leh87]). It is a complex hybrid architecture in which some levels use partitioned scheduling and other levels use global scheduling. Each of the architectural levels uses a different scheduling algorithm, in accordance with their criticality. Every task gets a "time budget" which is equal to its WCET in its criticality level, and if it does not use its entire budget when it runs, lower criticality tasks can use it. The whole architecture is based on the assumption that WCET are highly pessimistic, especially for high-criticality tasks. This is also obvious by their choice of using EDF for high criticality tasks, since EDF is very bad in handling overload scenarios. The approach Mollison et al. chose is fundamentally different from ours, because they do not consider overloads or run-time enforcement of execution times.

De Niz, Lakshmanan and Rajkumar [dNLR09] argue along similar lines as Mollison et al., but de Niz et al. also take WCET overruns into account. Based on the observation that the WCET of a task is hard to calculate and that it rarely occurs in practice, they propose a new algorithm they call zero-slack scheduling. It can be adapted to different scheduling algorithms, and they exemplify this with RM scheduling, the result of which they call Zero-Slack Rate-Monotonic (ZSRM) scheduling.

In a related paper, Lakshmanan, de Niz, Rajkumar and Moreno [LdNRM10] extend their previous work to distributed cyber physical systems, which in essence are partitioned multiprocessor systems. Once again they considered overload scenarios and they developed a new overload-resilience metric they call ductility. The higher the ductility of a system, the more resilient it is to overload. Ductility takes both the bin-packing and the scheduling algorithms into account. They present Compress-on-Overload Packing (COP), a new bin-packing algorithm, and they compare it with the WF decreasing bin-packing algorithm. As scheduling algorithm they use ZSRM, and they conclude that COP is more resilient to overload

than WF decreasing in the average case.

Santy, George, Thierry and Goossens [SGTG12] consider $\mathcal{L}$-criticality systems (i.e. systems with $\mathcal{L}$ criticality levels) and investigate under which conditions it is possible to relax the constraint of immediately dropping tasks whose criticality level has been exceeded. They also discuss when the criticality level can be reset to a lower level again. Santy et al. also provide a short proof that the critical instant for a FP scheduled mixed-criticality system is the same as for traditional FP systems. It should be noted that their system model is fundamentally different from ours. They assume that the WCET of low-criticality tasks does not increase, that is, $C(\texttt{LO}) = C(\texttt{HI})$ for all LO criticality tasks in a dual-criticality system. Their paper also nicely illustrates that system requirements play a large role in system design, since their goal of decreasing the number of dropped tasks is somewhat orthogonal to our goal of increasing the schedulability.

Dorin, Richard, Richard and Goossens [DRRG10] proved that Audsley's OPA is also optimal for traditional fixed-priority MC systems, which also implies optimality of Vestal's algorithm [Ves07] for those systems. They also extended the sensitivity analysis of Bini et al. [BDNB08] to MC systems.

# 4

# System Model and Notation

THE system consists of a set of $n$ sporadic tasks $\tau_i \in \Gamma$ where $i \in \{1..n\}$. It is a dual-criticality system with the levels low (LO) and high (HI) where LO $<$ HI, which is scheduled preemptively on $m$ homogeneous processors. Each partition of the task system $\Gamma$ will be denoted by $\Gamma_j$ where $j \in \{1..m\}$.

The tasks have constrained deadlines, and each task is defined by the tuple $(L_i, C_i, T_i, D_i)$, where

- $L_i \in \{\mathtt{LO},\mathtt{HI}\}$ is the criticality level of the task.
- $C_i$ is a vector $(C_i(\mathtt{LO}), C_i(\mathtt{HI}))$ of the WCETs of task $\tau_i$ at both criticality levels. The WCET of task $\tau_i$ at criticality level $\ell$ is equal to $C_i(\ell)$.
- $T_i \in \mathbb{R}^+$ is the minimum inter-arrival time (period) of the task.
- $D_i \in \mathbb{R}^+$ is the relative deadline such that $D_i \leq T_i$.

The WCET is monotonically non-decreasing, so $C_i(\mathtt{LO}) \leq C_i(\mathtt{HI})$ for each task $\tau_i \in \Gamma$ . The $n$th job of task $\tau_i$ will be denoted by $J_i^n$. Note that the WCET of a task does not directly correspond to the actual run-time of a specific job.

Furthermore, let $J_i^x$ denote the $x$-th job (invocation) of task $\tau_i$, and $r_i^x$ the release time of job $J_i^x$.

Each task $\tau_i$ also has a distinct fixed-priority $p_i$. The set of tasks with *higher priority* than task $\tau_i$ will be denoted by `hp(i)`. We can then define `hpH(i)`and `hpL(i)`as follows:

$$\mathtt{hpH(i)} = \{\tau_j \mid L_j \geq L_i \text{ and } \tau_j \in \mathtt{hp(i)}\}$$
$$\mathtt{hpL(i)} = \{\tau_j \mid L_j < L_i \text{ and } \tau_j \in \mathtt{hp(i)}\}$$

In other words, `hpL(i)` is the set of tasks with higher priority but *lower criticality*, and `hpH(i)` is the set of higher priority and *higher-or-equal criticality* than $\tau_i$. Note that `hp(i)` = `hpL(i)` ∪ `hpH(i)`.

The run-time model of the system is as follows. There is a system-wide criticality level indicator $\mathcal{L}$ which denotes the current criticality level. Task execution times are monitored at run-time. The system starts out at `LO`-criticality, but switches to `HI`-criticality when any of the tasks executes for more than its specified $C_i(LO)$ time units. As soon as the switch occurs, all `LO`-criticality jobs are suspended, i.e. they will not be allowed to run to completion. Resetting the criticality level of the system is not considered.

# 5

# Uniprocessor Schedulability Test - IAMC

PLEASE note that this chapter was originally written by Risat Pathan. I only made minor corrections and abbreviated parts of the analysis. Any mistakes in this document are obviously still my own. Since this analysis has not been published anywhere yet and it forms the basis for the rest of this thesis, it needs to be included. The basic techniques behind the analysis are similar to the ones presented in [Pat12].

The system model presented in the previous chapter is assumed in the remainder of this chapter. Since this schedulability test is essentially a more precise schedulability test using the run-time model and strategy proposed in [BBD11], we call it Improved AMC (IAMC).

According to the AMC strategy, all the `LO`-critical tasks are de-scheduled if any job executes for more than its corresponding low-criticality execution time. Here we present the response time analysis for the `HI`-critical tasks based on the AMC strategy. We find the largest response time for any job of a `HI`-critical task. If a job of a HI-critical task $\tau_i$ does not experience any criticality change[1], then it executes for at most $C_i(\texttt{LO})$ time units, and its response time is given (according to [BBD11]) as follows:

$$R_i^{\texttt{LO}} = C_i(\texttt{LO}) + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{\texttt{LO}}}{T_j} \right\rceil C_j(\texttt{LO}) \tag{5.1}$$

[1]A criticality change is first triggered by a job of task $\tau_i$ or by job of any other task. This "trigger" event happens when a job of task does not signal completion after executing for its corresponding low-criticality execution time

In the rest of this chapter, we present the *schedulability analysis* of a HI-critical task $\tau_i$ considering the case where it experiences a criticality change. Based on this schedulability analysis, we compute the worst-case response time of $\tau_i$. When a HI-critical task $\tau_i$ experiences a criticality change, the following facts are true:

- The job, say $J_i^x$, of the HI-critical task $\tau_i$ that *first* experiences the criticality change executes for $C_i(HI)$ time units.
- The criticality change occurs somewhere within $[r_i^x, r_i^x + R_i^{LO}]$ where $r_i^x$ is the release time of job $J_i^x$.

In this work, we derive an upper bound on the response time of job $J_i^x$ considering that the criticality change occurs anywhere within $[r_i^x, r_i^x + R_i^{LO}]$. It has already been pointed out in [BBD11] that the exact response time analysis of this AMC strategy is likely to be intractable. We present a sufficient schedulability test here based on response time analysis.

Assume $s$ is the time instant relative to the release time of job $J_i^x$ at which the criticality change is triggered, i.e., the criticality change occurs at $(r_i^x + s)$ where $r_i^x$ is the release time of job $J_i^x$. Let $t$ be the length of the problem window in which we analyse the schedulability of job $J_i^x$ where $s < t \leq d_i^x$ and $d_i^x$ is the absolute deadline of job $J_i^x$. This scenario is depicted in Figure 5.1.

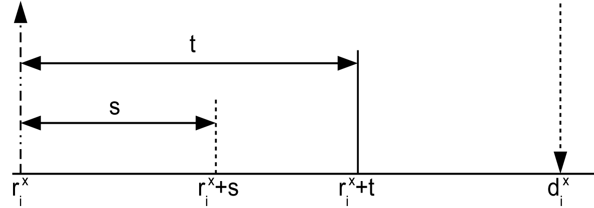


**Figure 5.1:** The problem window of length $t$ where the criticality is changed at $(r_i^x + s)$

According to the AMC strategy, no LO-criticality tasks are allowed to execute after $(r_i^x + s)$. Thus, the interference that job $J_i^x$ suffers within the problem window $[r_i^x, r_i^x + t]$ depends on the following two interference factors:

- $I_L(s)$: the interference due to the tasks in $hpL(i)$ within the interval $[r_i^x, r_i^x + s]$
- $I_H(s)$: the interference due to the tasks in $hpH(i)$ within the interval $[r_i^x, r_i^x + t]$

We denote $R_i^s$ the response time of task $\tau_i$ when a criticality change occurs at time $s$ relative to the release time of some job of task $\tau_i$. The value of $R_i^s$ is defined as follows [BBD11]:

$$R_i^s = C_i(HI) + I_L(s) + I_H(s) \tag{5.2}$$

In the following sections we present the schedulability analysis to calculate the values of $I_L(s)$ and $I_H(s)$.

## 5.1 Calculating $I_L(s)$

Since job $J_i^x$ experiences the criticality change at time instant $(r_i^x + s)$, this job has not signalled completion at or prior to time instant $(r_i^x + s)$. In other words, in the interval $[r_i^x, r_i^x + s]$ the system is busy executing job $J_i^x$ in addition to the jobs of the tasks in $(hpL(i) \cup hpH(i))$. According to the schedulability analysis in [BBD11], the interference on job $J_i^x$ due to the execution of the jobs of tasks in $hpL(i)$ is maximised if all the higher priority tasks in $hpL(i)$ are released simultaneously at time $r_i^x$.

We denote $\hat{I}_L(s)$ the upper bound on the sum of the interference due to the tasks in $hpL(i)$ within $[r_i^x, r_i^x + s]$. Since no task $\tau_j \in hpL(i)$ executes after $(r_i^x + s)$, the value of $\hat{I}_L(s)$ is given as follows:

$$\hat{I}_L(s) = \sum_{j \in hpL(i)} \left( \left\lfloor \frac{s}{T_j} \right\rfloor C_j(LO) + min\{C_j(\texttt{LO}), s - \left\lfloor \frac{s}{T_j} \right\rfloor T_j\} \right) \tag{5.3}$$

In contrast, the work in [BBD11] considers that any LO-critical job released at or before $s$ completes its execution. Such consideration may introduce significant amount of pessimism when the WCET of such a LO-critical task is large, since some of its execution may need to take place after time instant $(r_i^x + s)$. However, run-time monitoring is enforced in the AMC strategy. Therefore, as soon as the criticality change occurs at $(r_i^x + s)$, all the LO-critical tasks can be *de-queued* immediately from the ready queue.

One important observation regarding Eq. (5.3) is that the value of $\hat{I}_L(s)$ could be greater than $s$. However, the maximum amount of execution that can take place within the interval $[r_i^x, r_i^x + s]$ is upper bounded by $s$. Consequently, the value of $I_L(s)$, i.e., interference due to the task in set $hpL(i)$ on job $J_i^x$ within the interval $[r_i^x, r_i^x + s]$, is given as follows:

$$I_L(s) = min\{s, \hat{I}_L(s)\} \tag{5.4}$$

## 5.2 Calculating $I_H(s)$

In this subsection, we calculate the interference on job $J_i^x$ due to the tasks in $hpH(i)$ within the problem window $[r_i^x, r_i^x + t]$. We denote $I_H(k,s,t)$ the interference on job $J_i^x$ within the problem window of length $t$ due to the jobs of task $\tau_k \in hpH(i)$

where the criticality change occurs at $(r_i^x + s)$. Thus, we have

$$I_H(s) = \sum_{\tau_k \in hpH(i)} I_H(k,s,t) \tag{5.5}$$

We need to calculate the value of $I_H(k,s,t)$ for all $\tau_k \in hpH(i)$. In order to calculate $I_H(k,s,t)$, we consider two different cases: case (i) $s \leq D_k$, and case (ii) $s > D_k$. We calculate the upper bound on workload of task $\tau_k$ within $[r_i^x, r_i^x + t]$ for each of these two cases. The upper bound on workload is an upper bound on interference within $[r_i^x, r_i^x + t]$ due to task $\tau_k$.

**Case(i)** $s < D_k$**:** The earlier the criticality change occurs within $[r_i^x, r_i^x + t]$, the larger is the workload of task $\tau_k$ within $[r_i^x, r_i^x + t]$. This is because jobs of task $\tau_k$ execute for $C_k(HI)$ time units after the criticality change, and $C_k(\texttt{HI}) > C_k(\texttt{LO})$ by definition. Consequently, the workload of task $\tau_k$ within $[r_i^x, r_i^x + t]$ considering

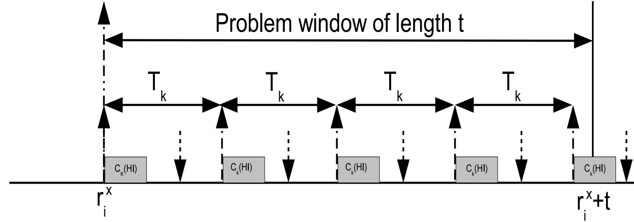


**Figure 5.2:** Worst-case release pattern of jobs of task $\tau_k$ where $s \leq D_k$

$s = 0$ is greater than or equal to that of considering $s > 0$. The worst-case releases of jobs of task $\tau_k$ assuming $s = 0$ is given in Figure 5.2. One job of task $\tau_k$ is released at time $r_i^x$ and the subsequent jobs of $\tau_k$ arrive as early as possible.

According to [BBD11], the number of jobs of task $\tau_k$ that may be released within the interval $[r_i^x, r_i^x + t]$ is at most $\lceil \frac{t}{T_k} \rceil$. The upper bound on workload of the jobs of task $\tau_k$ within $[r_i^x, r_i^x + t]$ is then given as follows:

$$I_H(k,s,t) = \left\lceil \frac{t}{T_k} \right\rceil C_k(HI) \qquad \text{if } s \leq D_k \tag{5.6}$$

**Case(ii)** $s > D_k$**:** For this case, we consider a particular release pattern of the jobs of task $\tau_k$ within $[r_i^x, r_i^x + t]$. We call this particular release pattern the ***reference pattern***. We first calculate the upper bound on the total workload of the jobs of task $\tau_k$ for this reference pattern. Then, by shifting the problem window within the reference pattern (both in leftward and rightward directions) we determine the ***maximum net increase*** in workload in addition to the workload calculated for the reference pattern. The sum of the workload of the reference

pattern and the net increase in workload due to any possible shift of the problem window within the reference pattern is the upper bound on interference caused by the jobs of task $\tau_k$ on job $J_i^x$.

The reference pattern is defined considering releases of the jobs of task $\tau_k$ within the interval $[r_i^x, r_i^x + t]$ as follows (see Figure 5.3):

- one job of task $\tau_k$ is released at time $(r_i^x + t - C_k(HI))$ and executes for $C_k(HI)$ time units as early as possible, and
- other jobs of $\tau_k$ are released as close as possible to the job of task $\tau_k$ that is released at $(r_i^x + t - C_k(HI))$. This scenario considers strictly periodic releases of the jobs of task $\tau_k$.

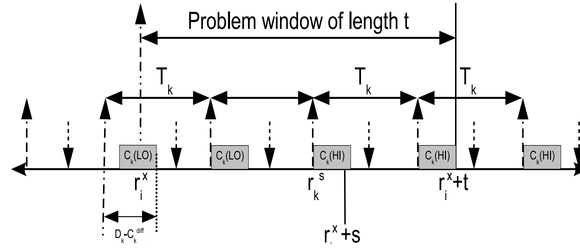


**Figure 5.3:** Reference Pattern for the case $s > D_k$

We compute the maximum workload of the jobs of task $\tau_k \in hpH(i)$ in the reference pattern within the problem window $[r_i^x, r_i^x + t]$ in two steps as follows:

- **STEP1:** The upper bound on total workload of task $\tau_k$ within $[r_i^x, r_i^x + t]$ in the reference pattern (Figure 5.3) is calculated.
- **STEP2:** By considering any possible shifts of the problem window in the reference pattern for $\alpha$ time units, $0 \leq \alpha \leq T_k$, either in leftward or rightward direction, we find the ***maximum net increase*** in workload due to such shift in addition to the workload calculated in Step 1. Note that shifting the problem window exactly for $T_k$ time units either in leftward or rightward direction results in the same release pattern as in Figure 5.3. Therefore, we do not need to consider any shifts greater than $T_k$ time units.

The sum of the two workload factors calculated in Step 1 and Step 2 is the maximum workload (hence, the interference) due to the jobs of task $\tau_k \in hpH(i)$ within the problem window $[r_i^x, r_i^x + t]$. Now we present the details of Step 1 and Step 2.

**STEP 1:** In this step, we find the upper bound on total workload of the jobs of task $\tau_k$ for the reference pattern in Figure 5.3. Consider a job $J_k^s$ that satisfies the following condition in the reference pattern:

$$r_k^s \leq r_i^x + s < r_k^{s+1} \tag{5.7}$$

According to Eq. (5.7), the criticality change occurs at or after the release time of job $J_k^s$ but prior to the release time of job $J_k^{s+1}$. No job released earlier than $J_k^s$ in the reference pattern can experience the criticality change. If job $J_k^s$ triggers the criticality change, then it can only trigger the criticality change at or prior to time instant $(r_k^s + R_k^{LO})$. Therefore, if $(r_k^s + R_k^{LO}) \leq (r_i^x + s)$, then job $J_k^s$ does not experience the criticality change and also has not triggered the criticality change (see Figure 5.4). On the other hand, if $(r_k^s + R_k^{LO}) > (r_i^x + s)$, then job $J_k^s$ may experience (or trigger) the criticality change (see Figure 5.5). Consequently, $J_k^s$ executes for $C_k(LO)$ time units if $(r_k^s + R_k^{LO}) < (r_i^x + s)$, otherwise, we consider job $J_k^s$ executes for $C_k(HI)$ time units in the reference pattern in Figure 5.3. Any job of task $\tau_k$ that is released after job $J_k^s$ executes for $C_k(HI)$ time units in the reference pattern in Figure 5.3. And, any job of $\tau_k$ that is released prior to the release of job $J_k^s$ executes for $C_k(LO)$ time units in the reference pattern.

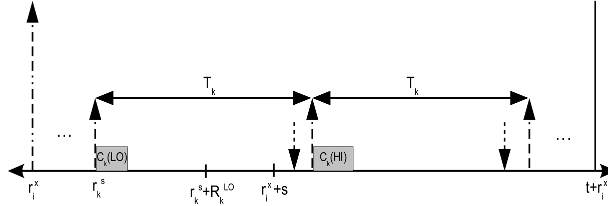


**Figure 5.4:** Reference Pattern with $(r_k^s + R_k^{LO}) < (r_i^x + s)$

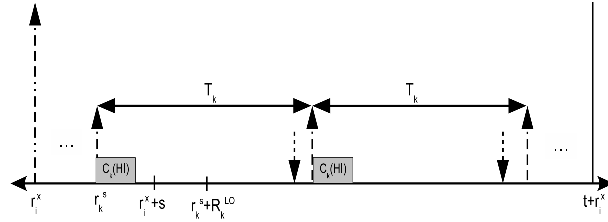


**Figure 5.5:** Reference Pattern with $(r_k^s + R_k^{LO}) \geq (r_i^x + s)$

Now we calculate an upper bound on interference according to the reference pattern in Figure 5.3 on job $J_i^x$. Our objective is to find how many of the jobs out of $\lceil \frac{t}{T_k} \rceil$ jobs of task $\tau_k$ observe the criticality change, and thus execute for $C_k(HI)$

time units. We denote $N^t_{k,s}$ the maximum number of jobs of task $\tau_k$ that may execute within $[r^x_i + s,\ r^x_i + t - C_k(HI)]$ in the reference pattern. The value of $N^t_{k,s}$ is given as follows:

$$N^t_{k,s} = \left\lceil \frac{max\{0, t - s - C_k(HI)\}}{T_k} \right\rceil \quad (5.8)$$

Since job $J^s_k$ satisfies Eq. (5.7), the release time of job $J^s_k$ is given as follows:

$$r^s_k = r^x_i + t - C_k(HI) - N^t_{k,s} \cdot T_k \quad (5.9)$$

We denote $\hat{M}(k,s,t)$ the number of jobs of $\tau_k$ that execute for $C_k(HI)$ time units in the reference pattern of Figure 5.3. The value of $\hat{M}(k,s,t)$ is given as follows:

$$\hat{M}(k,s,t) = \begin{cases} N^t_{k,s} & \text{if } (r^s_k + R^{LO}_k) < (r^x_i + s) \\ N^t_{k,s} + 1 & \text{if } (r^s_k + R^{LO}_k) \geq (r^x_i + s) \end{cases} \quad (5.10)$$

Equivalently, because $r^s_k = (r^x_i + t - C_k(HI) - N^t_{k,s} \cdot T_k)$ in Eq. (5.9), we have

$$\hat{M}(k,s,t) = \begin{cases} N^t_{k,s} & \text{if } (t - C_k(HI) - N^t_{k,s} T_k + R^{LO}_k) < s \\ N^t_{k,s} + 1 & \text{if } (t - C_k(HI) - N^t_{k,s} T_k + R^{LO}_k) \geq s \end{cases} \quad (5.11)$$

Thus, the value of $\hat{M}(k,s,t)$ is dependent of the values of $s$, $t$, task $\tau_k$'s parameters $(C_k(HI), T_k)$ and $R^{LO}_k$.

Note that when applying OPA to determine the priority of task $\tau_i$, the value of $R^{LO}_k$ is not known since (i) evaluating $R^{LO}_k$ requires to know $hp(k)$ and (ii) OPA must not be dependent on the relative priority ordering of the tasks in $hpH(i)$. If OPA is to be used, an upper bound on $R^{LO}_k$ can be used to evaluate the value of $\hat{M}(k,s,t)$. By assuming that task $\tau_k$ meets its deadline, an upper bound on $R^{LO}_k$ is $D_k - (C_k(HI) - C_k(LO))$.

The value of $(N^t_{k,s} + 1)$ in Eq. (5.10) is never greater than $\lceil \frac{t}{T_k} \rceil$ for the case where $s > D_k$. If the value of $\hat{M}(k,s,t)$ is equal to $\lceil \frac{t}{T_k} \rceil$, then all the $\lceil \frac{t}{T_k} \rceil$ jobs are considered to execute for $C_k(HI)$ time units and we have the following:

$$I_H(k,s,t) = \lceil \frac{t}{T_k} \rceil \cdot C_k(HI)$$

Consequently, if $\hat{M}(k,s,t) = \lceil \frac{t}{T_k} \rceil$, we do not need to consider shifting the problem window in the reference pattern. If $\hat{M}(k,s,t) < \lceil \frac{t}{T_k} \rceil$, then we have to consider shifting the problem window in the reference pattern to determine any possible increase in interference.

For the case where $\hat{M}(k,s,t) < \lceil \frac{t}{T_k} \rceil$, we denote $\hat{I}(k,s,t)$ the interference in the reference pattern due to task $\tau_k$ on job $J_i^x$ within the interval $[r_i^x,\ r_i^x + t]$. The value of $\hat{I}(k,s,t)$ is given as follows:

$$\hat{I}(k,s,t) = \hat{M}(k,s,t) \cdot C_k(HI) \ + \left( \left\lceil \frac{t}{T_k} \right\rceil - \hat{M}(k,s,t) \right) \cdot C_k(LO) \tag{5.12}$$

**STEP2:** In this step, by shifting of the problem window in Figure 5.3 for $\alpha$ time units, where $0 \leq \alpha \leq T_k$, we determine the ***maximum net increase*** in workload in addition to the workload calculated in Step 1. Let $\delta$ be the maximum net increase in workload of task $\tau_k$ due to any possible shifts of the problem window in Figure 5.3 either in leftward or rightward direction.

It has already been shown in [Pat12] that $\delta$ is bounded by $C_k(\texttt{HI}) - C_k(\texttt{LO})$. The upper bound on interference due to the jobs of task $\tau_k$ within the interval $[r_i^x, r_i^x + t]$ (not necessarily according to the reference pattern) is thus:

$$I_H(k,s,t) = \hat{I}(k,s,t) + C_k(\texttt{HI}) - C_k(\texttt{LO}) \tag{5.13}$$

## 5.3 Summary of IAMC

The response time of HI-critical task $\tau_i$ is calculated for a given pair $(s,t)$ as follows:

- The maximum number of jobs of task $\tau_k \in hpH(i)$ that may execute within an interval of length $(t - s - C_k(HI))$ is calculated as follows:

$$N_{k,s}^t = \left\lceil \frac{max\{0, t - s - C_k(HI)\}}{T_k} \right\rceil$$

- The maximum number of jobs of task $\tau_k \in hpH(i)$ where each such job is executed for $C_k(HI)$ time units in the reference pattern in Figure 5.3 is determined as follows:

$$\hat{M}(k,s,t) = \begin{cases} N_{k,s}^t & \text{if } (t - C_k(HI) - N_{k,s}^t T_k + R_k^{LO}) < s \\ N_{k,s}^t + 1 & \text{if } (t - C_k(HI) - N_{k,s}^t T_k + R_k^{LO}) \geq s \end{cases}$$

- The interference due to task $\tau_k \in hpH(i)$ is given as follows:

$$I_H(k,s,t) = \begin{cases} \left\lceil \frac{t}{T_k} \right\rceil C_k(HI) & \text{if } s \leq D_k \\ \hat{M}(k,s,t) \cdot C_k(HI) + (\left\lceil \frac{t}{T_k} \right\rceil - M(k,s,t)) \cdot C_k(LO) + \delta & \text{otherwise} \end{cases}$$

  where $\delta = C_k(\texttt{HI}) - C_k(\texttt{LO})$.

- The interference due to all the LO-critical tasks in $hpL(i)$ is given as follows:

$$I_L(s) = min\{s, \hat{I}_L(s)\}$$

  where

$$\hat{I}_L(s) = \sum_{j \in hpL(i)} \left( \left\lfloor \frac{s}{T_j} \right\rfloor C_j(LO) + min\{C_j(LO), s - \left\lfloor \frac{s}{T_j} \right\rfloor T_j\} \right)$$

- The response time of a HI-critical task $\tau_i$ is calculated as follows:

$$R_i^s = C_i(HI) + I_L(s) + \sum_{k \in hpH(i)} I_H(k,s,R_i^s)$$

  where $R_i^s = C_i(HI)$ is used as the initial value for $R_i^s$.

- The worst-case response time of task $\tau_i$ is $R_i^* = max(R_i^s) \forall s$.

# 6

# Evaluation of IAMC for Uniprocessors

In order to gain an understanding of the performance of the uniprocessor schedulability test IAMC presented in chapter 5, we will compare it empirically with AMC [BBD11].

In order to do the comparison, a large number of random task sets with fixed parameters have been generated. Each of the generated task sets was then tested for schedulability with both AMC and IAMC as the schedulability test. This gives an idea how many task sets were schedulable by IAMC which were not schedulable by AMC. Both AMC and IAMC were tested with deadline-monotonic priority ordering and Audsley's approach to priority assignment (OPA).

The details of the experimental setup and the results of the experiments will be explained in the rest of this chapter.

## 6.1 Experimental Setup

In this section we will outline the setup of the performed experiments. In order to facilitate the comparison, our setup is very similar to the one used by Baruah, Burns and Davis in [BBD11].

### 6.1.1 Taskset Generation

Each taskset is defined by its utilisation, i.e. by the sum of all individual task utilisations in the taskset. In order to randomly generate tasksets with a predetermined utilisation value, the UUniFast algorithm as described by Bini and Buttazzo

[BB05] was used. As input it takes the utilisation sum $U$ to which the individual utilisation values should add up, and the number of tasks $n$. UUniFast has runtime complexity O(n) and yields a uniform distribution of the utilisation values. For details see Bini and Buttazzo's paper [BB05]. Our C implementation is shown in listing 6.1.

```c
/**
   UUniFast(n, U) -
   returns an n-size vector, where the total sum is equal to U.
*/
long double* uunifast(int n, long double U)
{
    long double * vectU = malloc(n * sizeof(long double));

    long double sumU = U;
    long double nextSumU;

    for (int i = 0; i < n - 1; i++)
    {
        nextSumU = sumU * pow(rand(), 1.0 / (n - i));
        vectU[i] = sumU - nextSumU;
        sumU = nextSumU;
    }
    vectU[n - 1] = sumU;

    return vectU;
}
```

**Listing 6.1:** UUniFast implementation in C

The following parameters were used to randomly generate the tasksets for the experiments:

- Task utilisations ($U_i = C_i/T_i$) were generated with the previously discussed *UUniFast algorithm* [BB05].
- Task periods were generated using a log-uniform distribution with a range from 10ms to 1000ms.
- LO-criticality execution times were computed from the generated utilisation values and periods as follows: $C_i(\texttt{LO}) = U_i \cdot T_i$.
- HI-criticality execution times were computed based on $C_i(\texttt{LO})$ and a linear criticality factor $CF$: $C_i(\texttt{HI}) = CF \cdot C_i(\texttt{LO})$, where $CF \geq 1$.
- Task deadlines were constrained ($D_i < T_i$) and were generated using a random uniform distribution. The intervals differed depending on the criticality

level of the task: For LO-criticality tasks the interval was $[C_i(\texttt{LO}), T_i]$, and for HI-criticality tasks it was $[C_i(\texttt{HI}), T_i]$.

- The criticality probability $CP$ determined the probability of a task to be of HI-criticality. The function for determining the criticality can then be defined as:

$$crit(b, CP) = \begin{cases} \texttt{HI}, & \text{if } b \leq CP \\ \texttt{LO}, & \text{otherwise} \end{cases}$$

where $b \in [0,1]$ is randomly generated for each task.

### 6.1.2 Schedulability Tests

The following schedulability tests have been used for the experiments:

- UB-H&L: A composite upper bound, as described in [BBD11]. In order for a taskset to be deemed schedulable under this scheme, all tasks must be schedulable considering only $C_i(\texttt{LO})$, and all HI-criticality tasks must be schedulable using their $C_i(\texttt{HI})$ values.
- AMC: The schedulability test described in [BBD11] as AMC-max.
- IAMC: The schedulability test described in chapter 5.

With the exception of UB-H&L for which DM is optimal, each of these test has been run with both DM and OPA as priority ordering.

## 6.2 Experiments

In order to show the schedulability properties of each schedulability test, 1000 task sets were generated for a specific utilisation level. Each of those task sets was then tested, and the percentage of schedulable task sets at that specific utilisation level was recorded.

The taskset utilisation runs from 0.025 up to 1 in steps of 0.025, so there are a total of 40 data points. At each utilisation level (data point) the 1000 tasksets were tested for schedulability with AMC and IAMC. The results have been plotted and a number of these plots will be shown and discussed subsequently. The x-axis shows the taskset utilisation, and the y-axis shows the percentage of the 1000 tasksets that was found to be schedulable with the algorithm and priority assignment in question.

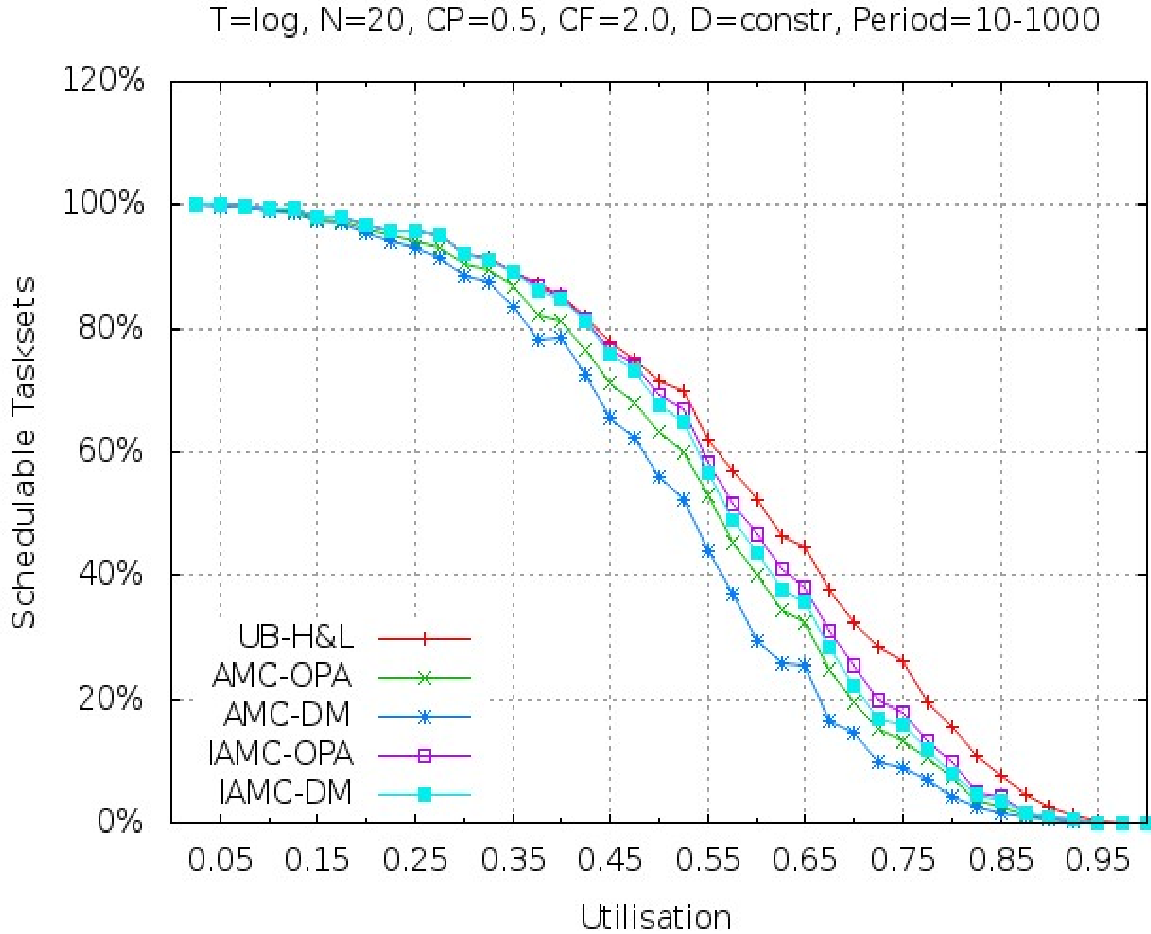


**Figure 6.1:** Percentage of schedulable task sets

Each plot has a header which explains the parameters which have been used for the experiment in question. The explanation of the header is as follows:

- T: indicates how the periods were generated. "log" means it was a logarithmic distribution, "unif" means it was a uniform distribution.
- N: the number of tasks per task set.
- CP: criticality probability. The higher the CP value, the higher the probability that a generated task is of high criticality (see section 6.1.1).
- CF: criticality factor. A scaling factor which determines the relation of $C_i(HI)$ and $C_i(LO)$ (see section 6.1.1).

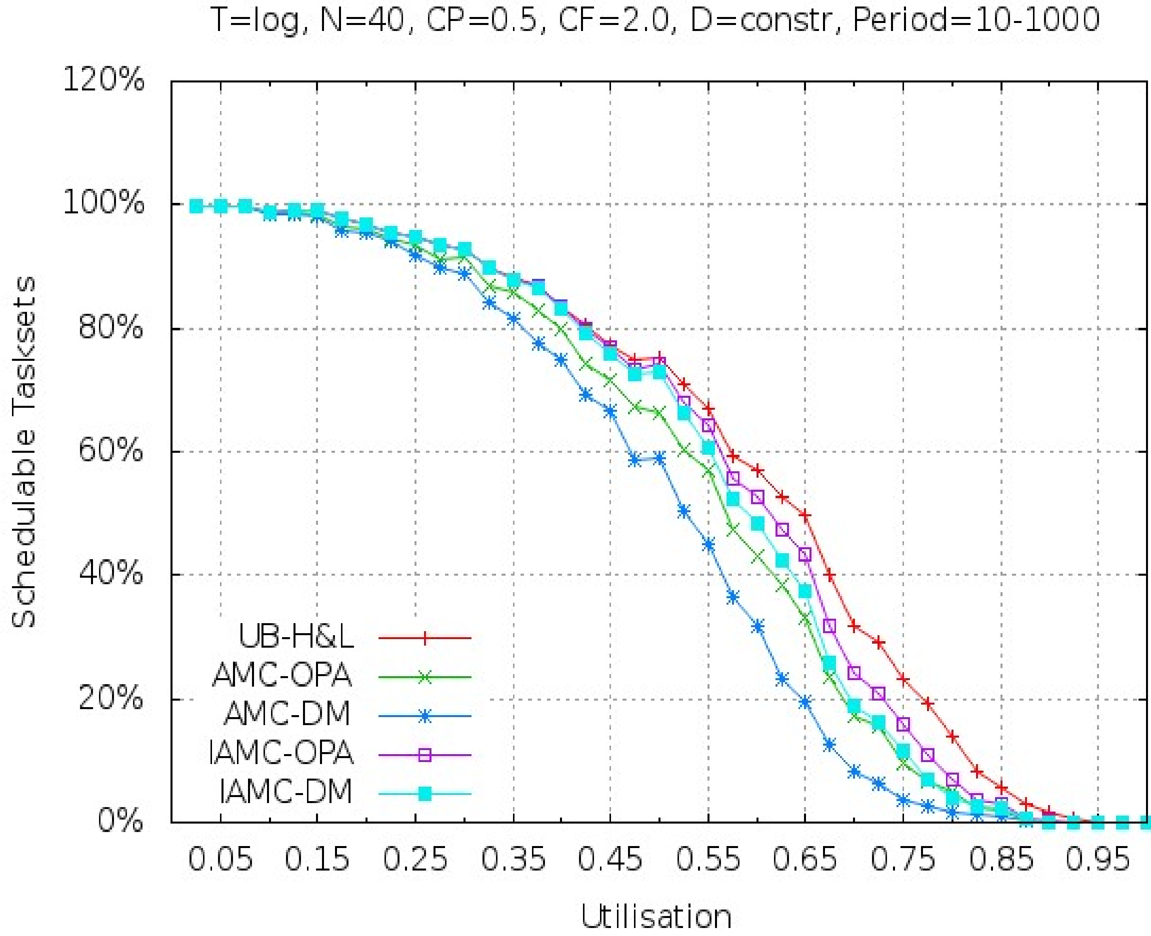


**Figure 6.2:** Higher number of tasks per task set

- D: indicates whether the deadlines are constrained (constr.) or implicit (DT).
- Period: indicates the range of the periods in milliseconds ($T_i$).

Figure 6.1 shows the plot for the case with 20 tasks per taskset ($n = 20$), a criticality factor of 2.0 and a criticality probability of 0.5, i.e. half of the tasks are of HI-criticality. All other parameters are as described in section 6.1.1.

Figures 6.2, 6.3 and 6.4 show variations for $n$, $CF$ and $CP$, respectively.

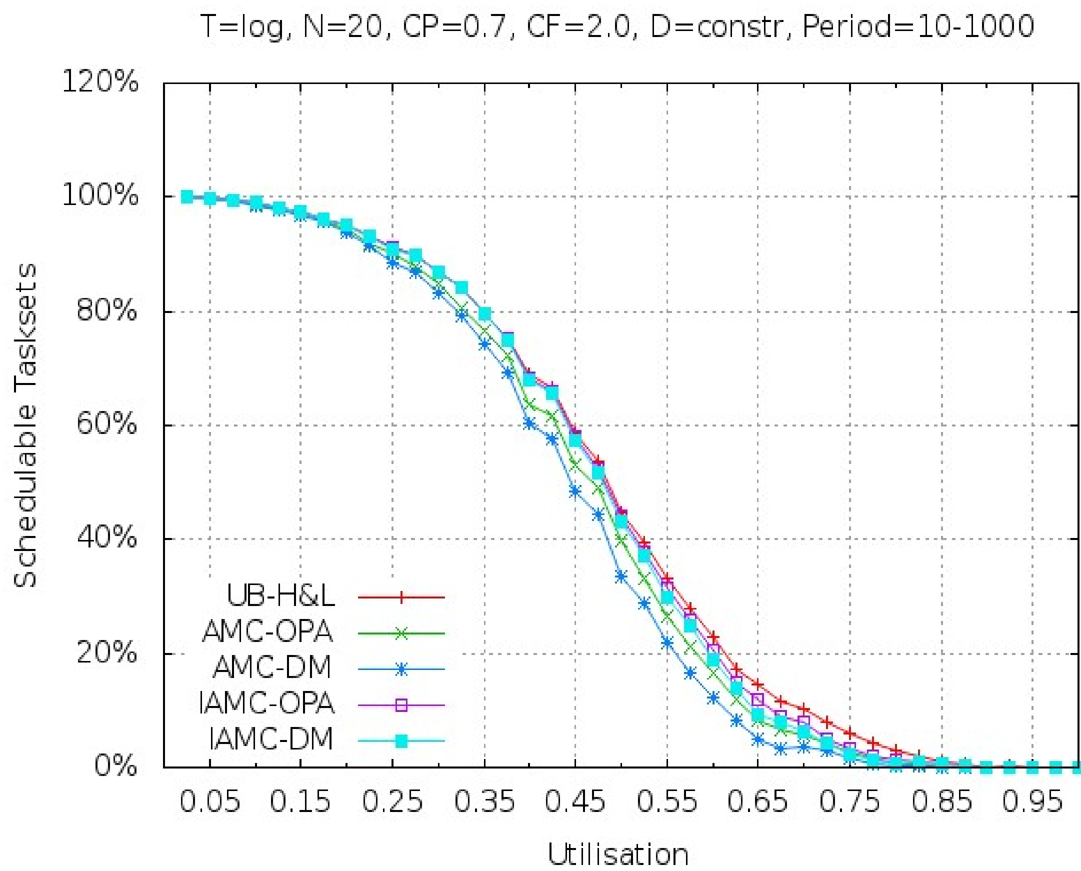


**Figure 6.3:** Higher criticality probability

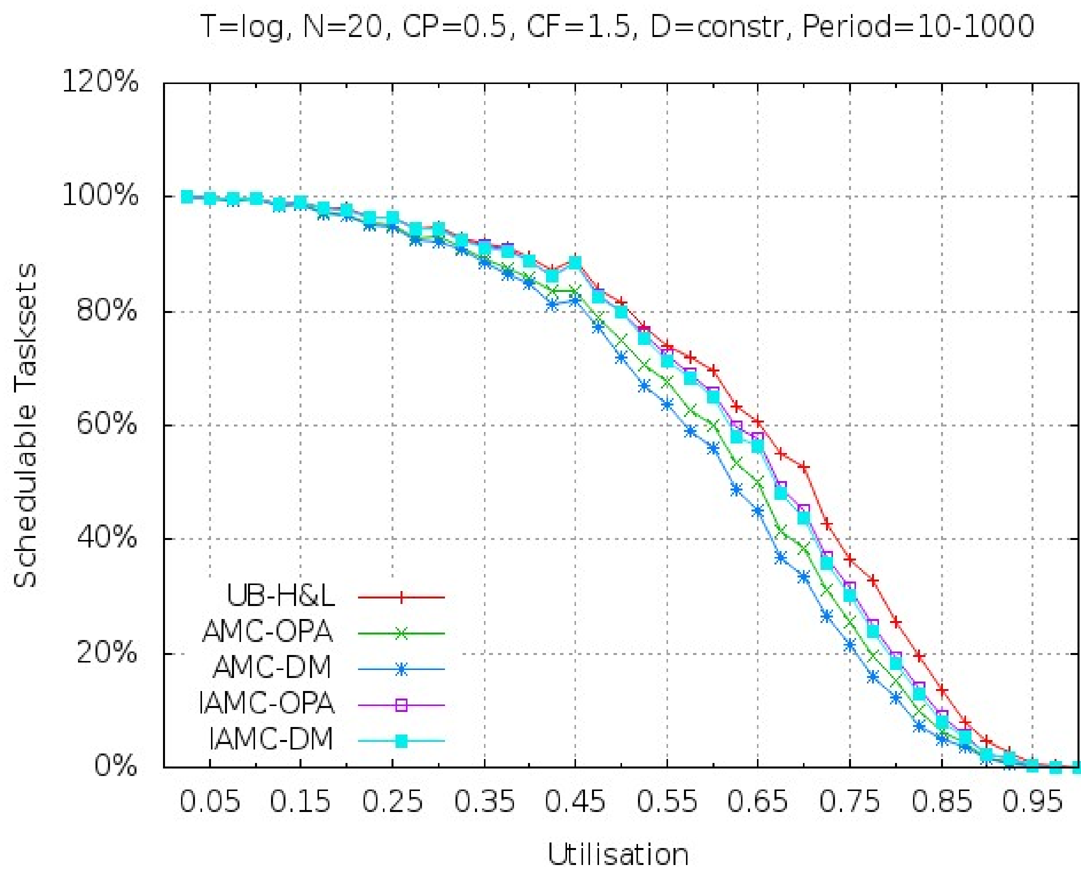


**Figure 6.4:** Higher criticality factor

So far, all figures show the same trends. However, the next two figures show a slightly different trend.

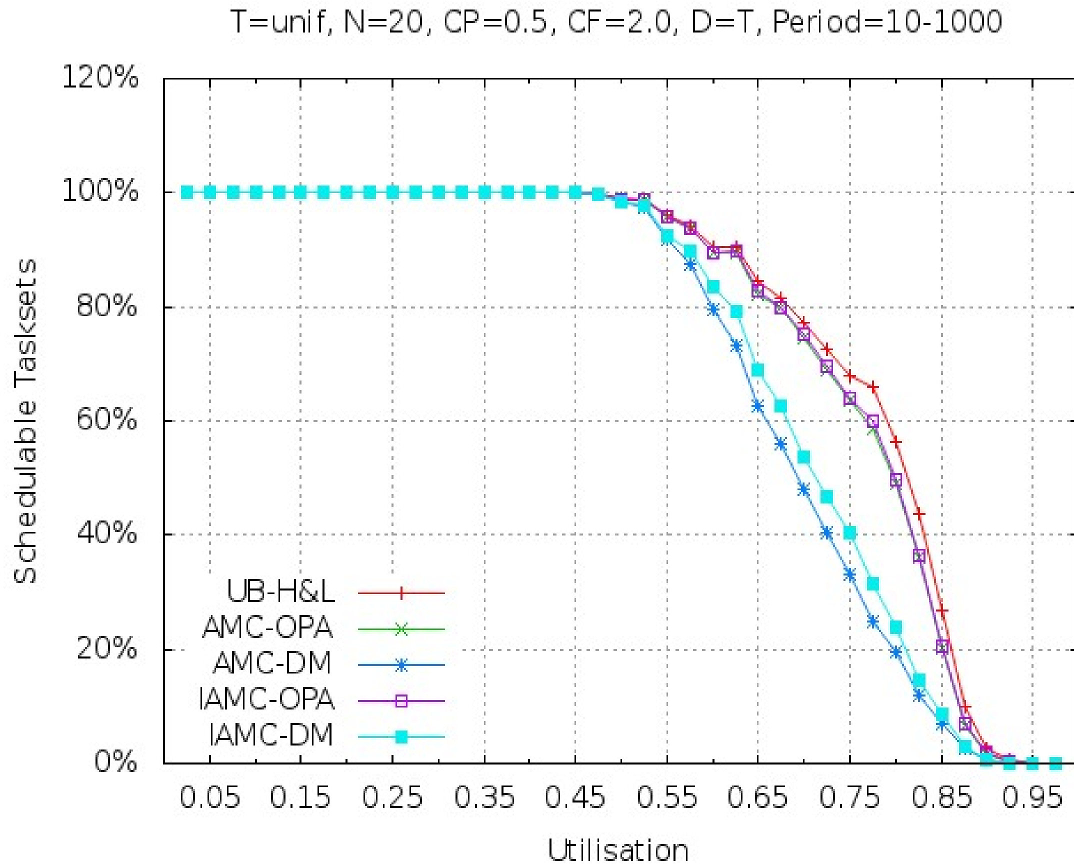


**Figure 6.5:** Periods with a regular uniform distribution

Figure 6.5 depicts the results with $N = 20$, $CP = 0.5$, $CF = 2$ and constrained deadlines, but with uniformly distributed periods (Tunif), i.e. there is a much higher chance of high-valued periods than in the case of log-uniformly distributed periods. Clearly this amplifies the impact of the priority ordering, because both AMC-OPA and IAMC-OPA perform much better than their DM counterparts. But more importantly AMC-OPA performs better than IAMC-DM in this case.

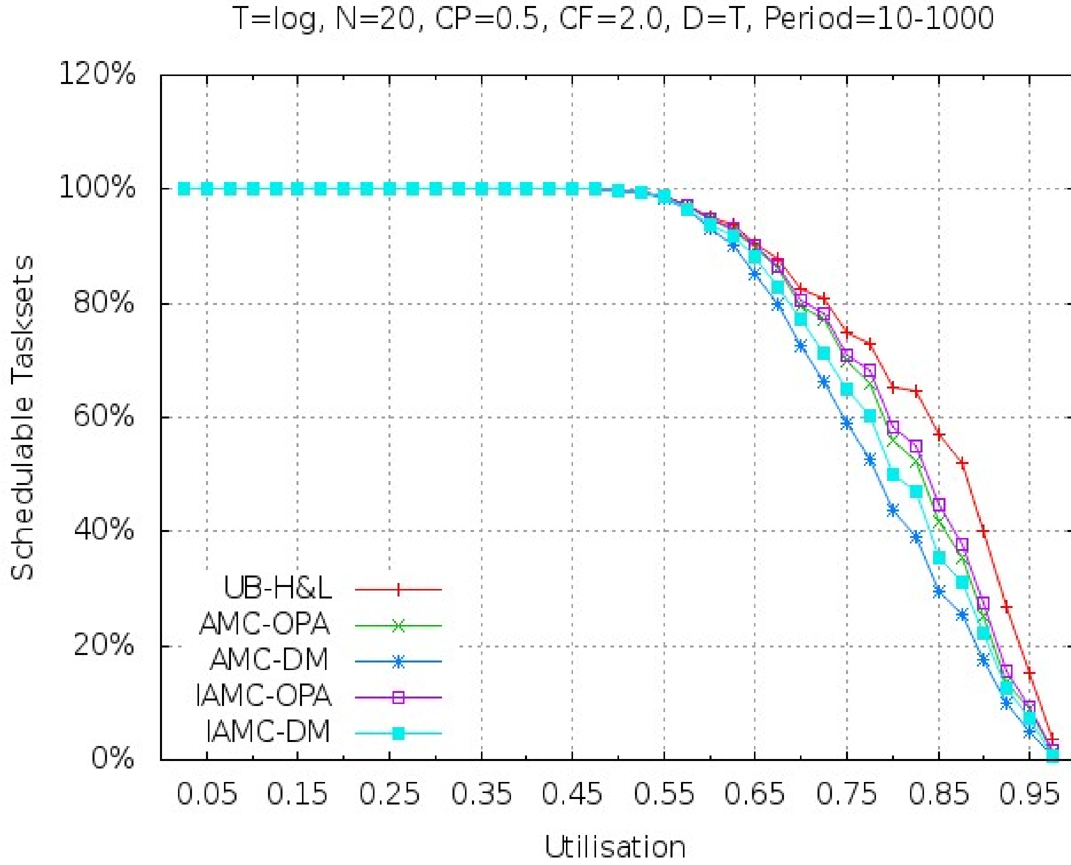


**Figure 6.6:** Implicit deadlines (D = T)

Similarly, figure 6.6 shows the comparison of AMC and IAMC for tasksets with implicit deadlines, and once again AMC-OPA outperforms IAMC-DM, although by a much lower margin than in figure 6.5.

We can draw the following conclusions:

1. For both AMC and IAMC the priority ordering OPA leads to better schedulability than DM. This clearly shows that the priority ordering is an important factor for improving schedulability.
2. Comparing IAMC-OPA with AMC-OPA and IAMC-DM with AMC-DM yields that IAMC performs better than AMC.
3. Finally, in many cases even IAMC-DM performs better than AMC-OPA. We can conclude that the schedulability test is at least as important as the priority ordering.

# 7

# Evaluation of IAMC for Multiprocessors

THE goal of this chapter is to present and evaluate a number of heuristics and parameters which play a role in mixed-criticality fixed-priority partitioned scheduling. We will use the previously described schedulability test IAMC to test a large number of parameters, in order to identify the best combination of heuristics for this test.

Since it has been previously shown that IAMC performs better than AMC, only IAMC will be considered in this chapter.

## 7.1 Heuristics

Two sets of heuristics will be considered. The first heuristic concerns the order in which the processors are selected for testing whether another task can be added. This correlates to the classic bin-packing problem. The second heuristic concerns the initial ordering of the taskset, that is, the order in which the tasks will be selected for schedulability testing on each processor.

Subsequently we will describe each of the heuristics in detail.

### 7.1.1 Processor Selection Heuristics

The following bin-packing heuristics have been considered to guide the processor selection:

- **First Fit (FF)**: The processors are traversed in a fixed order, and the selected task is assigned to the first processor in which it fits according to

the used schedulability test.

- **Best Fit (BF)**: The processors are ordered according to lowest remaining capacity. Processor capacity $\Omega_m$ is defined as $\Omega_m = 1 - \sum_{\tau_i \in \Gamma_m} C_i/T_i$. In other words, the processor with the highest utilisation is tested first, then the one with the second highest utilisation, etc.
- **Worst Fit (WF)**: The opposite of BF, the processors are ordered according to highest remaining capacity.

### 7.1.2 Initial Orderings

The following ordering heuristics have been evaluated:

- **Decreasing Utilisation (DU)**: The tasks are ordered by decreasing utilisation $U_i(\texttt{LO})$, where $U_i(\texttt{LO}) = C_i(\texttt{LO})/T_i$.
- **Deadline Monotonic (DM)**: The tasks are ordered by increasing deadlines.
- **Criticality Monotonic (CM)**: The tasks are ordered according to decreasing criticality levels $L_i$, where $\texttt{HI} > \texttt{LO}$. In case of equal criticality levels, deadline monotonic ordering is used.
- **Criticality Utilisation (CU)**: The tasks are ordered according to decreasing criticality levels $L_i$, where $\texttt{HI} > \texttt{LO}$. In case of equal criticality levels, decreasing utilisation is used. In [KAZ11] Kelly et al. named this Decreasing Criticality (DC).
- **Slack Monotonic (SM)**: The tasks are ordered by increasing slack, where the slack $S_i$ is computed as follows: $S_i = T_i - D_i$.
- **Criticality Slack Monotonic (CSM)**: The tasks are first ordered according to their criticality level $L_i$, and then according to their slack $(T_i - D_i)$.
- **Random (RAND)**: The tasks are not ordered at all, the taskset stays in the order in which it was input to the bin-packing algorithm.

## 7.2 Experimental Setup

The experimental setup is the same as in [Pat12]. With respect to the uniprocessor experiments in section 6.1, the only differences are that the task periods now range

from 1ms to 1000ms, and that the *UUniFast-Discard* algorithm [DB11] is used, instead of *UUniFast*.

*UUniFast-Discard* is an adaption of *UUniFast* which also works for taskset utilisation levels greater than 1. However, no single task may have a utilisation greater than 1, otherwise it would automatically be unschedulable. So in order to guarantee that all tasks fit on at least one processor, the generated task utilisations are checked for this condition. If during the generation phase a task with a utilisation greater than 1 is created, the whole taskset is discarded, and the taskset generation algorithm simply begins anew.

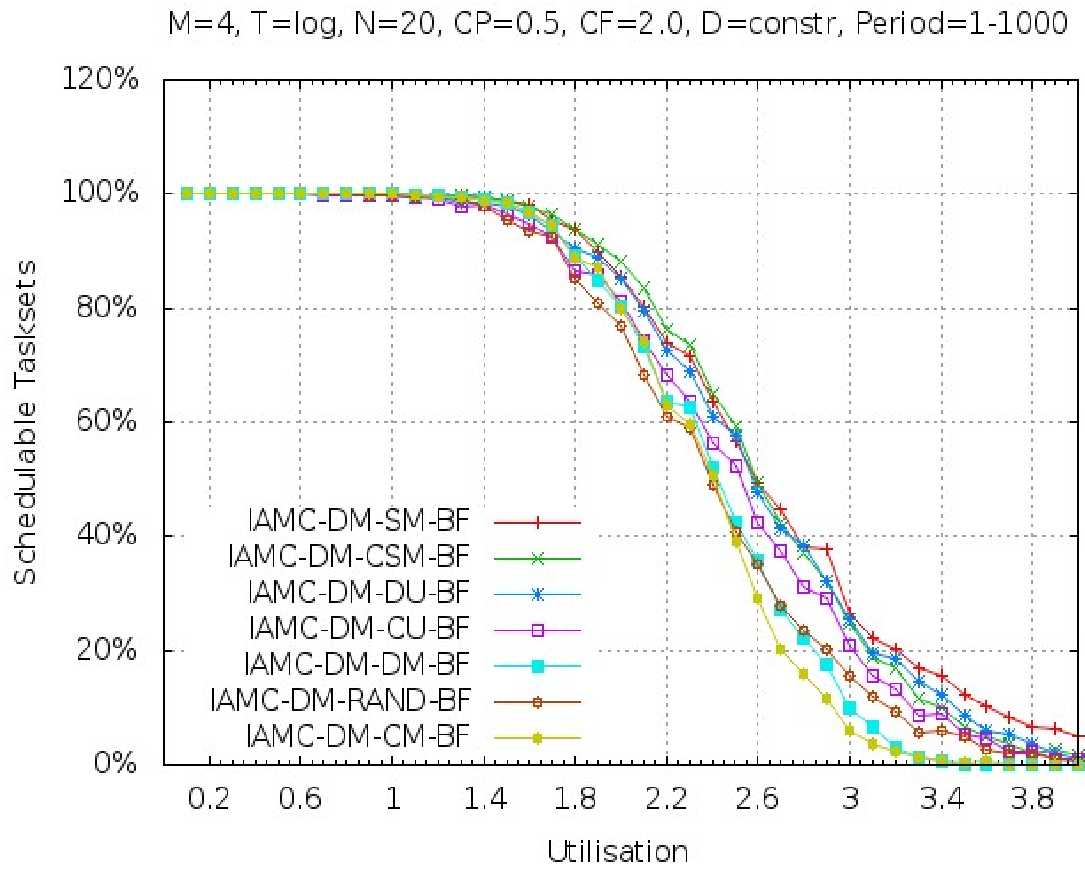


**Figure 7.1:** IAMC-DM-*-BF

## 7.3 Experiments

This section is divided into two parts. First, the best combination of initial task ordering, priority ordering and bin-packing algorithm will be identified. In the second part, that best combination will be investigated for its scalability properties.

The criticality probability $CP$ and the criticality factor $CF$ were kept steady at 0.5 and 2.0 respectively, because their impact is quite predictable and thus uninteresting.

### 7.3.1 Comparing The Heuristics

In the following a number of exemplary graphs are depicted. They have been chosen out of literally hundreds of plots to show the trends of the results. The lines of the plots are labelled as 4-tuples, e.g. IAMC-DM-SM-BF. The first part represents the uniprocessor schedulability test which was used, the second part stands for the priority ordering used by the test, the third part is the initial ordering that determines the task input order to the bin-packing algorithm and the last part indicates the bin-packing algorithm itself. An asterisk (*) at any position means that this is the parameter whose variability is depicted in the plot. The header of the plots is the same as the one described in chapter 6, except that there is an additional parameter $M$, which denotes the number of processors which have been used in the analysis. It also indicates the total utilisation level.

Figure 7.1 gives a first overview over the different initial orderings which can be chosen. It shows the results of all seven initial orderings when the IAMC schedulability test is used together with DM priority ordering and BF bin-packing on a system with 4 processors. It can clearly be seen that for high utilisations, IAMC-DM-SM-BF is able to schedule the most tasksets. There is also considerable spread in the performance of the different initial orderings.

Figure 7.2 shows a similar plot, but with OPA as the priority ordering for the test instead of DM. Interestingly the overall schedulability went down in comparison to the previously examined IAMC-DM-SM-BF. It is also much harder to distinguish the different initial orderings, since their performance in terms of schedulability is quite similar. Clearly the use of OPA as priority ordering diminishes the impact of the initial ordering considerably. This is a result which is symptomatic for all of the experiments.

The performance of FF is, on average, slightly worse than the performance of BF. An example is depicted in figure 7.3, IAMC-DM-*-FF. In this case IAMC-DM-DU-FF seems to perform better for most utilisations, although IAMC-DM-SM-FF catches up to it at the very end.

More interesting is the comparison of the initial orderings with the WF heuristic, as depicted in figures 7.4 and 7.5 for DM and OPA respectively. It is obvious

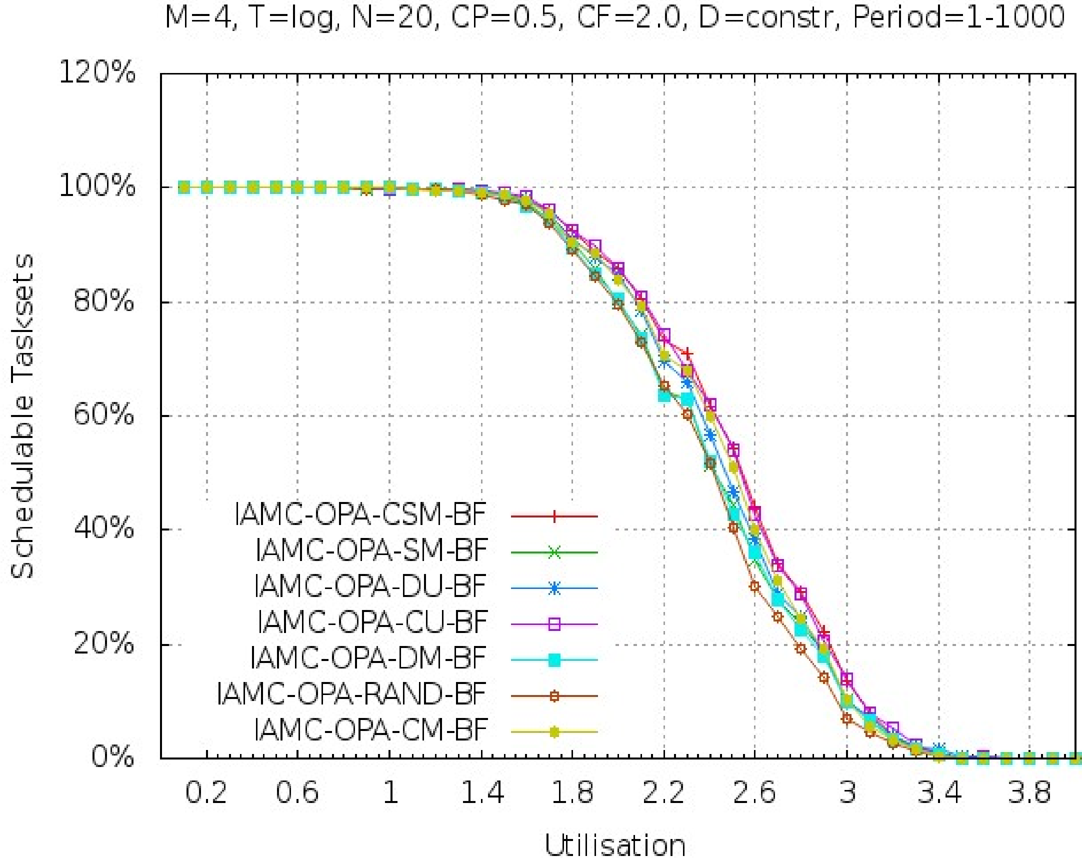


**Figure 7.2:** IAMC-OPA-*-BF

that in both cases DU and criticality utilisation (CU) perform much better than all other initial orderings. However, as we will see later, the combination of WF and DU does not perform better than the combination of BF and slack monotonic (SM).

The previously mentioned relationships of BF, FF and WF can also be seen in the figures 7.6 and 7.7. The three bin-packing heuristics are compared on the base of SM as initial ordering, and it is easy to see that BF gives the best schedulability while WF is much worse.

It is also worth mentioning that not ordering the taskset before task allocation, i.e. keeping it random, performs worse than the best heuristics, but not significantly. In fact the option of keeping random ordering performs better than several other initial ordering heuristics. In the case of offline partitioning this is not very interesting because pre-ordering the taskset has no cost, but in online scheduling algorithms the cost of keeping the ready-queue in an ordered state is significant.

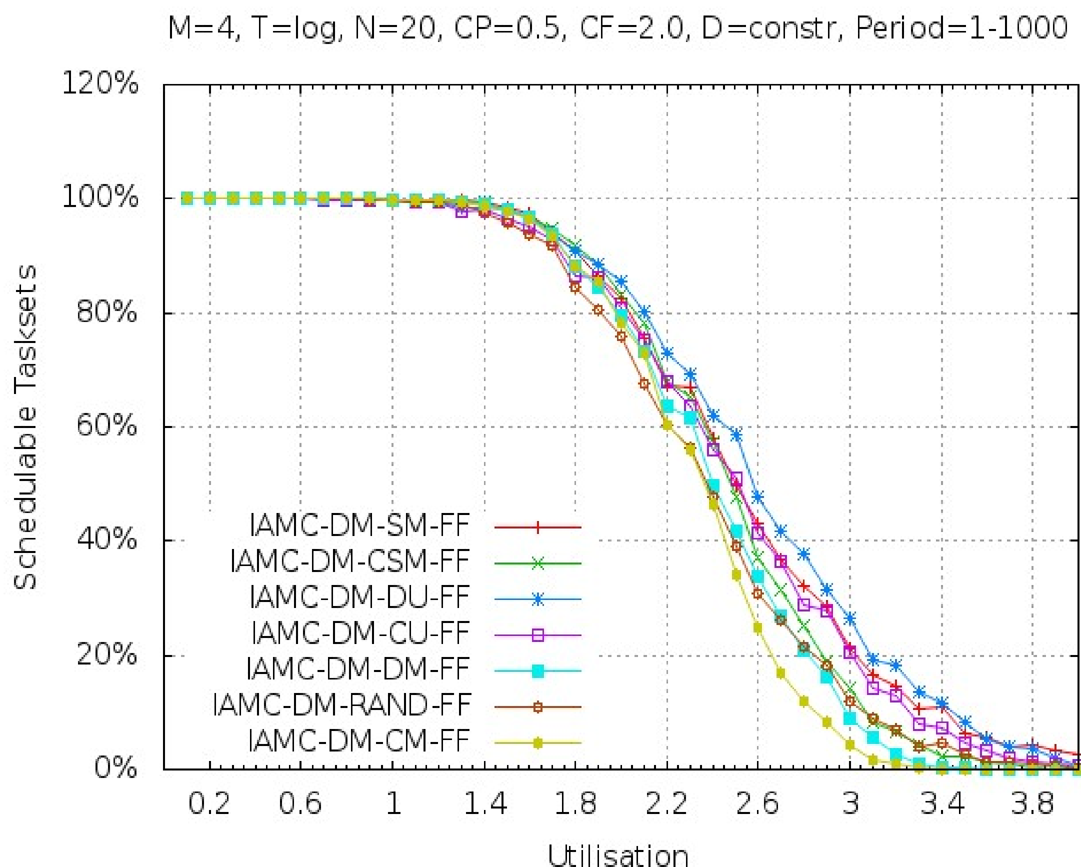


**Figure 7.3:** IAMC-DM-*-FF

Only selected examples out of the huge amounts of data have been shown, but the overall trend should be clear from these. IAMC-DM-SM-BF is by far the best combination and is able to schedule more tasksets than any other tested combination.

Especially the fact that the deadline monotonic priority ordering outperforms Audsley's OPA is rather surprising, and somewhat counter-intuitive. But it is important to remember that OPA is only optimal for uniprocessor systems, and only when a exact schedulability test is used, neither of which is the case here.

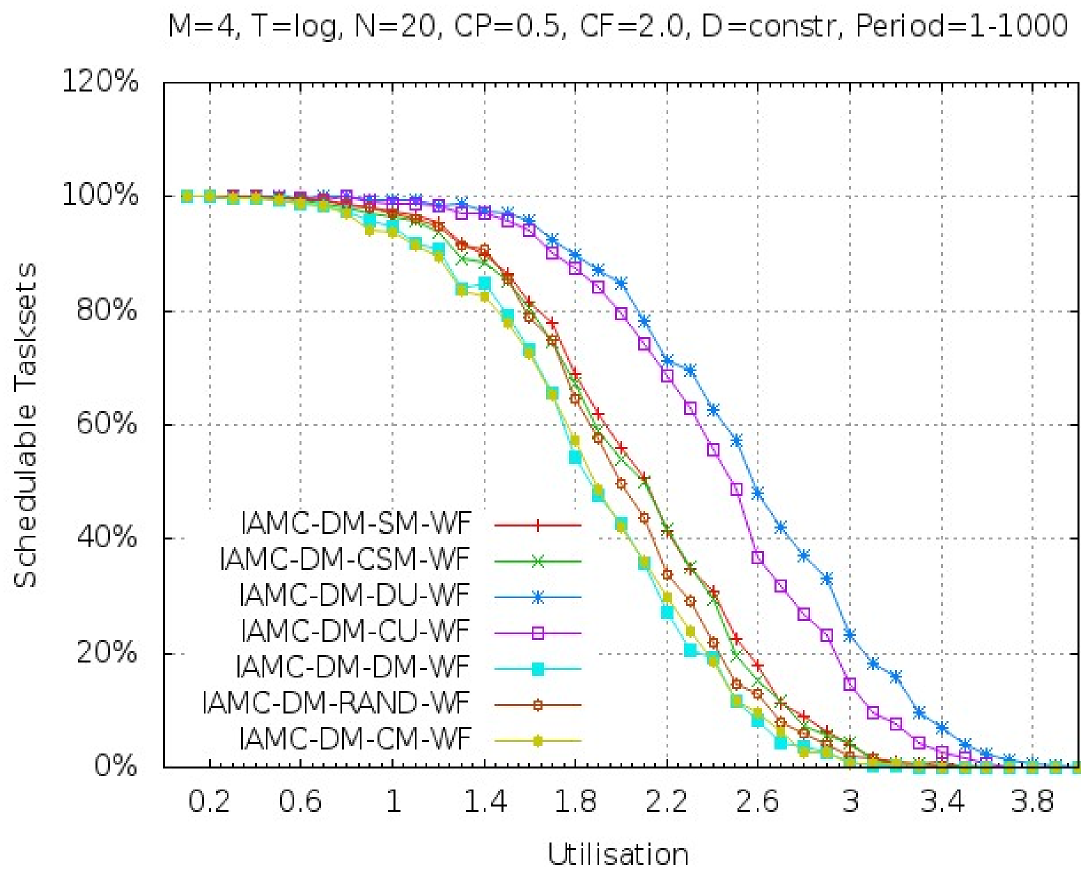


**Figure 7.4:** IAMC-DM-*-WF

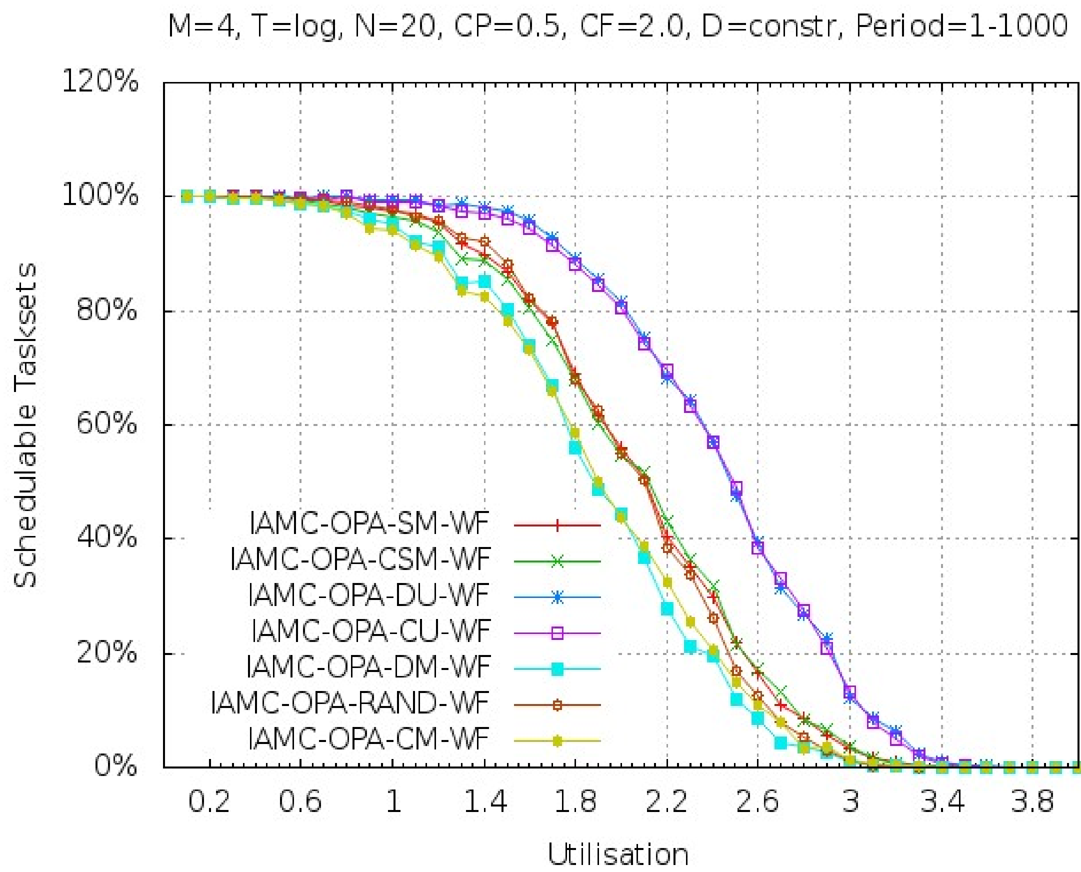


**Figure 7.5:** IAMC-OPA-*-WF

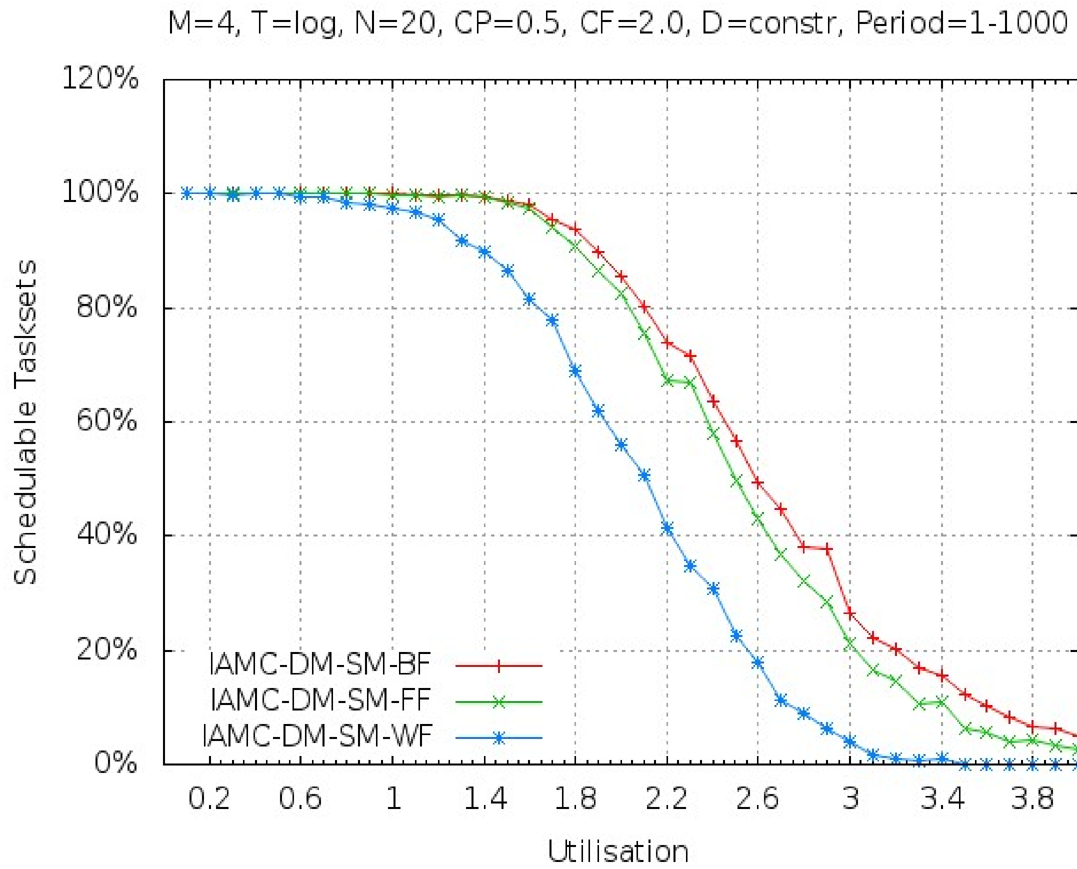


**Figure 7.6:** IAMC-DM-SM-*

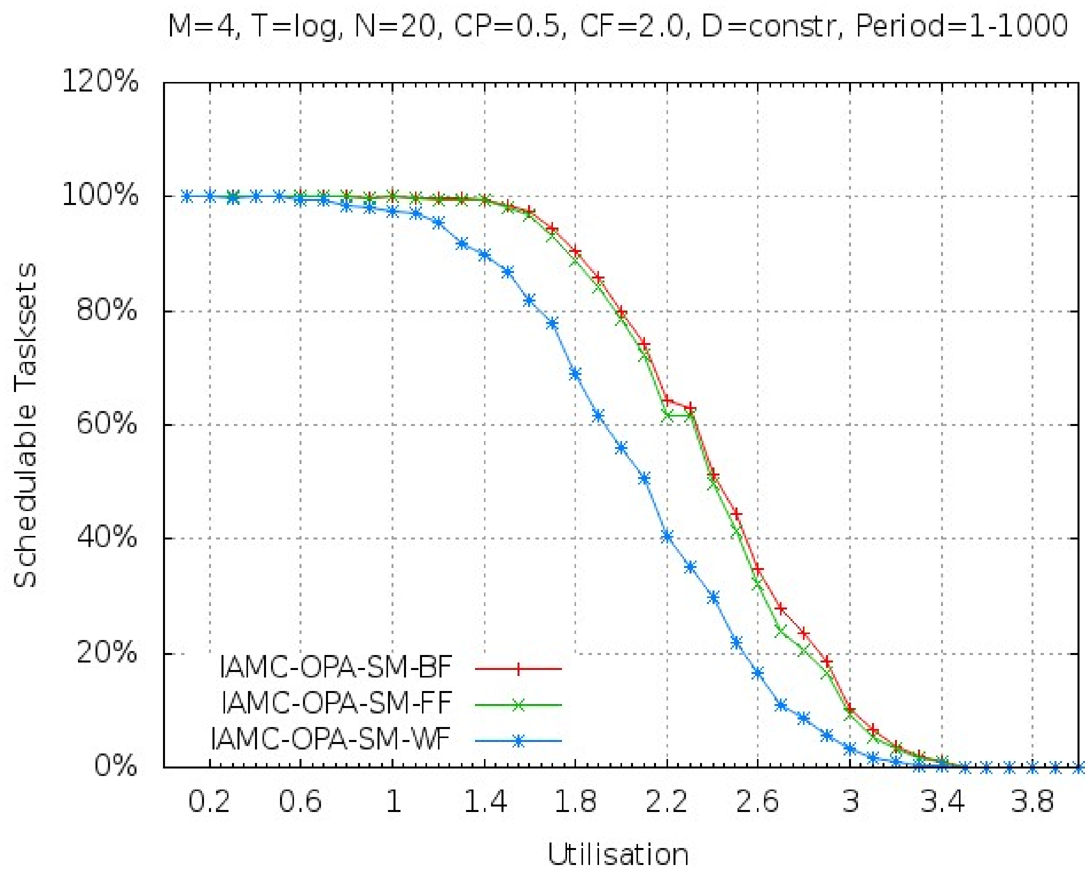


**Figure 7.7:** IAMC-OPA-SM-*

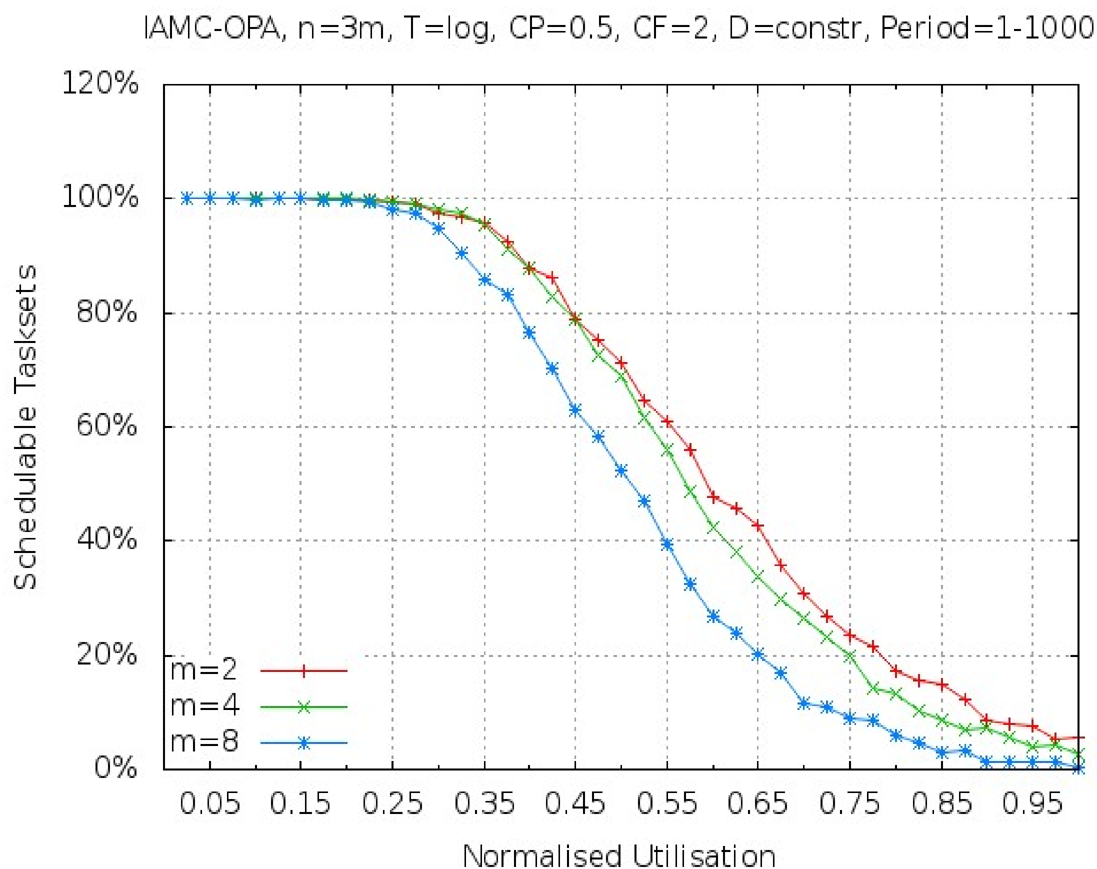


**Figure 7.8:** M*-NM3-IAMC-DM-SM-BF

### 7.3.2 Scalability of IAMC-DM-SM-BF

In the following the scalability properties of IAMC-OPA-DU-WF will be analysed. This is done by considering different numbers of processors and task set sizes, i.e. varying values of $m$ and $n$ . The number of processors $m$ ranged from 2 to 8 in powers of 2, and the number of tasks per taskset was scaled by $m$, with scaling factors of 3, 5, and 10 respectively.

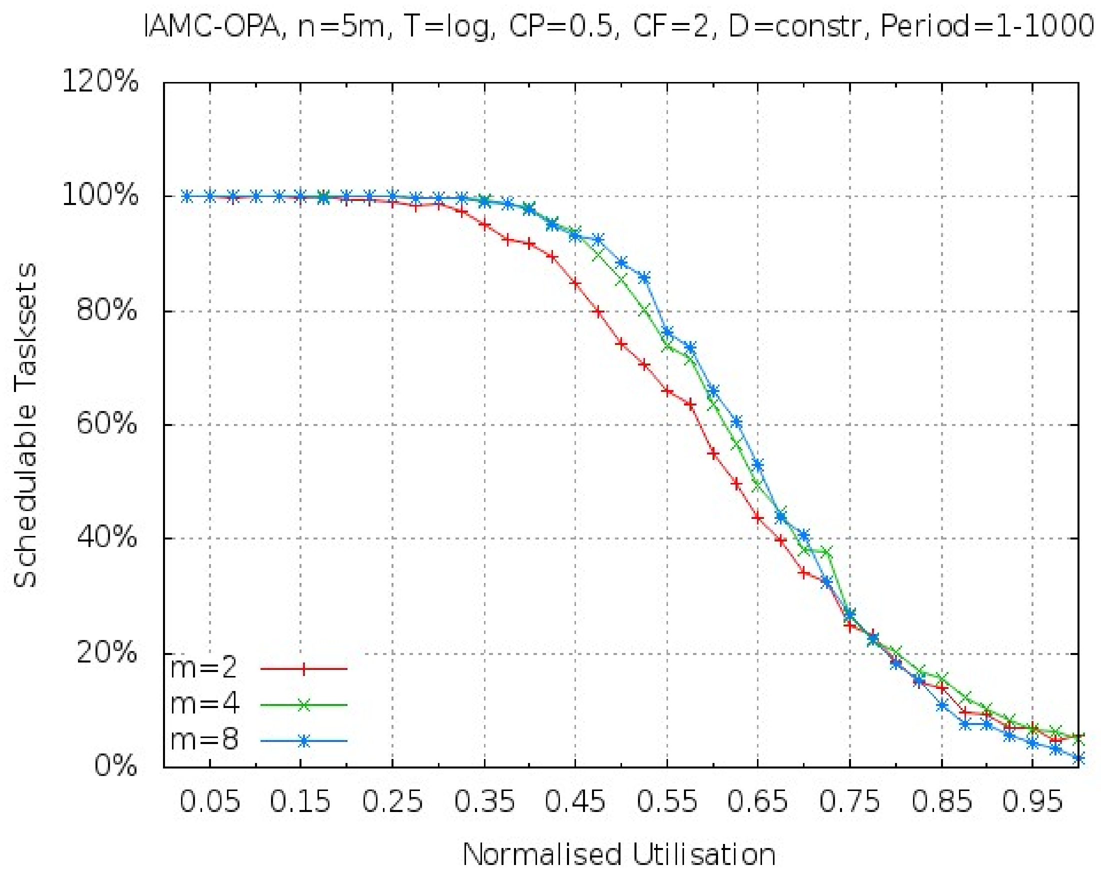


**Figure 7.9:** M*-NM5-IAMC-DM-SM-BF

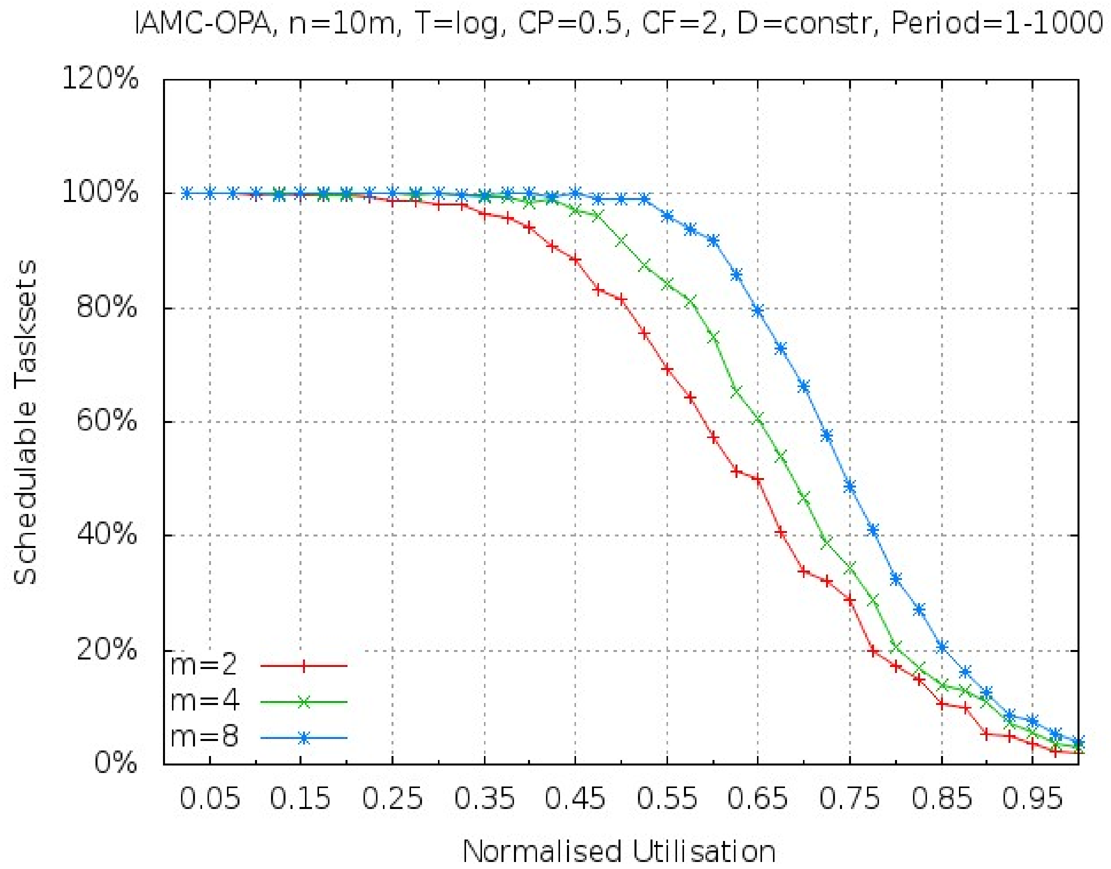


**Figure 7.10:** M*-NM10-IAMC-DM-SM-BF

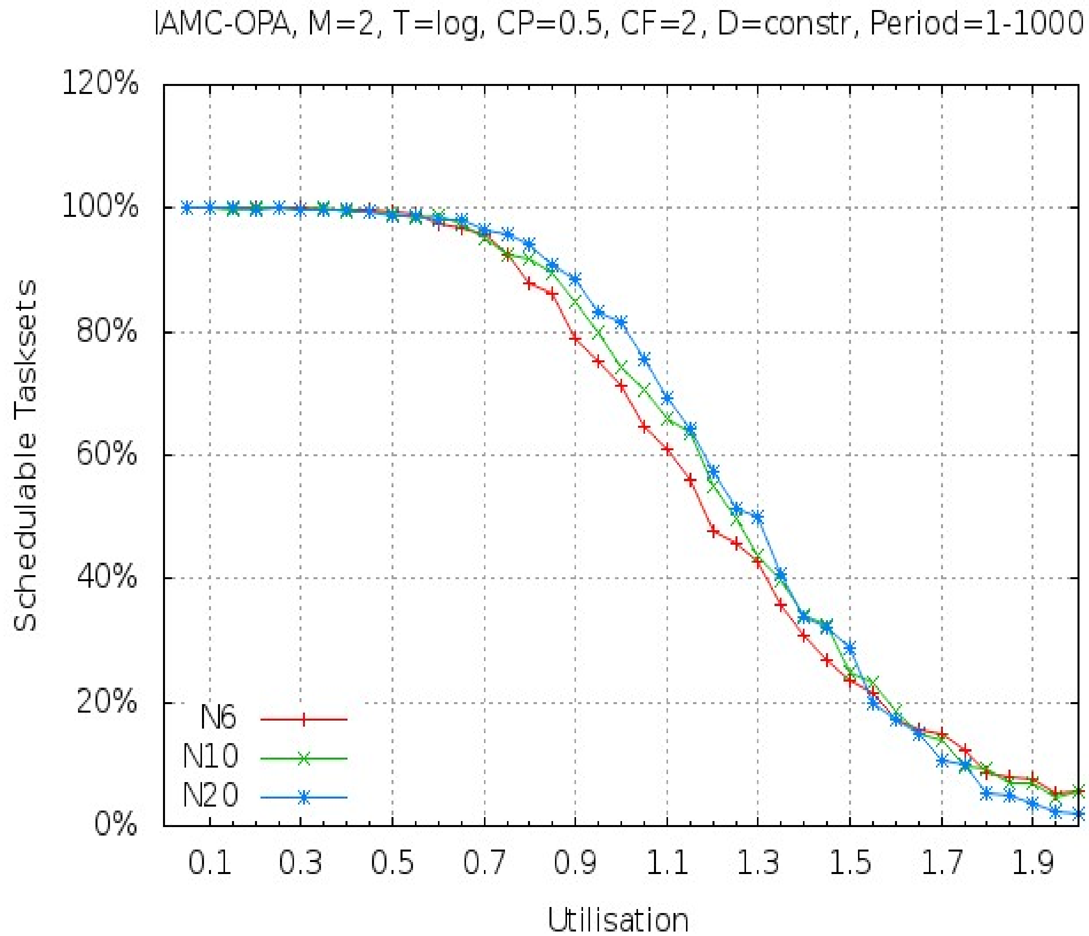


**Figure 7.11:** M2-N*-IAMC-DM-SM-BF

### Varying The Number Of Processors

Figure 7.8 shows the normalised utilisation for 2, 4, and 8 processors with a taskset size of $n = 3m$. The trend is clear, with increasing number of processors, the achieved utilisation is decreasing.

Figure 7.9 depicts the same, but for a taskset size of $n = 5m$. This figure is much less clear, at higher utilisation levels the systems with more processors are able to schedule more tasksets successfully.

Finally, figure 7.10 depicts this relationship for a taskset size of $n = 10m$. This time the results show in the opposite direction, the systems with more processors are able to schedule more tasksets.

This would indicate that IAMC indeed scales quite well, the more processors

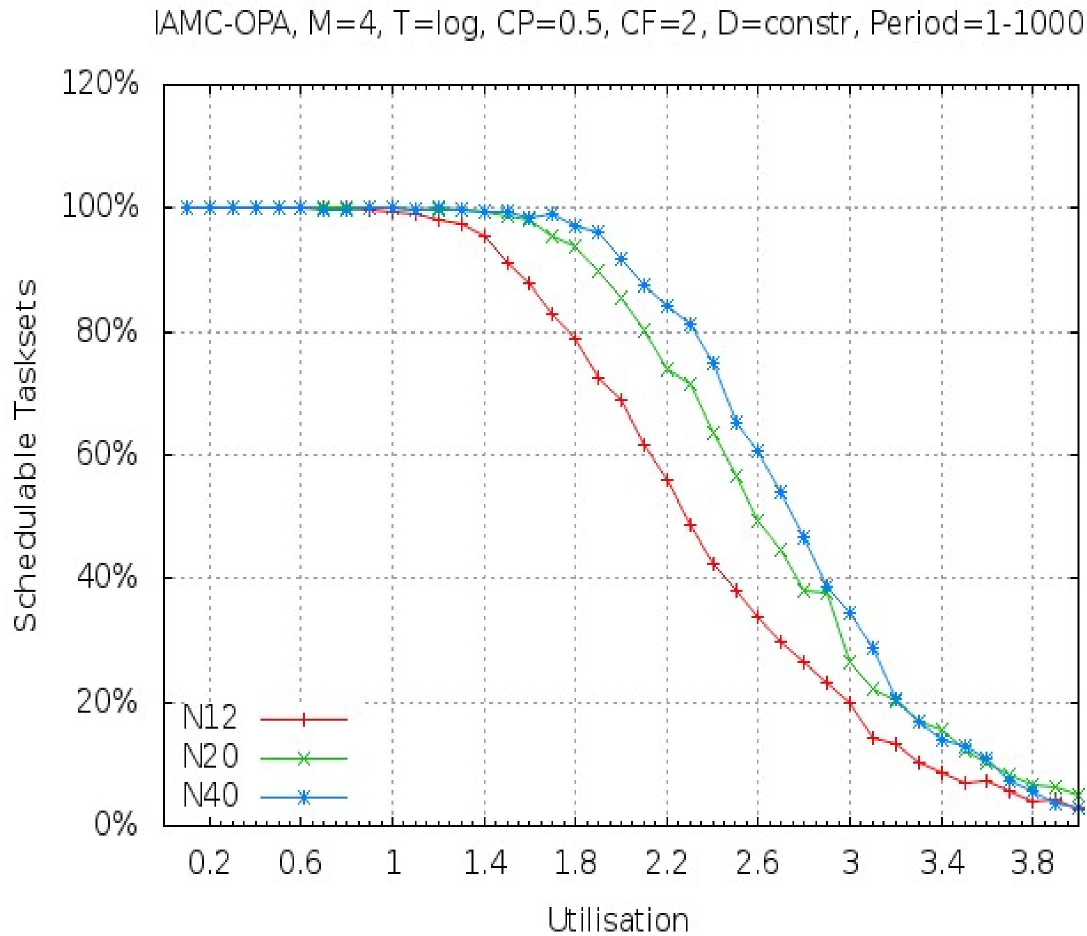


**Figure 7.12:** M4-N*-IAMC-DM-SM-BF

the better for the schedulability, if the taskset is "flexible" enough. But more experiments with a higher number of processors and a higher number of tasks per taskset might be needed in order to show if this trend holds.

### Varying The Number Of Tasks Per Taskset

When varying the number of tasks per taskset there is definite expectation that more tasks per taskset lead to higher schedulability. There are more possibilities to distribute the tasks over the processors. The intuition is that a higher number of smaller packets should be easier to pack than a smaller number of bigger packets.

Figure 7.11 depicts the impact of varying the number of tasks per taskset for 2 processors. This plot is not very clear, and it seems logical to assume that 2

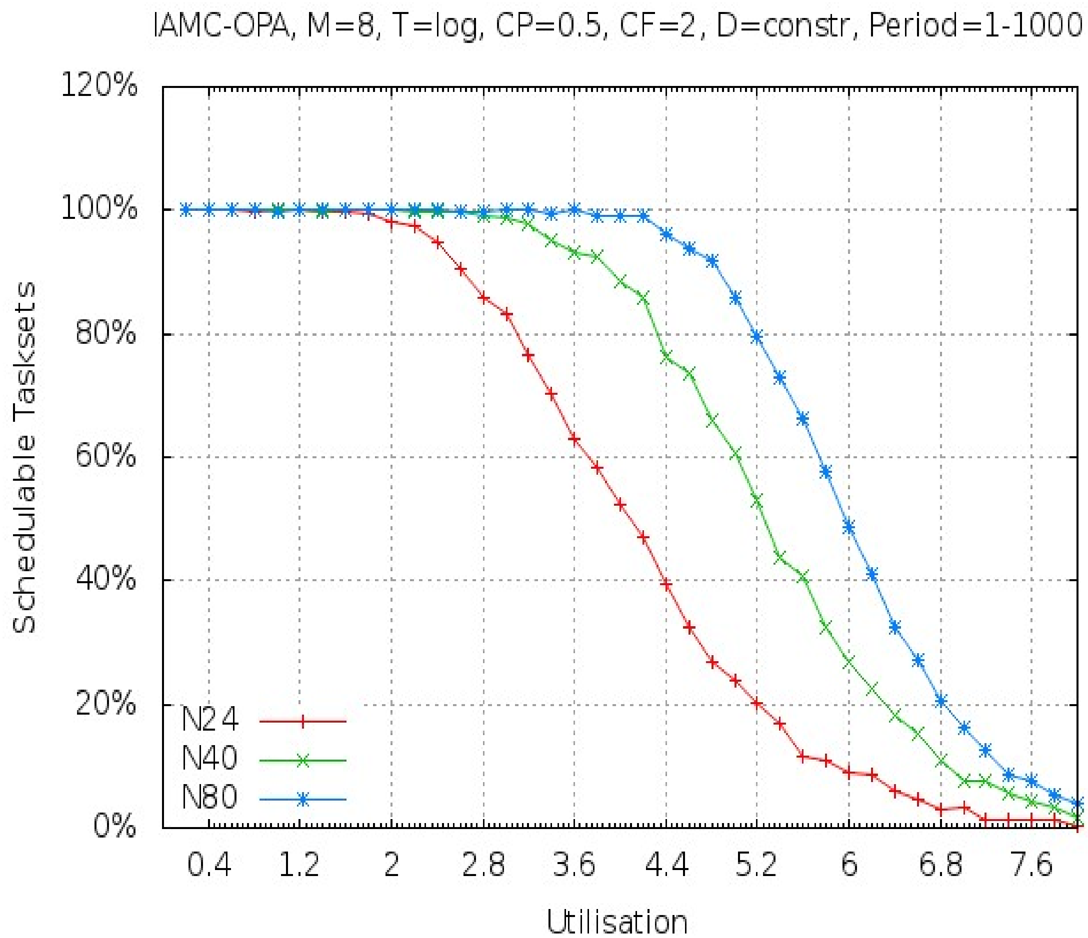


**Figure 7.13:** M8-N*-IAMC-DM-SM-BF

processors with the tested number of tasks is not sufficient to show a clear trend.

Figure 7.12 depicts the same for 4 processors. Here the expected trend is already visible, albeit not perfectly clear either.

Figure 7.13 depicts the situation for 8 processors. By now there can be no doubt that a higher number of tasks per taskset leads to better schedulability.

We conclude that a higher number of tasks with smaller utilisation values per task has a positive impact on schedulability as expected.

# 8

# Conclusions

The goal of this work was twofold. The first goal was to present and evaluate the new uniprocessor test IAMC for mixed-criticality systems. The second goal was to compare and evaluate a number of heuristics and strategies for partitioned MC multiprocessor systems.

Concerning the first goal, it was shown that IAMC clearly outperforms AMC in uniprocessor systems with constrained deadline tasksets. For systems with implicit deadlines however the improvement is relatively small, albeit still visible. It was also shown that the priority ordering and the schedulability test are equally important in determining the schedulability of a taskset.

With regard to the second goal, a large number of approaches to partitioning were investigated for an equally large number of scenarios. This yielded a number of interesting results. Using a slack-monotonic initial ordering in conjunction with best-fit task allocation and deadline monotonic priority ordering turned out to give the best schedulability for random tasksets. Most surprising is probably that using deadline monotonic priority ordering seems to be better than using Audsley's approach. On the other hand, it has also clearly been shown that worst-fit is a rather bad task allocation algorithm for systems employing an AMC-like run-time system. It is also interesting to see that IAMC seems to scale well with the number of processors, if the number of tasks per taskset is high enough.

There are many directions in which future work on this topic could develop. One obvious choice would be to look into other priority orderings than DM and OPA, for instance some of the ones used in this work as initial ordering heuristics. Another possibility could be to apply data mining techniques to the collected data to see if they can unearth relationships within the randomly generated tasksets which give clues to what makes a taskset easily schedulable.

# Bibliography

## A

[Aud91] Neil C Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Real-Time Systems*, 1991.

## B

[BB05] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[BBD+10] Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. In Petr Hliněný and Antonín Kučera, editors, *Mathematical Foundations of Computer Science 2010*, volume 6281 of *Lecture Notes in Computer Science*, pages 90–101. Springer Berlin Heidelberg, 2010.

[BBD11] Sanjoy K. Baruah, Alan Burns, and Robert I. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43. IEEE, 2011.

[BDNB08] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems*, 39(1-3):5–30, 2008.

[BLS10] S. Baruah, Haohan Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 13–22, 2010.

[BW09] Alan Burns and Andrew J Wellings. *Real Time Systems and Programming Languages: Ada, Real-time Java and C/Real-time POSIX*. Pearson Education, fourth edition, 2009.

## D

[DB11] Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[dNLR09] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 291–300, 2009.

[DRRG10] François Dorin, Pascal Richard, Michaël Richard, and Joël Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46(3):305–331, 2010.

## G

[GESY11] N. Guan, P. Ekberg, M. Stigge, and Wang Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 13–23, 2011.

## J

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

## K

[KAZ11] Owen R Kelly, Hakan Aydin, and Baoxian Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1051–1059. IEEE, 2011.

# L

[LB10] Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 183–192, 2010.

[LdNRM10] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 169–178, 2010.

[Leh87] J. P. Lehoczky. Enhanced aperiodic responsiveness in hard real-time environments. *Proceedings of the IEEE Symposium on Real-Time Systems*, pages 261–270, 1987.

[LGDG03] J.M. López, M. García, J.L. Díaz, and D.F. García. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Systems*, 24(1):5–28, 2003.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982.

# M

[MEA+10] 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 *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1864–1871, 2010.

# P

[Pat12] R.M. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 309–320, 2012.

## S

[SGTG12] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 155–165, 2012.

## V

[Ves07] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239–243. IEEE, 2007.