Multiprocessor Miss Rate Simulator

Mats Haring
Faculty of Computer Science
Technical University of Dortmund
Dortmund, Germany
mats.haring@tu-dortmund.de

Simon Koschel
Faculty of Computer Science
Technical University of Dortmund
Dortmund, Germany
simon.koschel@tu-dortmund.de

Franziska Schmidt
Faculty of Computer Science
Technical University of Dortmund
Dortmund, Germany
franziska3.schmidt@tu-dortmund.de

Jannik Drögemüller
Faculty of Computer Science
Technical University of Dortmund
Dortmund, Germany
jannik.droegemueller@tu-dortmund.de

I. Introduction

In this project we develop a task simulator. This is a useful tool to analyse the performance of different scheduling methods on task sets in a simulated system.

In the practical case of planning an embedded system, the performance can easily be tested for different scheduling algorithms and different hardware (e. g. amount of processor cores). The amount of tasks and the assumed utilization can be estimated to fit the needs of the system, so that the simulation is realistic. The estimated task set can then be used as an input for the simulator.

Instead of building a prototype for testing the performance, the simulator can make the testing and research for the optimal hardware components easier and cheaper.

The goal of this project is to build a multiprocessor miss rate simulator in Python 3 that will be able to simulate different scheduling algorithms. This work is based on a uniprocessor miss rate simulator, that was published in [2] and is further described in section II.

II. Preconditions

We were given a fully implemented miss rate simulator written in Python 2.7 that was able to handle the simulation of a single core processor. In section II we will further explain the preconditions of this project. Therfore, we will first explain the task and scheduling model and then move forward to the execution time model. After that, we define the term deadline miss rate and in the end we explain the structure of the given simulator.

A. Task and scheduling model

The simulator uses a soft real-time task system. This means that tasks can miss their deadline frequently and the results of the tasks would still have a value, especially not zero. The system has got n independant periodic or sporadic tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ and works with a uniprocessor. Since the priority of tasks could not be changed during runtime, there was only a preemptive fixed-priority scheduling policy implemented. Tasks are sorted by their period and indexed from 1 to n with τ_1 having the highest and τ_n having the lowest priority. With this setup the preemptive, fixed-priority rate-monotonic (RM) scheduling policy is implemented as the standard scheduling algorithm.

Jobs define the instances of a task and they are being released not before the time period T_i, that is set for a task τ_i individually, has passed. This means, that there is at least a time interval of T_1 after the release of a job, where no job of τ_i can be released. Every task τ_i also has a relative deadline D_i. A job of task τ_i has to be completed before a time interval of D_i has passed. We can differentiate between two types of task sets [10]:

1) *Implicit-deadline* task sets, where for every task τ_i the deadline D_i equals its period T_i.
2) *Constrained-deadline* task sets, where for every task τ_i the deadline D_i is lower or equal to its period T_i.

B. Execution time model

To simulate a realistic real-time task system, an execution time model was used, where every task τ_i has several values of execution time $C_{i,j}$ and each of them

is associated to a probability $P_{i,j}$ of this execution time. In this case, there are two different execution times for each task: "normal" execution time C_i^N of a task τ_i, when no faults occured and "abnormal" execution time C_i^A, when a fault was detected.
The "normal" execution time C_i^N already includes time for the detection of a fault, which is assumed to perform perfectly and is done at predefined checkpoints or the end of a job execution.
The probability for an abnormal execution P_i^A for a task τ_i is applied independantly from previous errors and executions and equals the probability of a fault for this task.

C. Deadline miss rate

For a schedule of a sequence of jobs of task τ_k the deadline miss rate describes the percentage of jobs of task τ_k missing their deadline in the schedule. Formally:

Definition 1 (Miss Rate): The miss rate of a task $\tau_k \in \Gamma$ for a given schedule S is the number of jobs missing their relative deadline D_k in S divided by the number of released jobs of task τ_k in S. [10]

The expected miss rate, however, is defined for a single task τ_k, not a schedule.

Definition 2 (Expected Miss Rate): The expected miss rate of a task $\tau_k \in \Gamma$, denoted by E_k, is the probability that a job of τ_k misses its deadline. [10]

D. Architecture of the simulator

There is an overview of the architecture of the simulator given in figure 1. For each task τ_i there are two types of events, `release` and `deadline`. The release event adds new workload to the entry of its task τ_i in the status table and also places a `deadline` event for τ_1 into the event list. The deadline event for τ_i checks if the current job of the task has missed its deadline or not by checking the remaining workload of task τ_i in the status table. If it is zero, then the job has met its deadline and if it is higher, then there is still workload left and the job has missed its deadline.

The main components of the simulator are defined in [10] as follows:

- **Task generator:** Builds a set of tasks with a given total utilization. Therefore it uses the UUniFast algorithm [9] and the suggestion from Davis et al. in [11]
- **Dispatcher:** dispatches events from the event list and stops when the number of released jobs of the lowest priority task equals to the desired number of jobs.
- **Event list:** keeps track of all the events in the task system. When a new event is inserted by another `release` event, the list is being sorted by the occuring time of the events.
- **Status table:** records the number of deadline misses, number of released jobs, the number of deadlines and the currently remaining workload of a task.

In this model, jobs are never aborted. If a job misses its deadline, the simulation does not get stopped and the remaining portion of execution time is still executed at the same priority level, before the next job can be executed. Whenever the dispatcher gets a new event, the workloads of the tasks in the status table are updated by the time from the last event to the current one and the processor gets reassigned to the task that currently has the highest priority. If there is no task to be executed, the processor runs idle until the next event.

III. Approach

In section III we will explain the approach to achieve our goals we defined in section I. The first step was to convert the given simulator from an outdated Python 2 version to the currently supported Python 3.6 version. The next step was to extend the given uniprocessor simulator into a multiprocessor simulator. Therefore we adjusted datastructures, the taskgeneration, the miss rate simulation and implemented the two types of multiprocessor scheduling global and partitioned. The last step was the implementation of different scheduling algorithms.

A. Conversion from Python 2 to 3.6

The Python Software Foundation has its own library for the translation of Python 2 source code into the current version. The library is called lib2to3. "2to3 is a Python program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code." [6] We used this library with default configuration to convert the given source code into Python 3.6 code. After using lib2to3 we still had some easy to solve compile errors that will be discussed in section IV.

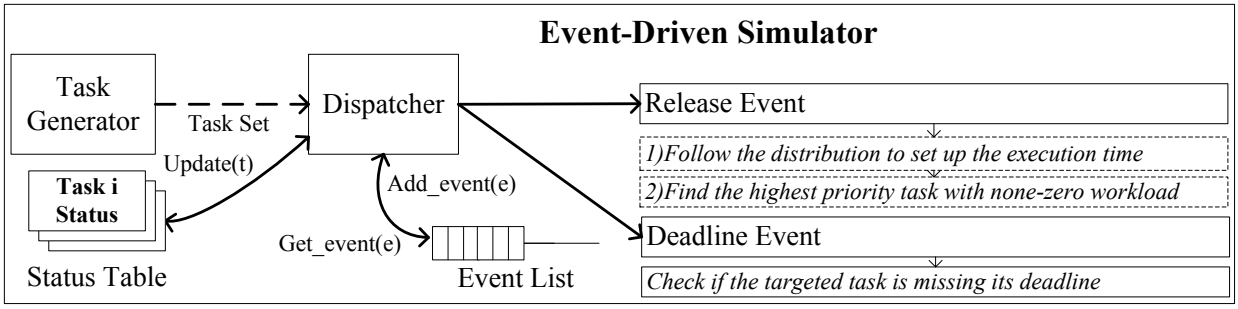

Figure 1: Given simulator [10]

B. Uniprocessor to multiprocessor environment

To update the given uniprocessor miss rate simulator to a multiprocessor simulator, we need to adjust and add several features to the program. In the following, we will give a detailed overview of the changes we made, beginning with basic adjustments, followed by the taskgeneration and the special features of multiprocessor scheduling.

A fundamental change was the implementation of the class 'Task'. Beside of more clearly arranged code, we needed an opportunity to differ between the workload of different jobs that were released by the same task, because the 'statusTable' only gave an overview of the total workload for each task. An object of the class 'Task' consists of an ID, a period, a deadline beginning from the current time, an execution time and an abnormal execution time in case of an incorrect job execution. So that a task object can manage a list of active jobs, we implemented a nested class 'Job' that defines its own deadline and its workload left for execution.

With the insertion of the new datastructure, we needed to adjust several parts of the simulator. Mostly we retrieved data from the task object instead of the 'statusTable', for example in the handling of deadline events, where we implemented another feature - stopOnMiss - to increase the performance of the simulator by stopping the simulation after the first registered deadline miss, that indicates an unschedulable taskset. Therefore we do not need to simulate the taskset no longer.

A very important step to update the simulator was to execute multiple jobs at the same time. Therefore we updated the 'elapsedTime'-function that processed the task execution until the next event has to be handled. The multiprocessor version uses multiprocessor scheduling to get access to the tasks that will be executed in this step. In order to prevent one processor from running idle we need to update the data as soon as one of the following two cases occurs. The first case is, that the time difference to the next event is higher than the workload of one executing job. In this case we need to update the workloads of the executed jobs and the left time difference to the next event. After that we use the scheduling algorithm again to get access to the next tasks until the second case occurs. In the second case, the time difference to the next event is less than all the executing jobs workloads. In this case we are also updating the data, but we first have to handle the next event before the next jobs can be executed.

1) Multiprocessor taskgeneration with UUniFastDiscard: The taskset generation is based on the UUniFast algorithm from Bini and Buttazzo [9]. It is capable of constructing random tasksets with a given utilization and number of tasks. The algorithm generates the tasksets uniformly over the space that is specified by the number of tasks and the utilization. The UUniFastDiscard algorithm is an extension of the other algorithm. It restricts the randomly calculated tasksets by given conditions to generate more realistic tasksets. We specified the conditions that a task should have at least an utilization of 0.1% and a maximum of 50%. Every time the algorithm generates a taskset that does not meet the conditions it is discarded and the generation starts over again until the conditions are met and the algorithm accepts the taskset.

2) Global and Partitioned Multiprocessor Scheduling: Unlike the uniprocessor scheduling, where only one job can be processed at a time, several can now be executed simultaneously. In the following let p be the number of processors considered by the simulator. To chose the next processed jobs, we can use either global or partitioned scheduling. A global scheduling algorithm uses one taskset for all processors, therefore any task can be executed on any processor. A partitioned scheduling algorithm on the other hand, splits the taskset into

multiple sets - one for each processor. We achieved this split by assigning one of the processors to each task. Now a task can only be executed on its assigned processor [8]. In the next step we need to consider both, global and partitioned scheduling, to implement the following scheduling algorithms.

C. Implementation of scheduling algorithms

Generally, multiprocessor scheduling requires the scheduling algorithms to return a maximum of p tasks to be processed, one per processor. Therefore every algorithm returns a list with exactly p task ids. In case of an empty processor, the algorithm fills the list with a negative value as a placeholder. For each algorithm we implemented a global and a partitioned version, that meets the requirements from section III-B2.
In the first step we extended the given scheduling algorithm. The uniprocessor version iterates over the list of all tasks and returns the first task with workload left. The global multiprocessor version just continues iterating until p tasks with workload are found. In case that there are less than p tasks with workload left, the list will be filled with placeholders. The partitioned multiprocessor will iterate p times over the taskset to write the first task with workload that is assigned to the current processor into the list. Analogue a negative value is used, if the algorithm does not find task with workload.
In the second step we implemented earliest deadline first (EDF). The implementation of this algorithm is almost the same as the previous one. The main different lies in the choice of the next processed task. In addition to the demand of having workload left, we need to chose the task which deadline is the next. According to the multiprocessor environment the algorithm needs to return the tasks with the p next deadlines for global scheduling and the p tasks with the lowest deadline for each processor for partitioned scheduling.
The third algorithm we implemented is fixed-task-priority (FTP). Every task has an assigned priority which is responsible for the execution order of the tasks. FTP returns the tasks with the p highest priority for global scheduling and the highest priority task for each processor for partitioned scheduling. FTP leads to different results depending of the chosen distribution of priorities. We implemented three common methods to assign the priorities. The first assignment is to assign the priorities randomly in a process called random priority (RP). In the second assignment the priorities depend on the tasks deadline. The lower the deadline, the higher the priority. This priority assignment is called deadline monotonic (DM). The third assignment is called rate monotonic (RM) and assigns higher priorities the lower the period of the task.

D. Further features

Besides the functionality of the miss rate simulator, we added some useful features which came in handy for debugging. One feature was the adjustment of the 'tableReport' function that gave an overview of the actual status of the simulator. To design this report more clearly, we used a pandas data framework instead of the two dimensional list that was used before. This has the advantage that columns can be indexed with strings, which give more information about the tables content.
Another feature was the implementation of plotting charts that show the execution sequence of the tasks for each processor. This feature uses the graphing library plotly for python to create interactive gantt charts [7].

IV. Problems

In the following we present the problems and challenges that we faced, while programming the simulator. When trying to run the simulator for the first time we ran into the first problems. The simulator was not able to run, because of two errors. Later, we had to figure out a more complex error that did not cause the simulator to crash, but to give us unrealistic results.

A. Before running the simulator

The first error that we got was:

```
ValueError: Object arrays cannot be
loaded when allow_pickle=False
```

This originates from security reasons [3]. We want to load a pickle, which is a datastructure saved as a file. That means, when you load a pickle, you load a file that could have been corrupted or modified by another person, which could now contain malicious code that would be executed when loading the file with `numpy.load()`. The rule of thumb for this is that you are safe, as long as you only load files, where you trust the source [4]. Since we only use files that we generated, we are able to safely use pickles. Therefore, the value of `allow_pickle` should be set to `True`. This can be achieved by adding the following code around the `numpy.load()` call [3]:

```
import numpy as np
# save np.load
np_load_old = np.load

# modify the default parameters
# of np.load
np.load = lambda *a, **k:
np_load_old(*a, allow_pickle=True, **k)

# here is the original numpy.load() call

# restore np.load for normal usage
np.load = np_load_old
```

This code will set `allow_pickle` to `True`, but only for this call, so in other calls of `numpy.load()` the value of `allow_pickle` is still set to `False`. Another error that occured, before we could start programming, was with the type `float128`. Since the given simulator was programmed on a linux computer, the programmers were able to use it without any problems. However, on some other operating systems, including Windows 64 bit, this type does not seem to be usable. [1] This caused an issue for us at first. The first workaround was changing to an operating system where `float128` was available by using a virtual machine with Ubuntu. That worked out well and we were able to actually start programming. But since this was rather a quick fix than a satisfying solution, later we decided to go for a different workaround by using a different type instead. While researching for a different solution, we discovered that on most operating systems the `float128` type is actually only a 80 bit float with 48 bits of padding to fill 128 bits. [5] This can be checked with the following command: `numpy.float128(1) + numpy.float128(2**-64) - numpy.float128(1)`. Normally, the result should be 2^{-64}, but on most systems, this would return 0.0, implying that the `float128` type contains less than 64 bits of precision. [1] Based on this, we decided to use the platform independant type `float64`. In our opinion, gaining platform independancy overweighs the loss of precision in the float.

B. Simulation time

The original processor wasn't developed with a multiprocessor environment in mind and has a functions, that greatly increased the simulation time. While trying to simulate with the configuration in table I, the time exceeded over 4 hours and was then aborted using the computer with configuration VII-A. Because of this we restricted the processors to 8, but stil had a lot of simulation time. A possible way to simulate 16 processors in a more timely manner, would be to decrease the amount of tasks per processor and jobnumber.

Number of Tasks per Processor	10
Number of Tasksets	100
Number of Processors	16
Scheduling Method	EDF
Processor Type	PARTITIONED (1)
Generation	1 (random)
Utilization	75
Faultrate	10^{-4}
Jobnum	1000

Table I: Configuration

The calculations for simulation rise drastically with the amount of processors, because the amount of tasks is correlated. With double the amount of processors we have double the amount of tasks and double the amount of release events in one simulation VII-B. This more than doubles the amount of calculations needed, because while the simulated timeframe stays roughly the same, the time between release events gets lower. This leads to more interruptions and smaller remaining execution times for jobs. Those also increase the minimum delta's in our program. For each delta simulated we need to calculate the current highest priority job according to the chosen algorithm to work on. Here the increased amount of tasks also creates more jobs to filter through and increases the time aswell. Another factor is the amount of calculations for each delta, as with more processors, more workloads have to be subtracted.

A huge time consume was also the way time progressed. At first the time always stayed at 0 and the events and deadlines moved closer to 0 with "passing time". This resulted in a lot of unnessecary calculations of upto $jobs + events$ additions for each delta to a new event. The amount of events during execution is always equal to $2 * tasks$, as each task has a deadline and release, and the current amount of jobs in a simulation is between 1 and $tasks$. The amount of jobs during the simulation rises with the utilization as more tasks have remaining workload. With a schedulable taskset on 16 processors and 10 tasks per processor this would mean between $(1 + 320) * 4,377,146.224 = 1,405,063,937.9$ and $(160 + 320) * 4,377,146.22 = 2,101,030,185.6$ added calculations. For the average amount of events see Appendix VII-B. The new approach is adding the delta to a continous counter and add the current time to the event deltas and deadlines. This way we only have one calculation for each event.

C. Jobnumber

When simulating the tasksets, the jobnumber is a vital parameter to get an accurate miss rate. If the jobnumber is too low, not enough jobs could be released to force an error. A set is only then fully tested, when the starting pattern (all tasks first released at the same time) is repeated. Till then a 101% utilization set could potentially work, on a low faultrate, till the last job and only then fail. To ensure the full test, the jobnumber should be set to the least common multiple of all periods divided by the maximum of all periods. This leads to jobnumbers higher than 200,000 and is not feasible to run on our computers. To limit the needed the jobnumber we restricted the possible periods during the generation to 1, 2, 5, 10, 50, 100, 250 and 1000. This creates a new problem, as the high difference in periods makes faults not as meaningful between tasks and wouldn't give a realistic represenation. Also the low jobnumber meant a lower average amount of releases, see Appendix VII-B, and those reduced the occurence of faults. This greatly reduces the amount of misses in lower utilization but gives a more accurate representation for 100% utilization.

V. Evaluation

In section V shows a overview for the configuration and operation of the simulator. We also show examples of our simulation and compare the different configurations and their effect.

A. Configuration

This program allows the simulation of a multitude of different scenarios. Some are done in experiments.py II and some in the command line execution III. When using the program the mode parameter selects the type of work to be done. The mode 0 generates tasksets using the other configuration parameters, that later can be simulated. The generated tasksets are saved in the inputs folder. Mode 2 starts a simulation with all combinations of the current configuration and saves the miss rate for each in the outputs folder and mode 3 creates a plot for the configuration. When some configurations in mode 3 haven't been simulated yet, it will simulate and save them. Mode 4 takes the current configuration and calculates the average amount of releases of all tasks for the generated taskset required, to reach the specified jobnumber. Mode 7 was used to run specific configuration not bound by the current way of possible configurations. There are 2 available generationtypes. The first (0) uses a preset of possible periods all being a divider of 1000 and the second (1) uses randomly generated periods between 1 and 100. Half of the randomly generated periods are between 1 and 10, the other between 10 and 100. "Part" allows for saving different tasksets with identical configuration. The implemented scheduling Methods are RP (1), EDF (2), DM (3), RM (4). The "Hard Task Factor" multiplied by the execution time is the time in case of a fault, wich are calculated on release of every task. In our case we always left it on 1.83. The jobnumber determines, when the simulation stops. As soon as the highest period task reaches the deadline equal times to the jobnumber the simulation is stopped. The jobnumber is overridden to 1 in case of generationtype 0.

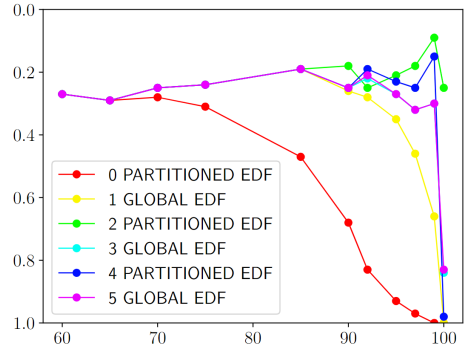

Figure 2: Miss rate of Partitioned/Global EDF with different number faultrates and generationtype 1

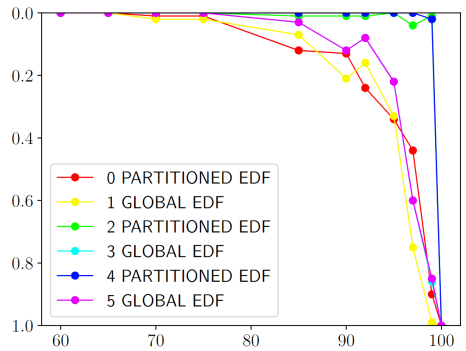

Figure 3: Miss rate of Partitioned/Global EDF with different number faultrates and generationtype 0

B. Example Simulation with Random Periods

To show the ineffectiveness of random periods with low jobnumbers we chose EDF as our scheduling method, as it is the most stable. Also we selected 8 processors, 10 tasks per processor, generationtype 1, 100 tasksets, jobnum of 100 and faultrates of 1e-2 (lines 0 and 1), 1e-4 (l. 2 and 3) and 1e-6 (l. 4 and 5). Figure 2 shows the miss rates of the simulations. The x-axis

Name	Possible Values	Type	Modes
Amount of Processors	$\geq$ = 1	Arr[int]	0, 2, 3, 4
Scheduling Methods	0, 1, 2, 3	Arr[int]	0, 2, 3
Hard Task Factor	$\geq$ = 1	double	0
Utilizations	$\geq$ = 0	Arr[double]	0, 2, 3, 4
ProcessorTypes	0, 1, 2	Arr[int]	0, 2, 3
Jobnumer	$\geq$ = 1	int	2, 3, 4
Faultrates	$\geq$ = 0	Arr[double]	2, 3

Table II: Configuration in experiments.py

shows the utilization of the tasksets and the y-axis shows the miss rate. The miss rate is defined as the amount of tasksets with atleast one miss divided by the overall amount of tasksets. The first problem is the fluctuating miss rate with increasing utilization. What we normally expect is a higher miss rate with higher utilization, but for example in case of line 2 the lowest point is at 97% utilization. Another problem is the miss rate on 100%. With 100% utilization should come 100% miss rate, but in figure 2 only line 0 reaches that point. For comparison the figure 3 used the same configuration except the generationtype 0. The previous outlined problems are not evident in this figure. The inconsistent results of the generationtype 1 show the problems with the low jobnumber and we stopped using the random periods.

C. Partitioned vs. Global

To compare partitioned and global scheduling we chose the same configuration as in the previous chapter and only changed the generationtype to 0. The figure 3 shows the results of the simulation. For the lower faultrates (lines 2-5) we can see, that partitioned performs better than global scheduling. On the higher faultrate 1e-2 (l. 0 and 1) the performance seems to be similar. The higher miss rate for global scheduling could be the result of choosing the lower periods tasks from "other processors" instead of the higher and longer periods from their "own processors". The figures 4 shows a scenario of 4 tasks on 2 processors. In case of partitioned scheduling all jobs reach their deadline and global scheduling misses the deadline of the task 4 by 0.133 seconds. This is because global EDF scheduling works at 1.5 on processor 2 on task 2 instead of task 4. In the end both processors can't work on the same task at the same time and processor 1 runs idle, while task 4 misses it's deadline.

D. Results

In this section we are going to have a look at some example results, provided by the multiprocessor miss rate simulator. Figure 5 shows nine line-charts. Each of the

τ_1 :$T_1 = 1$, $D_1 = T_1 = 1$, $E_1 = 0.5$, $P_1 = 1$
τ_2 :$T_2 = 1.5$, $D_2 = T_2 = 1.5$, $E_2 = 0.633$, $P_2 = 1$
τ_3 :$T_3 = 1$, $D_3 = T_3 = 1$, $E_2 = 0.5$, $P_3 = 2$
τ_4 :$T_4 = 3$, $D_4 = T_4 = 3$, $E_2 = 1.5$, $P_4 = 2$

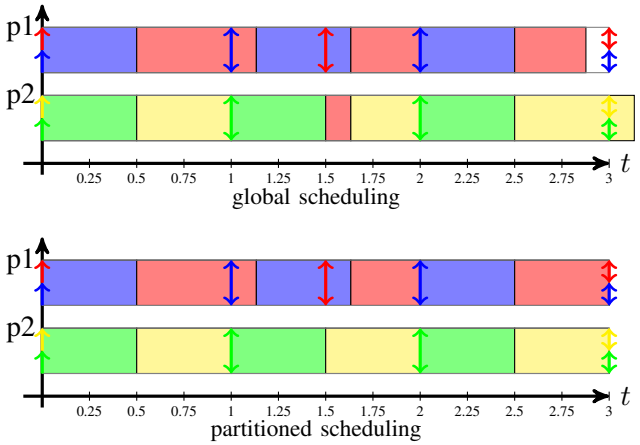

Figure 4: Example for partitioned vs global EDF scheduling

Name	Possible Values	Type
Mode	0, 2, 3, 4, 7	int
Amount of Tasks per Processor	≥ 1	int
Amount of Tasksets	≥ 1	int
Generation Type	0, 1	int
Part	any number	int

Table III: Configurations in run

nine charts shows the miss rate of the task sets on the y-axis and the utilization on the x-axis. The same task sets have been scheduled by partitioned EDF (the red lines), partitioned DM (the green lines), global EDF (the light blue lines), global DM (the purple lines). The task sets have been simulated with three different fault rates and three different number of processors. The fault rate is decreasing from the left to the right, while the number of processors is increasing with each row from top to bottom. The charts in the first column were simulated with a fault rate of 10^{-2}, the second column with a fault rate of 10^{-4} and the last column with 10^{-6}. Two processors were used in the first row of charts, four processors in the second and eight processors in the last row. The task sets were generated with task generation type 0, because of the more reliable and accurate results, which was mentioned in section V-A. Since we used implicit deadlines for the tasks, DM and RM are practically the same scheduling algorithms, so that all observations about DM can also be made for RM. First we will have a look on the different rows to compare

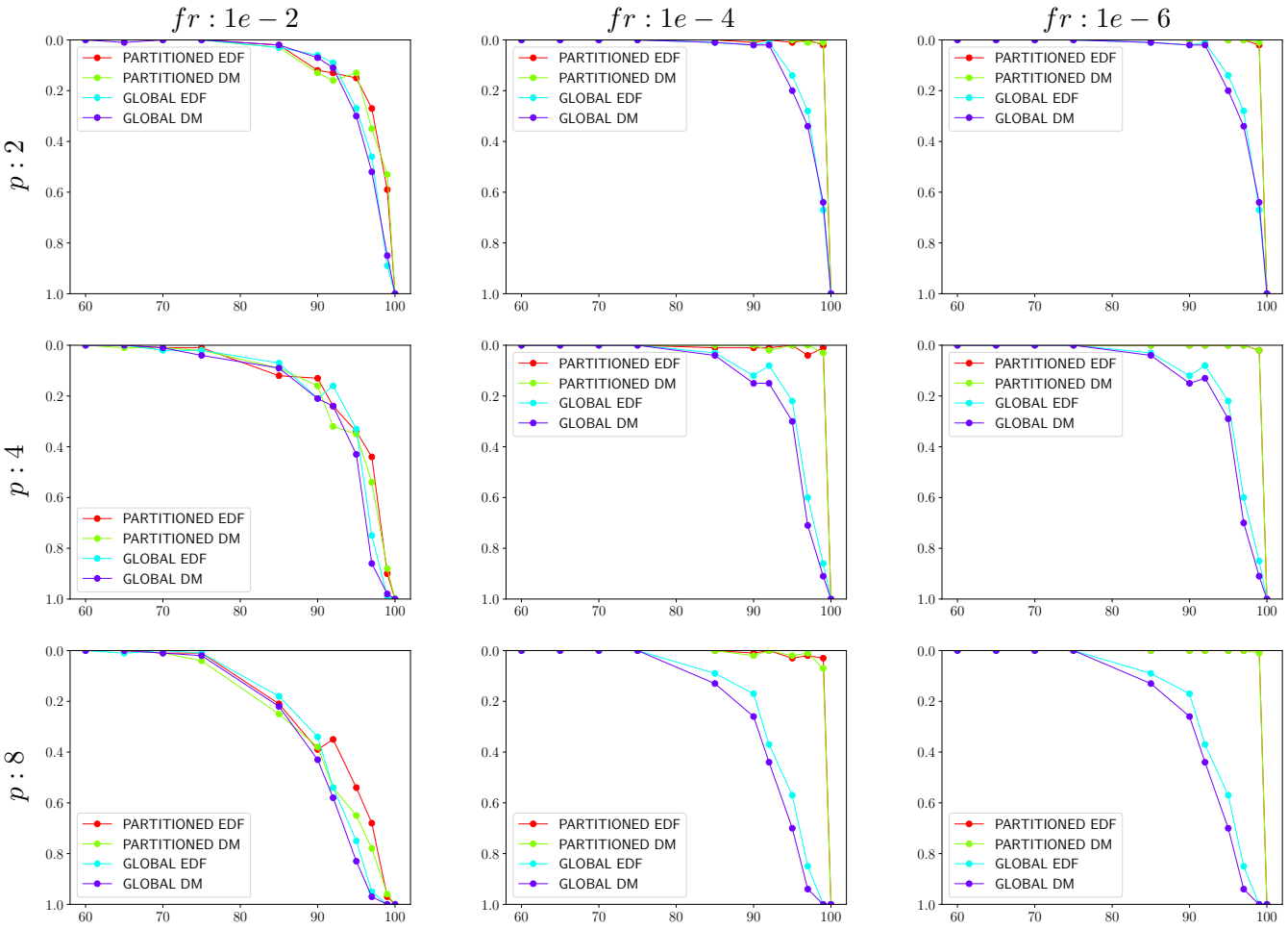

Figure 5: Miss rate of Partitioned/Global EDF/DM Scheduling with different number of processors and fault rates

the results according to a higher number of processors. In each column we can see a more flattened curve the more processors were use. This effect can especially be observed for the global scheduling algorithms. Although the miss rate increases with the amount of processors, we have to take into account that the number of tasks and the workload is increasing aswell. In the first column, partitioned scheduling is affected aswell, so that the different scheduling algorithms do not seem to vary a lot among each other. This is different when looking at the second and third column, where partitioned scheduling is much better than global scheduling. With a lower fault rate, the workload overall is decreasing and the miss rate is going down, as we can observe between the columns for each algorithm. If we are looking at the differences between EDF and DM scheduling, we can not see major differences. Only for global scheduling it seems like EDF has a slightly lower miss rate than DM.

VI. Conclusion

The conversion of the python version was successfully done with the script ”2to3”.

With more complex modifications we were able to build a multiprocessor environment with partitioned and global scheduling modes. We also extended the existing algorithms to fit the partitioned and global scheduling environments and added the scheduling methods EDF, DM and RP. The configuration possibilities were extended aswell to accomodate the new features.

While working on the simulator we also improved and/or changed features, like the passing of time and taskgeneration, to increase performance and accuracy IV.

With our simulations in section V we were able to show, that in some cases partitioned scheduling performed better than the global scheduling, and were able to compare EDF with DM.

The simulator can be further developed in the future. There are still features that can be adjusted to simulate more realistic task sets. The performance can be increased by optimizing the implemented scheduling methods. New modes to adjust the simulator can also be added, for example to disable the stopOnMiss feature.

VII. Appendix

A. Computer Configuration

Processor: Intel i7-4790k, RAM: 24gb

B. Average amount of task releases

Processor Amount	Jobnum	Av. Amount of Events
2	100	51,274.85
4	100	105,946.29
8	100	217,221.1
16	100	437,715.14
2	1000	512,747.66
4	1000	1,059,462.73
8	1000	2,172,212.34
16	1000	4,377,146.22

Table IV: With randomly generated Periods

Processor Amount	Jobnum	Av. Amount of Events
2	1	8,473.44
4	1	18,004.22
8	1	36,839.5
16	1	73,947.58

Table V: With set Periods

References

[1] Cannot use 128bit float in Python on 64bit architecture. https://stackoverflow.com/a/29821557. Accessed: 2020-09-08.

[2] Event-based Miss Rate Simulator and Deadline Miss Rate. https://github.com/kuanhsunchen/MissRateSimulator. Accessed: 2020-09-15.

[3] How to fix 'Object arrays cannot be loaded when allow_pickle=False' for imdb.load_data() function?'. https://stackoverflow.com/questions/55890813/how-to-fix-object-arrays-cannot-be-loaded-when-allow-pickle-false-for-imdb-loa/56062555. Accessed: 2020-09-08.

[4] NumPy: consequences of using 'np.save()' with 'allow_pickle=False'. https://stackoverflow.com/a/41696642. Accessed: 2020-09-08.

[5] What is the internal precision of numpy.float128? https://stackoverflow.com/a/17023995. Accessed: 2020-09-08.

[6] Automated python 2 to 3 code translation. https://docs.python.org/3/library/2to3.html, August 2020. Accessed: 2020-08-31.

[7] Gantt charts in python. https://plotly.com/python/gantt/, September 2020. Accessed: 2020-09-05.

[8] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor scheduling for real-time systems*. Springer, 2015.

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

[10] Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. Analysis of deadline miss rates for uniprocessor fixed-priority scheduling. In *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018, Hakodate, Japan, August 28-31, 2018*, pages 168–178. IEEE Computer Society, 2018.

[11] Robert Davis, Attila Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems.