Proportionate Fair based Multicore Scheduling for Fault Tolerant Multicore Real-Time Systems

by Tight Coupling of Error Detection and Scheduling

Stefan Krämer[1,2], Jürgen Mottok[1], Stanislav Racek[2]

[1]OTH Regensburg
Faculty of Electronics and Information Technology
Seybothstr. 2, D-93053 Regensburg, Germany
{stefan.kraemer, juergen.mottok}@oth-regensburg.de

[2]University of West Bohemia
Faculty of Applied Sciences
Univerzitní 22, 306 14 Plzen, Czech Republic
stracek@kiv.zcu.cz

***Abstract* – In this paper *we* present a scheduling approach for safety critical, fault tolerant, multicore real-time embedded systems. For this kind of systems, not only the correctness of a computed result but also the strict adherence to timing requirements of computation is essential to avoid any kind of damage. To react to unpredictable, arbitrary hardware faults suitable error detection mechanisms have to be applied. The caused error itself and the detection and correction have great impact on the system's timing behavior. To still keep the real-time requirements, the used scheduling algorithm has to ensure maximum flexibility to disturbances of the timing. The group of Proportionate Fair (*Pfair*ness) multicore scheduling algorithms has been proven to create an optimal schedule in polynomial time. The contribution of this paper is a *Pfair*-based algorithm that uses tight coupling between the error detection mechanisms and the scheduler of the real-time operating system to establish a loop-back connection.**

Fault tolerant systems; safe software processing; real-time operating systems; multicore scheduling; discrete event simulation; fault injection; Pfair scheduling; Stochastic simulation;

I. INTRODUCTION

The usage of multicore processors in embedded real-time systems is gathering pace. Even if – for example in the automotive domain – the migration towards multicore systems is done quite conservatively. Different legacy systems are combined on one new multicore ECU to reduce the number of ECUs and to reduce costs. Because of this trend, mixed criticality embedded systems will become more common due to the fact that multiple functional components are combined into one ECU [15]. Nowadays static partitioning of tasks to cores and manly priority based scheduling approaches are used. Different timing domains and different levels of criticality for each application part are quite common. This means that in these mixed criticality systems QM-tasks (non-safety critical application part) and safety-critical real-time tasks – where not only a wrong computed result, but also a missed deadline for a calculated value can cause damage – have to be scheduled. The safety-critical tasks need additional steps to ensure the reliability requirements. QM-tasks do not require additional actions of protection. Multicore systems offer not only an advantage regarding performance, but also have a high potential of increasing the reliability of software intensive embedded systems [12]. That is why a software approach can be more flexible and handles various safety requirements differently. It executes non-critical tasks in a single instance and safety-relevant tasks in e.g. redundant execution on a multicore system. Thus the overall performance and reliability can be increased by using software reliability mechanisms. The requirement to reduce costs of those systems leads to a trend of shifting safety mechanisms like diverse hardware towards software approaches [10].
A task set scheduled on a multicore with a global scheduling algorithm can in general react more flexible in case of influences by transient faults than singlecore scheduling algorithms [6], because a single scheduling instance is responsible for all computation resources (cores). In case of a detected transient error – e.g. by usage of coded processing – the scheduling unit can distribute the load over all execution resources more easily, for the benefit of the safety critical tasks. The impact of timing disturbances of the overall system can be reduced, if the scheduling algorithm has a tight connection to the error detection and correction subsystem. With this loop-back – without assuming always worst case execution times but using specific error reaction execution times – more efficient schedules can be created and the disturbance of the timing of all tasks (including QM-tasks) can be reduced to a minimum.
Therefore we present the discontinuous fluid schedule extension for fluid based schedules like *Pfair* that uses this tight connection to the error handling subsystem to optimize the system timing behavior in case of the occurrence of arbitrary hardware faults.

This paper is structured as follows. The first section gives a short overview of the general considered system architecture, including the impact of this architecture on the reliability of the system.
In the second section, the Loop-Back (LB) - *Pfair*

scheduling algorithm is introduced. The evaluation of the *LB-Pfair* is done by discrete event simulation that offers the possibility of a holistic approach for reliability analysis combined with schedulability analysis of complex safety-critical multicore real-time systems. A case-study with a representative automotive task set is given in section V. Finally Section VI gives a conclusion and outlook for further research.

II. Considered System Architecture

This section will describe the two safety mechanisms considered in this paper, coded processing and symmetric redundant execution [2], [9] and their combined application. The principal modeling of a safety task is shown. In case an error was detected the corresponding task instances are re-executed. The operating system architecture with loop-back connection from the error detecting subsystem to the operating system's scheduler is depicted.

A. Considered fault handling strategies

1) Coded processing

Coded processing is the protection of calculations and their results during operations in an arithmetic unit by means of error detection codes. Important of error detecting codes are the so-called arithmetic codes (AN-codes) that are based on ordinary algebra like addition and multiplication [1]. Forin made the first use of coded processing in a real application [13], [11], [17].

2) Symetric redundant execution

The duplication of components is a common method to increase the reliability of systems. The same task is duplicated into n task instances, which are executed one after another on a singlecore system (time redundancy) or on different cores on a multicore system (space redundancy). The computation results of the n instances are compared and evaluated after complete execution by the voter and possible single faults can be detected [12], [16].

3) Combined approach

According to the needed Safety Integrity Level (SIL) both approaches can be combined. By coding an error can be detected in each individual instance. For resolving the error, re-execution is only necessary if both redundant and coded instances have detected an error at the same time.

B. Safety-Task Design

1) Coded Task Processing

The scheduler – and within that the *LB-Pfair* algorithm – is directly called by a system call, if the underlying coded processing library detects an error. The check for error occurrence at coding processing is directly integrated in the task execution. The time stamp and the additional execution time are reported to the *LB-Pfair* algorithm.

2) Symmetric redundant task execution

For employing symmetric redundant execution in a multicore system, some effort has to be made to achieve the necessary synchronization between the two task instances using standard scheduling algorithms. The synchronisation in case of *LB-Pfair* scheduling is included in the actual scheduling algorithm. Paired safety-task instances are controlled by a task allocation constraint. The first task waits for the second instance to reach the synchronisation point. The comparison of the computation results is done in a global voter.

This could be integrated in a safety supervisor [7]. After comparison the LB-Fair scheduler gets the information if additional execution time for re-executing the instances is necessary and can consider this for computation of the next scheduling decision. In the following a combined approach with coded processing and redundant execution is assumed.

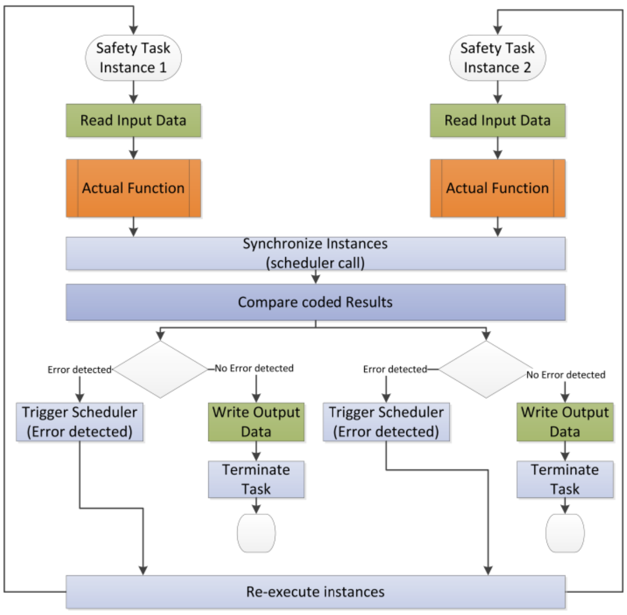

Figure 1: Modeling of symmetric redundant task execution of two coded safety-critical tasks with loop-back connection (trigger Scheduler) to the *LB-Pfair* scheduling algorithm.

C. Operating System architecture

For considering overhead caused by error detection and correction, a tight coupling between the error handling subsystem and the *LB-Pfair* scheduling algorithm executing within the operating system's scheduler has to be established.

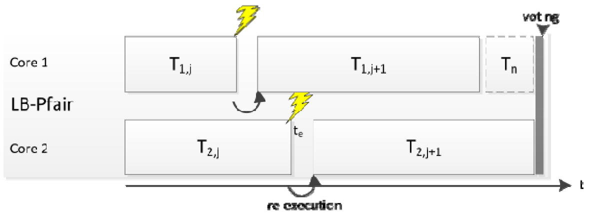

Figure 2: Schematic depiction of the execution of two coded instances of a safety task, scheduled by the *LB-Pfair* algorithm. For impact on the scheduling see Figure 4 at time t_e.

Figure 3 shows the principle concept. The error detection – e.g. by coded processing – is encapsulated in a library function, such that the user can incorporate this functionality transparently [7]. The so called safety supervisor is the connection between the application, the error handling subsystem and the scheduler. The safety supervisor provides information of time and position of error detection. With that information the *LB-Pfair* algorithm can predict – at the moment of error detection (t_e) – how much additional execution time (c_i) is needed for this error correction and can change the schedule dynamically.

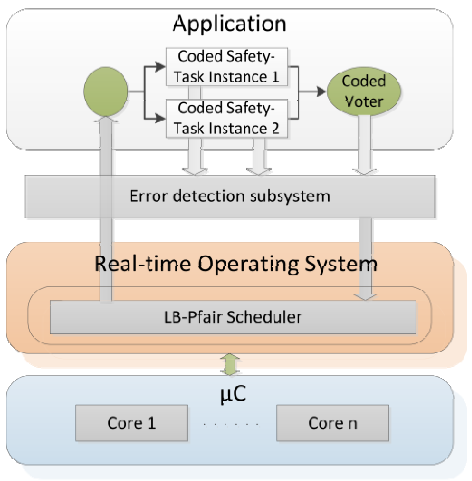
Application
Coded Safety-Task Instance 1
Coded Safety-Task Instance 2
Coded Voter
Error detection subsystem
Real-time Operating System
LB-Pfair Scheduler
µC
Core 1
Core n

Figure 3: Basic architecture of an operating system using *LB-Pfair* scheduling. The error detection subsystem is connected via a feedback-loop to the *LB-Pfair* scheduler.

III. *LB-PFAIR* SCHEDULING

The basic principles of the *Pfair* scheduling algorithm are briefly summarized. In the second part our contribution, the discontinuous fluid schedule as main property of the *LB-Pfair* algorithm is introduced.

A. Pfair scheduling

Pfair [14] based multicore real-time scheduling for periodic tasks is different in one aspect compared to other real-time scheduling approaches. The task instances are executed at a steady rate. Meaning they execute at constant speed in a way that they finish right within their deadline derived from their period. This execution along this steady rate is called fluid schedule. As the execution rate of a core cannot be varied, the execution of a fluid schedule is approximated by dividing the task execution time in so called quants. That are either executed or not. The division into quants – by bypassing the bin-packing-problem – is one of the reasons, why a *Pfair* schedule can be calculated online in polynomial time for periodic task sets [14].

A periodic task instance T_i is defined by its execution time e_i and period (p_i). The correspondent subtask (j) is denoted as $T_{i,j}$. To fulfill the requirement for approximate an uniform execution rate each subtask (quant) has to be executed in a certain window. Different subtasks may run on different cores but they may not execute in parallel on different cores.

There are several extensions and modifications of the original *Pfair* algorithm, such as PF, PD, PD² or ER (early release) [19]. The latter is suitable for fault tolerant scheduling because it does not misspend execution time by executing tasks, if a free execution resource *R* is available.

The main scheduling criteria, besides some tie breaking rules, is referred as the task weight wt_i that is the criterion for calculating the pseudo deadline $d(T_{i,j})$ for each individual. It is defined by the execution time e_i and the activation period p_i of the task *i as follows*:

$$wt_i = \frac{e_i}{p_i} \qquad (1)$$

Informally the weight is the rate at which the task should be executed. The pseudo deadline follows:

$$d(T_{i,j}) = \left\lceil j \frac{1}{wt_i} \right\rceil . \qquad (2)$$

Pfair is proven to create feasible schedules for *k* tasks on *M* execution resources [14], if:

$$\sum_{i=0}^{i<k} \frac{e_i}{P_i} \leq M . \qquad (3)$$

B. Extension of Pfair Scheduling – LB-Pfair

The herein introduced extension – the discontinuous fluid schedule (see Figure 4) – is applicable for many fluid schedule based algorithms.

Besides the actual extension of the scheduling algorithm *LB-Pfair* includes execution allocation constrains and subtask-synchronization. Same subtask instances of coupled safety critical tasks – that are executed in space redundant manner – are not allowed to execute on the same execution resource R. For the couple T_i, T_{i+1} follows:

$$R(T_{i,j}) \neq R(T_{i+1,j}) \qquad (4)$$

To avoid delays for comparing voting results the coupled task instances have to be synchronized. This criterion is realized indirectly by considering the same execution rate and initial task weight. Therefore on an equally balanced system this criterion is fulfilled.

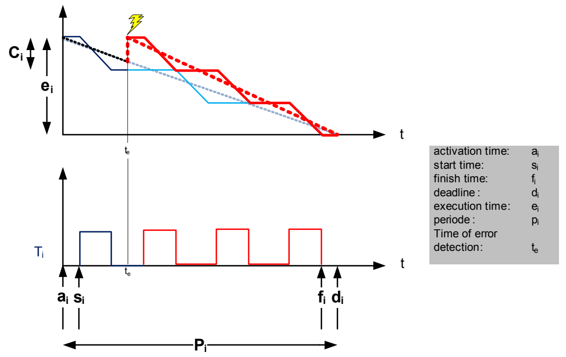

Figure 4: Discontinuous fluid schedule, representing erroneous task execution, with error detection at time t_e.

With the proposed discontinuous fluid schedule, the task weight wt_i is no longer constant over system lifetime (In original *Pfair* algorithm the weight is considered constant over time. But nevertheless the real execution time on a target may fluctuate.). The weight is now additionally dependent on occurrence and detection of errors by the error handling subsystem and the loop-back connection to the algorithm. For correcting the error the task instance is re-executed. This additional execution time c_i results in a temporally increased task weight wt_i*.

This in turn leads to a temporarily higher dynamic priority of the faulty task instance. The ER-extension (early release) for *Pfair*-algorithms is applied for *LB-Pfair* scheduling. Because of its non-work-conserving properties, it is suitable for safety-critical task to ensure free resource capacity at the end of a period, in case of error occurrence. If the safety supervisor layer detects an error the scheduler is informed about the time of error occurrence t_e by the loop-back connection. With the start time s_i of the task, the additional error correcting time c_i is defined by:

$$c_i = t_e - s_i \tag{5}$$

The dynamic task weight wt_i* follows:

$$wt_i^* = \frac{e_i + c_i}{p_i}. \tag{6}$$

wt_i^* depends on the error rate λ that affecting the system, as the rate and time of faults influence the time of error detection t_e.

The schedule criterion – the local deadline – can now be defined by:

$$d^*(T_{i,j}) = \left\lfloor j \frac{1}{wt_i{}^*} \right\rfloor \tag{7}$$

According to equations (3) and (6) a feasible schedule can be calculated, if the following equation is satisfied for every point in time.

$$\sum_{i=0}^{i<k} wt_i^* \leq M\Big|_{\forall t}. \tag{8}$$

Because of the complexity of task systems and time of error occurrence as well as the considered execution time, wt_i^* can hardly be determined in an analytical way for an arbitrary system. Therefore evaluation is done by a discrete event simulation approach (See section IV).

IV. EVALUATION BY SIMULATION

For the simulation based evaluation of the *LB-Pfair* scheduling algorithm, a multi seed discrete event simulation is used. The discrete event simulation is used to model the variance of task execution durations and the general feasibility of a schedule in a multicore real-time embedded system. The reliability indices are determined by probabilistic fault injection of transient faults. To cover a wide range of the design space, each simulation run is executed multiple times. In order to establish a reasonable confidence in the result and to gain a high precision, the simulated time span of one single run was set to 5000 seconds that were executed 1000 times each.

The modeling of the task execution is very close to real task execution on a target (see Figure 1 and Figure 3). In this example, at the beginning input data is read, then the safeguarded processing ("Actual Function") is executed and the result is checked. If no error occurred, the task terminates normally and writes to the output. Otherwise, if an error was detected, the scheduler is called and informed about the additional execution time in case of *LB-Pfair* scheduling.

A. Simulation Model

The real system behavior can be depicted quite accurately starting at the level of instructions up to the modeling of the application with tasks and operating systems by usage of Monte-Carlo based discrete event simulation framework. The used model consists of a hardware part, defining the processor, which itself consists of multiple cores. These cores execute the instructions defined in the application part of the model. The execution time of these instructions is defined by the clock of the core and the amount of instructions the core can execute per tick.

A next part of the model describes the operating system architecture, including the *LB-Pfair* algorithm.

In the application part of the model the different task and interrupt service routines are defined. A task itself contains several function calls and has the capability to model conditional branching of the execution flow. Functions are assembled by instruction blocks, they define the amount of instructions (constant number, or specified by a distribution) that have to be executed by the core.

Furthermore the stimulation of the simulated system has to be described. These stimulation mechanisms can either be configured periodically or can be varied by a distribution [18]. The stimulation system is e.g. responsible the activation of tasks and interrupt service routines (ISRs). It also used for implementing the fault injection.

B. Simulation input parameters

During the simulated task executions, faults are injected via system stimulation. The fault injection rate is varied over simulation time in a certain range (see section V). The runtimes of the tasks are defined by uniform distribution to model the actual variation of task execution on a real target.

C. Simualtion output parameters

By tracing the different events of the simulation, a great variety of timing and reliability metrics can be evaluated. The simulation is used to simulate not only the standard reliability indices but also to simulate the real-time characteristics of an embedded system. In this paper we focus on the response time as an evaluation criteria.

V. CASE STUDY AUTOMOTIVE TASK SET

A. Task set for evalutation

The *LB-Pfair* algorithm is evaluated using a simplified automotive-like task set. The task set consists of one safety-critical application, executed with coded processing and redundant execution. Four further non-safety-critical applications are executing on the ECU and are causing additional load. The tasks are located in different timing domains. This represents the different application types, running on the target. A symmetric multicore processor is used as target device.

OSEK (OSEK: "Open Systems and their Interfaces for the Electronics in Motor Vehicles") is a standard for automotive embedded real-time system, that is based on partitioned priority based scheduling. OSEK-algorithm, *Pfair* and *LB-Pfair* are compared in this study. The quant-size of the latter is set to 0.1ms. Safety-critical tasks cannot be preempted by QM-tasks in the OSEK-scheduler. The safety-critical task has the highest priority in the task-set, despite a short running task (QM-task 4). For OSEK-scheduling the tasks are statically allocated to cores. The priorities are given in Table 1. For *Pfair* algorithms the priorities are dynamically calculated by the weight and in case of *LB-Pfair* additionally depending on the presence of a detected error A dualcore *M=2* system is considered.

Table 1: Configuration of the simplified task set

	Execution time [ms]	Period/ Deadline [ms]	OSEK Core	OSEK Priority
QM-task 1	10 – 20	100	1	30
QM-task 2	10 – 20	100	2	30
QM-task 3	90 - 110	500	2	20
QM-task 4	0.1 – 0.2	1	1	50
Safty-task Instance 1	75	1000	1	40
Safty-task Instance 2	75	1000	2	40

1) Error detection

According to Figure 1 error detection for the two safety-critical tasks is considered by dual modular redundancy and coded processing. That means that data is processed in the coded domain by two different instances (different coding) in space redundancy on the dualcore. In case an error is detected by the comparator the two task instances are activated again and therefore re-executed.

2) Fault injection

The simulation is configured to simulate bursts of faults and consequently covers heavily faulty situations, e.g. caused by fluctuations of the power supply. Thus multiple faults are quite likely. The injected fault rate (λ) is varied in a range from 1ms to 200ms, which means that there will be at least one injected fault during the execution period of the safety-critical task. Faults are injected on each core individually. Thus faults would affect safety and QM-tasks. However, error detection and correction is only implemented for safety-tasks.

B. Evaluation of simulation results

1) Response time metrics

Table 2 and Table 3 summarize the results of the discrete event simulation runs. As mentioned, the individual simulations were executed 1000 times each with a simulation time of 5000 seconds and different seeds for the Monte Carlo-based simulation.

In general, it can be seen that there is a difference between the response times for the QM-tasks with *LB-Pfair* scheduling and OSEK scheduling. The average response times (see Table 3) are better for QM tasks in *Pfair* and *LB-Pfair* scheduling, because of dynamic task allocation during runtime, which allows a more evenly utilization of the cores. For the important safety-critical tasks the average response time for *LB-Pfair* is better compared to OSEK and *Pfair*. The bad result for standard *Pfair* is an effect of the not considered additional execution time in case of error handling and the originally relative low weight (because of the long period) of the safety critical task. Hence, a low dynamic priority for this task is the consequence. It can be seen (see Table 2) that for safety-critical tasks the maximum response times are almost half as long as for *LB-Pfair* scheduling compared to OSEK scheduling. *LB-Pfair* scheduling recognizes the higher execution time requirements of safety-critical tasks in case of an error and therefore increases their local scheduling priority. A side effect is the higher maximum response time at *LB-Pfair* scheduling for QM-tasks, compared to the other two algorithms.

In average the response times of the safety-critical task are lower, compared to the standard algorithms. The maximum response time of the relevant safety-critical tasks ($t_{response,\ max} = 3421\text{ms}$) can be lowered significantly compared to OSEK ($t_{response,\ max} = 6024\text{ms}$) and original *Pfair* scheduling.

The response times of the safety-critical tasks at *LB-Pfair* scheduling have a standard deviation of 480ms over all multi-seed simulation runs. At OSEK scheduling the standard deviation is 949ms. This shows that *LB-Pfair* scheduling produces results with less outlier.

Table 2: Maximum response time for OSEK, *Pfair* and *LB-Pfair* scheduling for task set of Table 1.

	OSEK	*Pfair*	*LB-Pfair*
QM 1[ms]	6042.94	48.47	3135.63
QM 2[ms]	184.78	47.99	3133.83
QM 3[ms]	2381.29	176.55	3040.67
QM 4[ms]	0.19	0.69	303.13
Safety-task instance 1 [ms]	6023.76	53091.50	3420.50
Safety-task instance 2 [ms]	6023.60	53088.55	3421.00

Table 3: Average response time for OSEK, *Pfair* and *LB-Pfair* scheduling for task set of Table 1.

	OSEK	*Pfair*	*LB-Pfair*
QM 1[ms]	142.53	21.087	71.39
QM 2[ms]	71.66	21.11	71.51
QM 3[ms]	551.73	136.58	255.00
QM 4[ms]	0.14	0.14	0.75
Safety-task instance 1 [ms]	803.82	7069.14	667.70
Safety-task instance 2 [ms]	803.65	7069.85	667.95

2) Deadlines and response time distribution

Table 4 shows the percentage of task instances of the safety-critical tasks that violate their deadline requirement. Task instances, which could not be started as the previous instance is still executing – due to longer execution time resulting from error correction – are counted as deadline violation.

Table 4: Quota of deadline violations for OSEK, *Pfair* and *LB-Pfair* scheduling for task set of Table 1.

	OSEK	*Pfair*	*LB-Pfair*
deadline violations safety [%]	53.55	99.05	29.54

Figure 5 and Figure 6 depict the distribution of response times for OSEK and *LB-Pfair* scheduling. The step in the aggregated frequency graph represents task instances that are not executed. Low response times do not occur at *LB-Pfair* scheduling. This results from the fair scheduling policy with early release. Thus QM-tasks are executed in a higher quota compared to the OSEK scheduling, where the safety-critical task can be executed right at the beginning of the period because of the higher task priority. But the deviation of response times and the number of longer response times for OSEK scheduling are higher compared to LB-Fair scheduling (see also Table 2 and Table 3). The extensions by *LB-Pfair* algorithm leads to a better performance regarding response times and deadline violations.

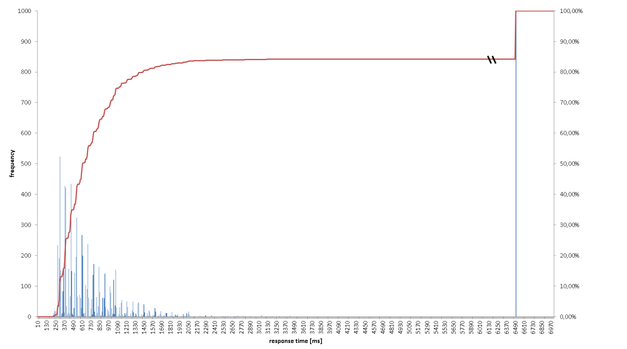

Figure 5: Response time histogram (frequency, blue and aggregated values, red) for the safety-critical tasks in *LB-Pfair* scheduling. Not activated task instances – because of overload condition – are depicted by the step of the aggregated values at the right hand side.

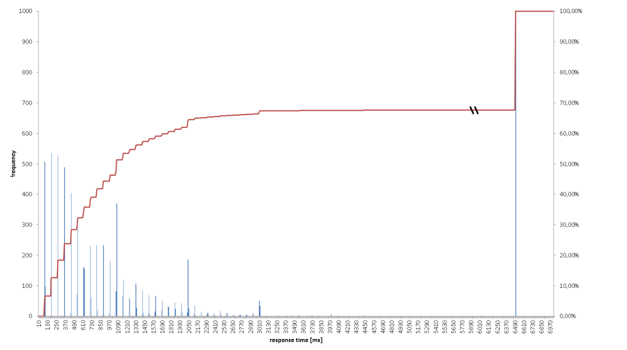

Figure 6: Response time histogram (frequency, blue and aggregated values, red) for the safety-critical tasks in OSEK scheduling. Not activated task instances –- because of overload condition – are depicted by the step of the aggregated values at the right hand side.

VI. CONCLUSION AND FURTHER STEPS

It has been shown that with the extension of the *Pfair* scheduling by the discontinuous fluid schedule, and the execution time loop-back from the error handling subsystem, the important response time metric of the safety-critical tasks can be lowered, compared to other scheduling approaches. The *LB-Pfair* algorithm achieves this by dynamic priority reassignment for the benefit of safety-critical tasks. At the same it maintains the advantage of global dynamic algorithms for scheduling complex task sets dynamically on a multicore. This has been shown with a case study of a simplified task set applicable for the automotive domain.

For the *LB-Pfair* scheduling the early release scheduling policy was chosen. By applying this policy to all tasks the result is a low percentage of short response times compared to the OSEK algorithm. A further extension here is to apply the ER-policy only selectively on critical tasks, which will be topic of future research.

Other points of research are the evaluation of more complex systems and the evaluation of the influence of different applied fault rates to determine a maximum fault rate the system can tolerate. Furthermore the determination of the optimal quant-size and its relation to the synchronization granularity will be part of future research. The basic idea of the discontinuous fluid schedule will be ported to other fluid schedule based algorithms.

REFERENCES

[1] P. Raab, S. Kraemer, and J. Mottok. Cyclic codes and error detection during data processing in embedded software systems. In Proceedings of the 4rd Embedded Software Engineering Congress, pages 577–590, December 2011.

[2] B. Douglass. Doing hard time. Developing real-time system with UML, objects, frameworks, and patterns. Addison-Wesley, 2007.

[3] Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A concept for a safe realization of a state machine in embedded automotive applications. In *Proceedings of the 26th Safecomp Conference*, ISBN 978-3-540-75100-7, pages 283-288, 2007.

[4] L. Yang, Z. Cui and X. Li. A case study for fault tolerance oriented programming in multi-core architecture. IEEE computer society, pages 630 – 635, 2009.

[5] E. Beckschulze, F. Salewski, T. Siegbert and S. Kowalewski. Fault handling approaches on dual-core microcontrollers in safety-critical automotive applications. CCIS 17, pages 82 – 92, 2008.

[6] M. Deubzer, J. Mottok, and A. Baerwald. Dependability-Betrachtung von Multicore-Scheduling. *HANSER Automotive*, November 2010.

[7] P. Raab, S. Kraemer, J. Mottok, H. Meier, and S. Racek. Safe software processing by concurrent execution in a real-time operating system. In *Proceedings of 16th International Conference on Applied Electronics*, pages 315 - 319, September 2011.

[8] G. Buttazzo. Hard real-time computing systems – Predictable Scheduling Algorithms and Applications. Springer, 2004

[9] N. Leveson. Safeware: System Safety and Computers. Addison-Wesley, 1995.

[10] A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-time Task Sets. In Proceedings of EURWRTS, 1996

[11] P. Raab, S. Racek, S. Kraemer, and J. Mottok. Reliability of Task Execution during Safe Software Processing. In *Proceedings of the 15th Euromicro Conference on Digital System Design*, pages 84-89, September 2012

[12] S. Kraemer, P. Raab, J. Mottok, and S. Racek. Reliability analysis of real-time scheduling by means of stochastic simulation. In Proceedings of 17th International Conference on Applied Electronics, pages 151-156, September 2012

[13] P. Forin, "Vital Coded Microprocessor principles and application for various transit systems," in IFAC Symposia Series, 1989, pages 79-84.

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. a. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, Jun. 1996.

[15] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "RTOS Support for Multicore Mixed-Criticality Systems," *2012 IEEE 18th Real Time Embed. Technol. Appl. Symp.*, pp. 197–208, Apr. 2012.

[16] M. Hoffmann and P. Ulbrich, "A Practitioner's Guide to Software-Based Soft-Error Mitigation Using AN-Codes," *(HASE), 2014 IEEE*, no. 0704, 2014.

[17] M. Hoffmann, C. Borchert, and C. Dietrich, "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs," *cs.fau.de.*

[18] S. Krämer, P. Raab, J. Mottok, and S. Racek, "Comparison of Enhanced Markov Models and Discrete Event Simulation," in *2014 Euromicro Conference on Digital System Design (DSD)*, 2014.

[19] R. I. Davis and A. Burns, "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems," *Univ. York, Dep. Comput. Sci.*, 2009.