Suresh Gyan Vihar University, Jaipur
International Journal of Converging Technologies and Management (IJCTM)
Volume 1, Issue 1, 2015
ISSN : 2455-7528

Efficient fault tolerant Scheduling techniques And Backup Overloading techniques For Real Time System

Vipinder Bagga[1], Akhilesh Pandey[2]

Suresh Gyan Vihar University, Jaipur

Abstract: To provide the performance analysis of off-line scheduling algorithms which address the issues of fault tolerance, reliability, real-time, task precedence constraints and heterogeneity in real time systems. The performance measures of these algorithms on which they are differentiated on each other are performance, reliability, schedulability. To compare the performance of backup overloading techniques based on fault tolerance dynamic scheduling algorithms in real time systems. These techniques are measured on the basis of fault rate, task load, task laxity and time to second failure.

Keyword: *Fault tolerant, Real Time System, Scheduling.*

INTRODUCTION

Real-time computer systems are required to produce correct results not only in their values but also in the times at which the results are produced better known as timeliness. In this systems, the execution times of the real-time programs are usually constrained by predefined time-bounds that together satisfy the timeliness requirements. The trademark of real-time systems is that their tasks have deadlines and missing too many such deadlines in a row can result in catastrophic failure. As a result, much effort has been devoted in recent years to develop techniques by which to deliver such systems highly reliable. These efforts have generally involved the use of massive hardware redundancy, i.e., of using many more processors than are absolutely necessary to ensure that enough will remain alive, despite failures, to continue providing acceptable levels of service. A failure in a real-time system can result in severe consequences. Such critical systems should be designed a priori to satisfy their timeliness and reliability requirements. To guarantee the timeliness, one can estimate the worst-case execution times for the real-time programs and check whether they are schedulable, assuming correct execution times. Despite such design, failures can still occur at runtime due to unexpected system or environment behaviors. Fault-tolerant deals with building computing systems that continue to operate sufficiently in the presence of faults. A fault-tolerant system may be able to tolerate one or more fault-types including – (i) Transient, intermittent or permanent hardware faults, (ii) Software and hardware design errors, (iii) Operator errors (iv) Physical damage.

An extensive approach has been developed in this field over the past thirty years, and a number of fault-tolerant machines have been developed - most of them dealing with random hardware faults, while a smaller number deal with software, design and operator faults to uniform degrees. A large amount of supporting research has been reported. Fault tolerance and dependable systems research covers a wide spectrum of applications ranging across embedded real-time systems, commercial transaction systems, transportation Systems, and military/space systems - to name a few. The supporting research includes system architecture, real-time processing, design techniques, coding theory, testing, validation, and proof of correctness, modeling, operating systems, parallel processing, and software reliability.

These areas often involve widely diverse core expertise ranging from formal logic, mathematics of conditional modeling, software engineering, and hardware design and graph theory. Redundancy has long been used in fault-tolerant systems. However, redundancy does not inherently make a system fault-tolerant and adaptive; it is necessary to employ fault-tolerant methods by which the system can tolerate hardware component failures, avoid or predict timing failures, and be reconfigured with little or graceful decline in terms of reliability and functionality.

Early error detection is clearly important for real-time systems; error is an abstract for erroneous system state, the observable result of a failure. The error detection suspension of a system is the interval of time from the instant at which the system enters an erroneous state to the instant at which that state is detected. Keeping the error detection suspension small provides a better chance to recover from component failures and timing errors, and to display graceful reconfiguration. However, a small suspension alone is not sufficient; fault-tolerant methods need to be provided with sufficient information about the data processing underway in order to take appropriate action when an error is detected. Such information can be obtained during system design and implementation. In current practice, the design and implementation for real-time systems often does not sufficiently address fault tolerance and adaptiveness issues. The performance of a real time system can be improved by proper task allocation and an effective uniprocessor scheduling. In this thesis a brief study of the existing uniprocessor scheduling schemes and backup overloading techniques has been presented and work has also been done to find an appropriate schemes for allocation and scheduling in real time system.

The fault tolerant scheduling problem is defined as follows:

The fault tolerant scheduling requirements are given as follows: 1. Each task is executed by one processor at a time and each processor executes one task at a time.

2. All periodic tasks should meet their deadlines.
3. Maximize the number of processor failures to be tolerated.
4. For each task Ti, the primary task PriTi or the backup BackTi is assigned to only one processor for the duration of ci and can be preempted once it starts, if there is a task with early deadline than the presently executed task.

Suresh Gyan Vihar University, Jaipur
International Journal of Converging Technologies and Management (IJCTM)
Volume 1, Issue 1, 2015
ISSN : 2455-7528

Fault tolerance mechanism involves no of following steps, each step is associated with specific functioning hence they can be applied independently in the process of fault handling and the life cycle of fault handling is illustrated in Figure1.

- **Fault Detection: -** One of the most important aspects of fault handling in real time systems is to detect a fault immediately and isolate it to the appropriate unit as quickly as possible [3]. In system there is no central point for lookout from which the entire system can be observed at once [9] hence fault detection remains a key issue in real time system. [16] Reveals the impact of faster fault detectors in Real Time System. Design goals and architecture for fault detection in grid has been discussed in [13, 42]. Commonly used fault detection techniques are Consensus, Deviation Alarm and Testing.
- **Fault Diagnosis:**-Figure out where the fault is and what caused the fault for example Voter in TMR can indicate which module failed and Pinpoint can identify failed components.
- **Fault Isolation:**-If a unit is actually faulty, many fault triggers will be generated for that unit. The main objective of fault isolation is to correlate the fault triggers and identify the faulty unit and then confine the fault to prevent infection i.e. prevent it to propagate from its point of origin.
- **Fault Masking:**-Ensuring that only correct value get passed to the system boundary inspite of a failed component.
- **Fault Repair/Recovery:**-A process in which faults are removed from the system. Fault Repair/Recovery Techniques can be of the following types include Checkpoint and Rollback.
- **Fault Compensation:**-If a fault occurs and is confined to a subsystem, it may be necessary for the system to provide a response to compensate for output of the faulty subsystem.

During fault tolerance all of the above steps may not be involved.

1.2.1 Fault Types

There are three types of faults:

- Permanent,
- intermittent,
- transient.

A permanent fault does not die away with time, but remains until it is repaired. This is an intermittent fault cycle between the fault–active and fault benign states. A transient fault dies away after some time.

I. Past Stduy

Real-time systems are computerized systems with timing constraints. Real-time systems can be classified as hard real-time systems and soft real-time systems. In hard real-time systems, the consequences of missing a task deadline may be catastrophic. In soft real-time systems, the consequences of missing a deadline are relatively milder. Examples of hard real-time systems are space applications, fly-by-wire aircraft, radar for tracking missiles, etc. Examples of soft real-time systems are on-line transaction used in airline reservation systems, multimedia systems, etc. This thesis deals with scheduling of periodic tasks in hard real-time systems. Applications of many hard real-time systems are often modeled using recurrent tasks. For example, real-time tasks in many control and monitoring applications are implemented as recurrent periodic tasks. This is because periodic execution of recurrent tasks is well understood and predictable. The most relevant real-time task scheduling concepts are: periodic task system, ready (active) task, and task priority, and preemptive scheduling algorithm, feasibility condition of a scheduling algorithm, offline and online scheduling. **2.1.1 Types of Real-Time Systems**

Consider two categories into which real-time systems can be classified.

- Hard real-time systems are those whose failure (triggered by missing too many hard deadlines) leads to catastrophic. For example, if the computer on a fly-by-wire aircraft fails completely, the aircraft crashes. If a robot carrying out remotely commanded surgery misses a deadline to stop cutting in a particular direction, or a cancer treatment irradiation machine delivers too high a dose of radiation (by not switching off the beam at the right time), the patient can die.
- In a soft real-time system, missing any number of deadlines may be a cause of user annoyance; however, the outcome is not catastrophic. A server telling you the latest score in a cricket match may cause some users great distress by freezing at the most exciting point of the match, but that is not a catastrophe. An airline reservation system may take a very long time to respond, driving its more impatient customers away, but that may still not be classified as a catastrophe.

Thus, the classification of real-time systems into hard and soft depends on the application, not on the computer itself. The same type of computer, running similar software, may be classified in one context as soft while it is hard in another. For example, consider a video or audio-only conferencing application. When used for routine business communications, it can be considered a soft real-time system. If, on the other hand, it is used by police officers and firefighters to coordinate actions at the scene of a major fire, it is a hard real-time system. Or, take a real-time database system. When used to provide sports scores (as we have seen), it is soft; however, if it is used to provide stock market data, an outage can cause significant losses and may be regarded by its users as catastrophic. In this case, it would be considered a hard real-time system. Perhaps a good rule of thumb is to say that the designers of hard realtime systems expend a significant fraction of the development and test time ensuring that task deadlines are met to a very high probability; by contrast, soft real-time systems are really of the "best effort" variety, in which the system makes an effort to meet deadlines, but not more than that.

Note that the same computer system may run both hard and soft tasks. In other words, some of its workload may be critical and some may not be.

Most of the applications for which one requires fault-tolerant scheduling require hard real-time computers. Such systems run two types of tasks: periodic and aperiodic.

- As the term implies, a periodic task, Ti, is issued once every period of Pi seconds. Typically (but not always), the deadline of a periodic task is equal to its period: most of the results in realtime scheduling are based on assumption.
- An aperiodic task, can be released at any time; however, specifications may limit their rate of arrival to no more than one every τ seconds.

The majority of real-time systems are extremely simple, many built out of primitive eight-bit processors. It is not uncommon for such processors to have either no pipeline (meaning that they do not overlap instruction execution) or a very simple pipeline (lacking such features as out-of-order execution or branch prediction); most have no caches. Fault-tolerance features in many such applications are early at best.

In many important applications, however, the real-time system carries a heavy workload and is in the control of significant physical systems. Such systems have been used in aerospace applications for many years; more recently, they have begun to proliferate in other

hard real-time applications as well. These include industrial robots, chemical plants, cars, and mechanisms for remote surgery. It is these types of application for which the fault-tolerant scheduling approaches that we survey here are meant.

2.2 Periodic Task Systems

The basic component of scheduling is a *task*. A task is a unit of work such as a program or block of code that when executed provides some service of an application. Examples of task are reading sensor data, a unit of data processing and transmission, etc. A *periodic task system* is set of tasks in which each task is characterized by a period, deadline and worst-case execution time (WCET).

Period: Each task in a periodic task system has an inter-arrival time of occurrence, called the period of the task. In each period, a job of the task is released. A job is ready to execute at the beginning of each period, called the released time, of the job.

Deadline: Each job of a task has a relative deadline that is the time by which the job must finish its execution relative to its released time. The relative deadlines of all the jobs of a particular periodic task are same. The absolute deadline of a job is the time instant equal to released time plus the relative deadline.

WCET: Each periodic task has a WCET that is the maximum execution time that that each job of the task requires between its released time and absolute deadline.

If relative deadline of each task in a task set is less than or equal to its period, then the task set is called a constrained deadline periodic task system. If the relative deadline of each task in a constrained deadline task set is exactly equal to its period, then the task set is called an implicit deadline periodic task system. If a periodic task system is neither constrained nor implicit, then it is called an arbitrary deadline periodic task system. i.e

Relative deadline of task set <= period (constrained deadline periodic task system)

Relative deadline of constrained deadline task set = period (implicit deadline periodic task system)

2.3 Task Independence

The tasks of real-time applications may be dependent on one another, for example, due to resource or precedence constraints. If a resource is shared among multiple tasks, then some tasks may be blocked from being executed until the shared resource is free. Similarly, if tasks have precedence constraints, then one task may need to wait until another task finishes its execution. The only resource the tasks share is the processor platform.

2.1.3 Ready Tasks

In a periodic task system, job of a task is released in each period of the task. All jobs that are released but have not completed their individual execution by a time instant t are in the set of ready (active) tasks at time t. Note that, there may be no job in the set of ready tasks at one time instant or there may be a job of all the tasks in the set of ready tasks at another time instant.

2.4 Task Priority

When two or more ready tasks compete for the use of the processor, some rules are applied to allocate the use of processor(s). This set of rules is governed by the priority discipline. The selection (by the runtime dispatcher) of the ready task for execution is determined by the priorities of the tasks. The priority of a task can be static or dynamic.

- **Static Priority:** In static (fixed) priority, each task has a priority that never changes during run time. The different jobs of the same task have the same priority relative to any other tasks. For example, according to Liu and Layland, the well known RM scheduling algorithm assigns static priorities to tasks such that the shorter the period of the task, the higher the priority [2].
- **Dynamic Priority:** In dynamic priority, different jobs of a task may have different priorities relative to other tasks in the system. In other words, if the priority of jobs of the task changes from one execution to another, then the priority discipline is dynamic. For example, the well known Earliest-Deadline-First (EDF) scheduling algorithm assigns dynamic priorities to tasks such that a ready task whose absolute deadline is the nearest has the highest priority [2].

2.5 Preemptive Scheduling

A scheduling algorithm is preemptive if the release of a new job of a higher priority task can preempt the job of a currently running lower priority task. During runtime, task scheduling is essentially determining the highest priority active tasks and executing them in the free processor. For example, RM and EDF are examples of preemptive scheduling algorithm.

Under non-preemptive scheme, a currently executing task always completes its execution before another ready task starts execution. Therefore, in non-preemptive scheduling a higher priority ready task may need to wait in the ready queue until the currently executing task (may be of lower priority) completes its execution. This will result in worse schedulability performance than for the pre-emptive case.

2.6 Work-Conserving Scheduling

A scheduling algorithm is called work conserving if it never idles a processor whenever there is a ready task awaiting execution on that processor. A work conserving scheduler guarantees that whenever a job is ready to execute and the processor is free for executing the job is available, the job will be dispatched for execution. For example, scheduling algorithms RM and EDF are work-conserving by definition.

A non work conserving algorithm may decide not to execute any task even if there is a ready task awaiting execution. If the processor should be idled when there is a ready task awaiting execution, then the non work-conserving scheduling algorithm requires information about all tasks parameters in order to make the decision when to idle the processor. Online scheduling algorithms typically do not have clairvoyant information about all the parameters of all future tasks, which means such algorithms are generally work conserving.

2.7 Feasibility and Optimality of Scheduling

To predict the temporal behavior and to determine whether the timing constraints of an application tasks will be met during runtime, feasibility analysis of scheduling algorithm is conducted. If a scheduling algorithm can generate a schedule for a given set of tasks such that all tasks meet deadlines, then the schedule of the task set is feasible. If the schedule of a task set is feasible using a scheduling algorithm **A**, we say that the task set is A-schedulable. A scheduling algorithm is said to be optimal, if it can feasibly schedule a task set whenever some other algorithm can schedule the same task set under the same scheduling policy (with respect to for example, priority assignment, preemptivity, migration, etc.). For example, Liu and Layland [2] showed that the RM and EDF are optimal uniprocessor scheduling algorithm for static and dynamic priority, respectively.

Feasibility Condition (FC)

For a given task set, it is computationally impractical to simulate the execution of tasks at all time instants to see in offline whether the task set will be schedulable during runtime. To address this problem, feasibility conditions for scheduling algorithms are derived. A feasibility condition is a set of condition(s) that are used to determine whether a task set is feasible for a given scheduling algorithm. The feasibility condition can be necessary and sufficient or it can be sufficient only.

Suresh Gyan Vihar University, Jaipur
International Journal of Converging Technologies and Management (IJCTM)
Volume 1, Issue 1, 2015
ISSN : 2455-7528

Necessary and Sufficient FC (Exact test): A task set will meet all its deadlines if, and only if, it passes the exact test. If the exact FC of a scheduling algorithm **A** is satisfied, then the task set is **A**schedulable. Conversely, if the task set is **A**-schedulable, then the exact FC of algorithm **A** is satisfied. There-fore, if the exact FC of a task set is not satisfied, then it is also true that the scheduling algorithm can not feasibly schedule the task set.

Sufficient FC: A task set will meet all its deadlines if it passes the sufficient test. If the sufficient FC of a scheduling algorithm **A** is satisfied, then the task set is **A**-schedulable. However, the converse is not necessarily true. Therefore, if the sufficient FC of a task set is not satisfied, then the task set may or may not be schedulable using the scheduling algorithm.

2.8 Minimum Achievable Utilization

A processor platform is said to be fully utilized when an increase in the data processing time of any of the tasks in a task set will make the task set unschedulable on the platform. The least upper bound of the total utilization is the minimum of all total utilizations over all the sets of tasks that fully utilize the processor platform. This least upper bound of a scheduling algorithm is called the minimum achievable utilization or utilization bound of the scheduling algorithm. A scheduling algorithm can feasibly schedule any set of tasks on a processor platform if the total utilization of the tasks is less than or equal to the mini-mum achievable utilization of the scheduling algorithm.

II. Implementation

This experiment evaluates performance in terms of schedulability among the four algorithms, namely, OV, FGLS, FRCD and eFRCD, using the *SC* measure. The workload consists of sets of independent real-time tasks that are to be executed on a homogeneous distributed system. The size of the homogeneous system is fixed at 20, and a common deadline of 100 is selected. The failure rates are uniformly selected from the range between $0.5*10^{-6}$ and $3.0*10^{-6}$. Execution time is a random variable uniformly distributed in the range [1, 20]. Schedulability is first measured as a function of task set size as shown in Fig. 6.

Fig. 6 shows that the SC performances of OV and eFRCD are almost identical, and so are FGLS and FRCD. Considering that eFRCD had to be downgraded for comparability, this result should imply that eFRCD is more powerful than OV, because eFRCD can also schedule tasks with precedence constraints to be executed on heterogeneous systems, which OV is not capable of. The results further reveal that both OV and eFRCD significantly outperform FGLS and FRCD in SC, suggesting that both FGLS and FRCD are not suitable for scheduling independent tasks. The poor performance of FGLS and FRCD can be explained by the fact that they do not employ the BOV scheme. The consequence is twofold. First, FGLS and FRCD require more computing resources than eFRCD, which is likely to lead to a relatively low SC when the number of processors is fixed. Second, the backup copies in FGLS and FRCD cannot overlap with one another on the same processor, and this may result in a much longer schedule length. The number of Deadline= 100, and processor m=16 as constant for all the four algorithms.

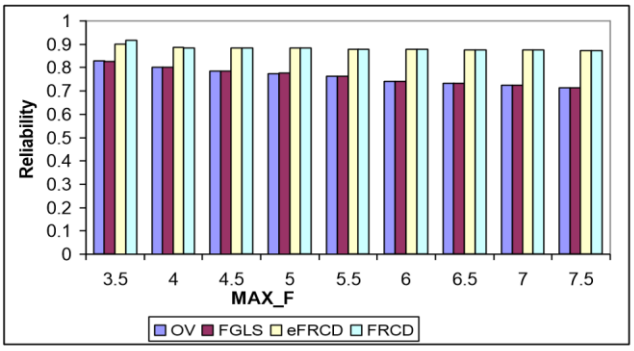

Figure 1: Reliability as function of MAX_F

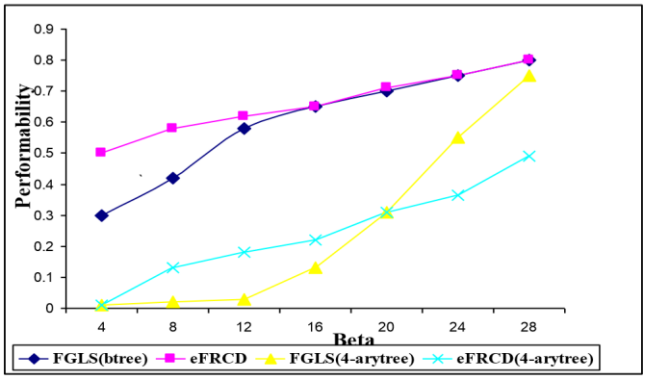

Figure 2: Performability of btree and 4-ary tree as a function of heterogeneity level

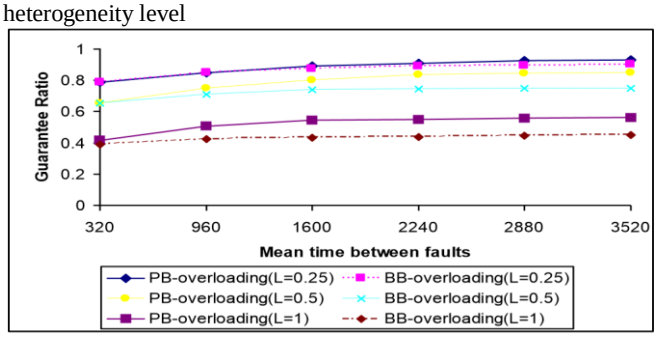

Figure 3: Effect of fault rate (N = 3; R = 4)

III. Conclusion

In this paper, an efficient fault-tolerant and real-time scheduling algorithm and overloading techniques for real time systems. To the best of our knowledge, the scheduling algorithms comprehensively address the issues of reliability, real-time, task precedence constraints, fault-tolerance, and heterogeneity. The performance analysis of OV [12], FGLS [5], FRCD[15] and eFRCD[9] are analyzed and found that eFRCD is the most relevant scheduling algorithm. The analysis also indicates that eFRCD is the superior to the rest of algorithms.

PB-Overloading technique uses dynamic grouping and is defined as the process of dynamically dividing the processors of the system into logical groups as tasks arrive into the system and finish executing. It is flexible in nature because we can take n number of processors and groups to use this overloading technique. BB-Overloading technique uses static grouping and where a processor can be a member of only one group. In PB-Overloading technique a processor can be a member of more than one group which allows efficient use of backup overloading. It also offers better schedulability than BB-overloading

technique due to its flexible nature of overloading backups. It is also analyzed that PB-overloading offer higher fault-tolerance degree than the BB-Overloading due to its dynamic nature of forming the groups. The primary backup overloading offers better schedulability and reliability to the system than BB-Overloading.

References

[1] S. Lauzac, R. Melhem, and D. Mossé. An Efficient RMS Control and Its Application to Multiprocessor Scheduling. In Proceedings of the InternationalParallel Processing Symposium, pages 511–518, 1998.

[2] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, 20(1):46 – 61, 1973.

[3] B. Andersson, S. Baruah, and J. Jonsson. Static-Priority Scheduling on Multiprocessors. In Proceedings of the IEEE Real-Time Systems Symposium, pages 193–202, 2001.

[4] Viacheslav izosimov. Scheduling and Optimization of Fault Tolerant embedded Systems, Ph.D.Thesis,Linkopings university, November 2006.

[5] A. Girault, C. Lavarenne, M. Sighireanu and Y. Sorel, "Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems," In Proc. of the 21st International Conference on Distributed Computing Systems(ICDCS), Phoenix, USA, April 2001.

[6] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In Proceedings of the IEEE Real-Time Systems Symposium, pages 166–171, 1989.

[7] T. P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. Real-Time Systems, 32(1-2):49–71, 2006.

[8] Akash Kumar,"Scheduling for Fault-Tolerant Distributed Embedded Systems",IEEE Computer 2008.

[9] Babaoglu, Ozalp, and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, Distributed Systems, pages 55–96. Addison-Wesley, 1993. [Cited at p. 27]

[10] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. The Computer Journal, 29(5):390–395, 1986.

[11] L. Lundberg. Analyzing Fixed-Priority Global Multiprocessor Scheduling. In Proceedings of the IEEE Real-Time Technology and Applications Symposium, pages 145–153, 2002.

[12] Y. Oh and S. H. Son, "Scheduling Real-Time Tasks for Dependability," Journal of Operational Research Society, vol. 48, no. 6, pp 629-639, June 1997.

[13] Paul Stelling, Cheryl DeMatteis, Ian T. Foster, Carl Kesselman, Craig A. Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. Cluster Computing, 2(2):117–128, 1999. [Cited at p. 27, 43]

[14] N. Audsley, A. Burns,M. Richardson, K. Tindell, and A.J.Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal, 8(5):284–292, 1993.

[15] X. Qin, H. Jiang, and D.R. Swanson, "A Fault-tolerant Real time Scheduling Algorithm for Precedence-Constrained Tasks in Distributed Heterogeneous Systems," *Technical Report TRUNL- CSE 2001-1003*, Department of Computer Science and Engineering, University of Nebraska-Lincoln, September 2001.

[16] Marcos K. Aguilera1, Gerard Le Lann, and Sam Toueg. On the impact of fast failure detectors on real-time faulttolerant systems. Springer-Verlag DISC 2002, pages 354 – 369, 2002. [Cited at p. 5, 27]

[17] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In Proceedings of the IEEE Real-Time Systems Symposium, pages 181–191, 1986.

[18] S. Baruah and J. Goossens. The Static-priority Scheduling of Periodic Task Systems upon Identical Multiprocessor Platforms. in Proc. of the IASTED Int. Conf. on PDCS, pages 427–432, 2003.