**Scheduling for Fault-Tolerant Hard Real-Time Systems**

Rogatiya Mohmad Aspak Arif (A0179741U)

Chee Vui Wong (A0163203W)

Department of ECE, National University of Singapore

[e0269760@u.nus.edu](mailto:e0269760@u.nus.edu), [e0146767@u.nus.edu](mailto:e0146767@u.nus.edu)

**ABSTRACT**

Real-time systems are increasingly being used in life-critical and complex systems where tasks usually have hard deadlines and missing of such deadlines can have a catastrophic effects. The reliability of a real-time system is measured in terms of probability of meeting the hard deadlines even in the presence of hardware of software failures. We will discuss about various scheduling strategies to achieve deadline in the presence of various faults, in uniprocessor and multiprocessor systems. Various scheduling algorithms presented in the papers [1-4] are studied and given critic comments. Few algorithms described in these papers are just limited to particular circumstances. This report is then concluded by comparing the various algorithms in terms of schedulability analysis, implementation complexity, application specific advantages, resource sharing, resource utilization, rejection ratio etc.

**INTRODUCTION**

The real-time systems are very important applications of computer systems. They are used in life critical applications such as aircraft control, patient health monitoring systems, chemical plants where making mistakes are not allowed. The hallmark about the real-time systems is that they usually have hard deadlines and missing of such deadlines can have catastrophic effects like death of a patient, road accidents or aircraft crash. Great efforts are done by researchers to develop strategies and techniques to guarantee the reliability of such systems even in the presence of hardware and software failures. One such approach is to have hardware redundancy i.e. to have more than required processors which can take over in the time of primary processor failures. But having massive hardware redundancy is not always a solution due to the fact that most embedded systems have to be effective in terms of cost, size and energy. Most of the hardware failures are transient in nature, that is why some sort of techniques are required to make sure the system works smoothly during the hardware outage or at least provides graceful degradation.

In this paper we will first give a brief introduction about the real-time systems, various processor failure modes and fault tolerance models. Then we will talk in details about various fault tolerant scheduling algorithms on uniprocessors and multiprocessor real-time systems.

**TYPES OF REAL-TIME SYSTEMS**

There are three categories of real-time systems: *soft, firm* and *hard*.

*Soft* real-time systems are those in which missing deadlines has a trivial effect on the system reliability, at most an annoying experience to the user and it doesn’t have any catastrophic effects. For example, weather monitoring system where the sensors are supposed to report the environment data after a predefined interval. Even when the sensors reports the data a little early or later than expected, the data still remains relevant. Another soft real-time system example is gaming console. Various software components of this game consoles are supposed to be in sync with each other. If the deadlines are met, the game will be enjoyable, else user will experience a little lag.

*Firm* real-time system allows infrequently missed deadlines and usually the output generated after the deadline is useless. For example manufacturing assembly lines with robots are used to assemble the parts. Missing a deadline may result in an improperly assembled product. As long as the quality control process is able to catch this part and parts are not too expensive, the system is still not considered as failed and the production goes on.

*Hard* real-time systems are those in which missing a deadline is considered a *System Failure,* and sometimes it has catastrophic consequences. For example, Air France flight 447 was crashed into Atlantic Ocean on 1 June 2009, due to pressure sensor malfunction. The pilots stalled the aircraft while responding to the outdated sensor data. Around 216 passengers and 12 crews were killed. Another example of a hard real-time system is an inkjet print head control software which is responsible to drop the ink onto a specific coordinates on paper. Missing a deadline causes the print job to be ruined. There are no catastrophic consequences here, but missing a deadline is considered as system failure. The correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical time instant at which these results are produced.

The classification of real-time systems into hard, firm or soft depends on the application and not much on the computer system being used. The same computer system, running the same software can be considered as soft in one context but as hard in another scenario. For example a video or audio conference software. Missing a deadline when used for personal or business communication purpose may cause an annoying experience to the end user. But the same system when used by firefighting team to monitor or taking actions at the scene of a huge fire, it is considered as a hard real-time system. The rule of thumb to classify a system as hard real-time is that designers have to put great effort to guarantee and test the probability of meeting the task deadlines. Please note that also the workload of the same real-time system can be categorized as hard and soft.

Hard real-time systems run two types of tasks: periodic and sporadic. A periodic task T*i* occur at a regular interval of P*i*. The deadline of such systems are typically assumed to be equal to its period P*i*. A sporadic task T*i* can be asserted any time, but usually RT limit the rate of arrival of such task to be no more than one every *τ* seconds.

**TASK SCHEDULING**

Real time task scheduling algorithms assume that Worst Case Execution Time (WCET) of the tasks are known in advance. Determining the WCET of tasks in modern real time system is not an easy job.

In practice many designer simply run the program with large amount of experimental data and find out the execution time, which is multiplied by a safety factor (~30%) to estimate WCET. Practically it is not guaranteed to find the perfect WCET even with very big amount of experimental data. Multiple level of cache, pipelining, %memory access times, threading etc. make it very difficult to determine the WCET of tasks.

Real time task schedulers can be categorized as offline or online schedulers. Offline schedulers are usually used for systems where most of the tasks are periodic and their arrival rate and hence the arrival times are known in advance. A successfully designed offline scheduler can guarantee that all tasks will meet the deadlines. An online schedule is generated during run time as the tasks arrive and hence it cannot guarantee that all tasks will meet the deadlines. When a new task arrives, the scheduler has to decide if it has enough time to execute the newly arrived task as well as the other tasks on hand, if yes then the meeting deadline is guaranteed, else the task will be rejected and the user has to do something about the rejected task. Real time tasks might have precedence constraints. The majority of fault tolerant scheduler assumes the tasks to be independent and hence no precedence constraints, unless specified.

There are two categories of schedulers: preemptive and nonpreamptive. Preemptive scheduler allows the higher priority task to preempt the lower priority task. Lower priority task will stop the execution, higher priority will take over and when finished, lower priority task will resume. Nonpreemptive schedulers preserves the processor for a task on hand and doesn’t allow to preempt the ongoing task if a higher priority task appears.

**FAULT TOLERANCE**

There are two types of faults in computer system: hardware and software faults. Hardware faults can be of transient nature or permanent. Transient failures are those which go away quickly. Permanent failures are those which do not go away over time and they may result into system failure. Redundancy in terms of hardware, time, information or software is used for the fault tolerance.

Triple Modular Redundancy (TMR) and Primary Backup are widely used hardware redundancy fault tolerance strategies. TMR is very expensive because it needs three copies of hardware to run a task on three different processor. Output from three hardware will be compared to mask the errors. TMR is only used for life-critical fault tolerant systems. TMR provides forward error masking by executing multiple copies of the tasks. While in PB, a backup copy is executed only if the primary copy fails. It is not necessary for the backup task to be the exact copy of the primary task. Instead, it might be the lighter version of the original task and it might generate an output of mere acceptable quality.

In a time-redundant fault tolerant system, the slack between the finish time of current task and its deadline is reserved to execute a backup copy if a fault occurs. The slack must be long enough for a backup copy to be scheduled and executed and still meet the deadline. Few systems use voltage scaling method to increase the clock rate and hence boost the execution during the slack time in order for the backup task to meet the deadline. Voltage is directly proportional to power and hence voltage scaling will increase the power consumption of such system. Fault is a rare occurrence and hence the fault tolerance using voltage scaling is used even in the system with power and energy constraints.

The dependability is system’s ability to deliver a quality service. The techniques to attain ability are grouped into four major categories: fault avoidance, fault tolerance, fault removal and fault evasion. *Fault avoidance* aims to reduce the chances of fault in system. Emphasis is given to design models and tools to identify potential faults during design phase itself. *Fault tolerance* aims at guaranteeing meeting the deadlines even in the presence of transient of permanent failures. *Fault removal* aims at identifying and removing faults from the system once the operational model of the system is ready. *Fault evasion* aims at estimating the future faults and identifying the possible consequences.

**FAULT TOLERANT SCHEDULING**

All fault-tolerant scheduling procedures aim to use time or processor redundancy to ensure that, despite the transient or permanent failure of a processor, the hard deadlines of critical real-time tasks will continue to be met. The target is to provide emergency response to a processor failure. In case of transient failures, such an emergency response is all that is necessary, because the faulty processor will soon recover and resume. In case of permanent failure, the system will have different emergency response depending on whether the scheduler is online or offline. In case of online schedulers, the incoming tasks will be either guaranteed to meet the deadline or rejected. The system will take in account the reduced computational resources while scheduling future tasks. In case of offline scheduler, the system will generate a new schedule considering one fewer processor available for computations. In case of software failures, a lighter version of the software is run to produce lower but acceptable quality of results.

**FAULT TOLERANCE ON UNIPROCESSORS**

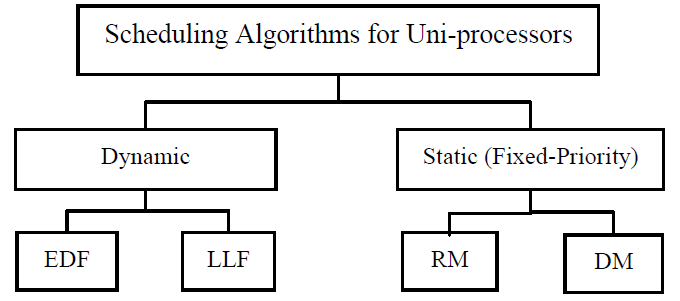


Figure 1: Scheduling algorithms in uniprocessors [3]

**Quasi-Static Scheduling [4]**

By comparing to static scheduling, Quasi Static scheduling for fault tolerance is able to guarantee meeting the deadlines for hard processes even in the case of faults and yet executing soft task with higher utility value. The main idea of quasi-static scheduling is to generate off-line a set of schedules, each adapted to a particular situation that can happen online.

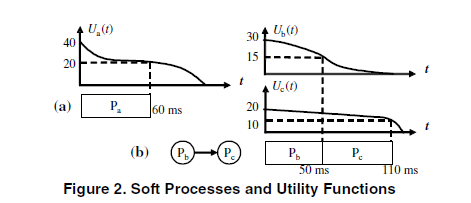


Figure 2: Soft processes and utility functions [4]

Each soft process is assigned with utility function, referring to Fig. 2a the soft process *P*a is assigned with a utility function *U*a (*t*) and completes at 60 ms. Thus, its utility would equal to 20. Sets of schedules is organized as a tree, where each node corresponds to a schedule.

As show in figure above, off-line a set of schedules is set for 4 groups

* If process *P*1 completes after 40, the scheduler switches to schedule *S*12, which will produce a higher utility.
* If the fault happens in process *P*1, the scheduler will switch to schedule *S*21 that contains the re-execution *P1/2* of process *P1*. In this time, schedule is switched in order to fulfil fault tolerance ability.
* In Fig. 5c., when fault is happened in P*1*, If the re-execution *P*1/2 completes between 90 and 100, the scheduler switches from *S21* to *S22*, that gives higher utility, and, if the re-execution completes after 100, it switches to *S23* in order to satisfy timing constraints.
* Schedule *S23* represents the situation illustrated in Fig. 4c2, where process *P3* had to be dropped. Otherwise, execution of process *P3* will exceed the period *T*. Note that we choose to drop *P3*, not *P2*, because this gives a higher utility value.

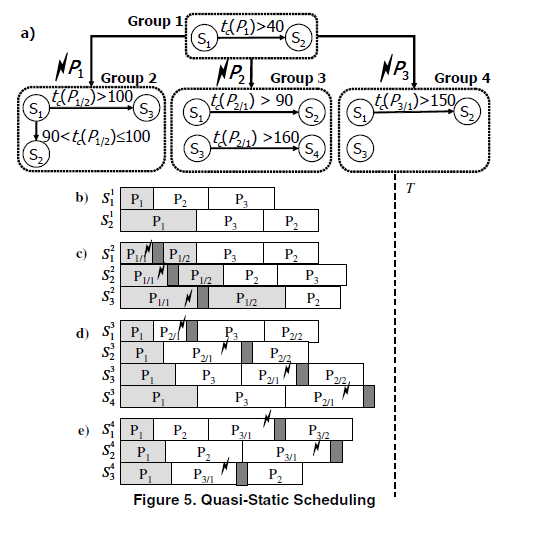


Figure 3: Quasi-static scheduling [4]

With Quasi-Static Scheduling, higher utility can be archived by comparing to Static Scheduling with handling of fault tolerance. Quasi static scheduling approach confirmed the deadlines of hard processes during the case of failure and yet schedule wisely base on severity of fault to archive high utility.

**RM (Rate Monotonic) Algorithm [3]**

Priority of task is based on period of task; higher priority is assigned to task with lower period. Duration of task is assuming within the period of task, Utilization of all task should be less than 100%. The utilization, U*i*, of task T*i* is C*i*/T*i*, there is natural amount of slack in uniprocessor, to guarantee fault handling feature, additional slack time must be added to ensure retry mechanism is able to re-execute before its deadline.

Referring to figure below, highest priority is P*2*, P*1*, P*4* and P*3*. Tasks are schedule to fulfil task periodicity and priority.

**DM (Deadline Monotonic) Algorithm [3]**

Tasks are scheduled base on deadline, the sooner the dead line the higher the priority. Referring to figure below, Priority of task is scheduled purely base on Deadline disregarding completion time, period and utilization. In case of fault on T1, T1 will be re-executed while task of T2, T3 and T4 will be push back which might causing missing deadlines.

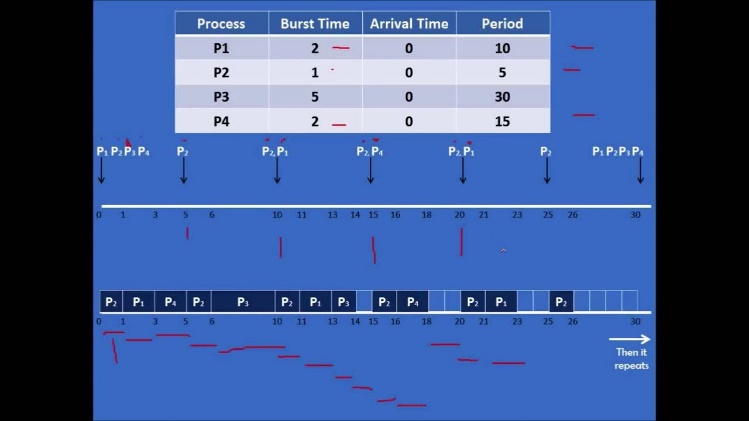
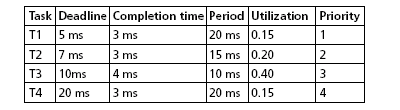


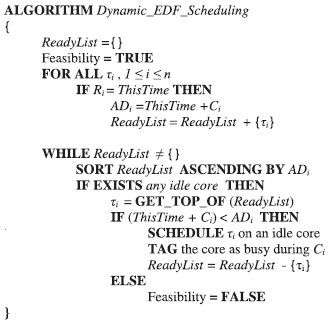
Figure 4: Example for deadline monotonic algorithm



With utilization factored is low “0.2”, DM algorithm gains benefit from passive duplication and small processors is required in order to guarantee fault tolerance. However, when utilization factor is increasing with more backup task is active performance of this algorithm is decrease. In conclusion FTDM algorithm can provide fault-tolerant schedules which require from 40% to 99% less processors than those used by the active duplication of RMFF scheduling.

**Earliest Deadline First [3]**

EDF is a dynamic algorithm which assigns the highest priority to a task with earliest absolute deadline.



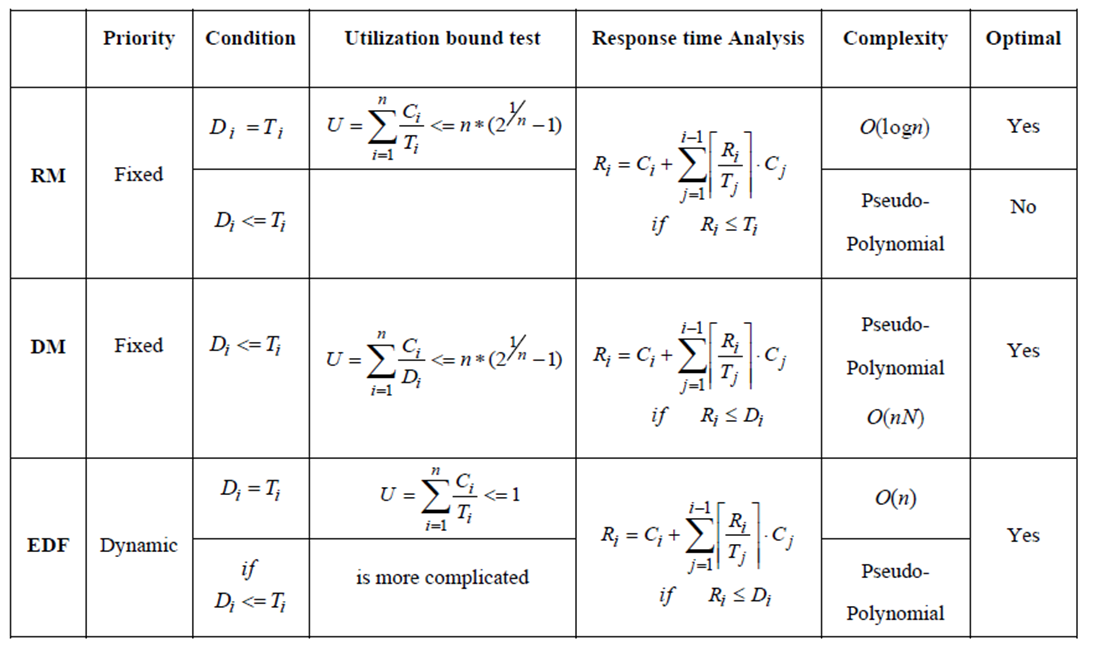
****

Figure 5: Comparisons between EDF, RM and DM algorithms [3]

**FAULT TOLERANCE ON MULTIPROCESSORS**

**Guarantee Algorithm [2]**

This algorithm aims to decide before the decision deadline if primary and backup of a task is schedulable on one of the available processor. If it is schedulable, then the task is guaranteed to meet the deadline, and it is rejected otherwise. The algorithm assumes the system and task model as *aperiodic hard real time tasks are to be scheduled on a multiprocessor system with identical processors*. The tasks are independent and do not share resources. A task request is specified by its arrival time t*a*, its worst-case execution time c, its relative deadline d*rel*, its absolute deadline d*abs* = t*a* + d*rel* and its decision deadline d*d*. A slot is the actual time interval on a processor’s schedule which is allocated to a certain task instance.

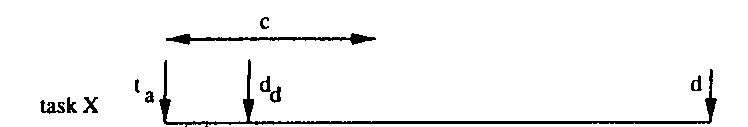


Figure 6: Task timing parameters

Fault-tolerance is achieved by the *primary backup* approach: for each task request two identical instances are scheduled on different processors within the relative deadline of the request. The instance in the earlier slot is called the primary Pr (i). It is executed if no failure occurs. The slot of the backup Bk (i) starts at some time after the end of the slot of the primary. The backup is only executed if the primary fails, otherwise it is deallocated - its slot is removed from the schedule - after the successful completion of the primary so that the capacity of the backup slot is again available for new requests.

Fault tolerant scheduling on a multiprocessor consists of few parts: primary and backup processor selection strategy, and primary as well as backup checking routines. There are three strategies for processor selection: sequential search, load-based selection and random processor selection. In load-based selection, the load is calculated in terms of WCET of tasks allocated and then added a safety factor of 0.3, then processors are sorted in the order of increasing load. The processor with the least load is selected to execute primary task T1, the second least load is selected as backup B1 for T1, the third least load is selected as primary for task T2 and so on. Once first primary and secondary are successfully scheduled, remaining primary and secondary has to be scheduled effectively. There are routines to schedule primary and secondary candidates.

The primary check algorithm checks for each gap between a specified starting time and the time [d*abs* – c] whether a primary of size c (WCET) may be placed into it. The check terminates with the first successful fit. The backup check algorithm places the backup in the slot [dabs - c] and makes no special effort to overload backups. There are modified checking routines to select primary and backup candidates.



Figure 7: Decision deadline to schedule primary task [2]

As shown in figure 7, Pr(X) is to be scheduled before the decision deadline. If it cannot be scheduled before that, it has to be rejected. Here Pr(X) can be scheduled on Processor 1, but it has to overlap of Bk (B). To do that, it has to wait for Pr (B) to be completed. Once Pr (B) is completed, the time assigned to Bk (B) can now be used for Pr(X). At time instance t2 which is fortunately before the decision deadline dd of task Pr(X). Each task has a time period after its arrival, in which the decision about its acceptance or rejection has to be taken. In above example the decision was taken to overlap primary of a task to overlap with a backup copy of another task which is completed. Example given below shows when decisions are taken to overlap a backup copy of task to in pipeline with another backup copy for which primary is already executed.

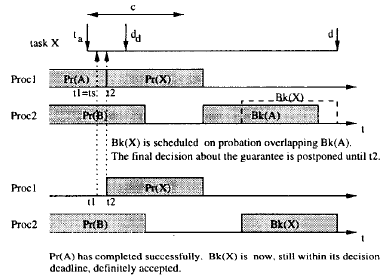


Figure 8: Decision deadline to schedule backup task [2]

As shown in figure 8, the decision was taken at time instance t2 before the decision deadline d*d* to schedule the backup of Pr(X) i.e. Bk(X) to overlap with Bk (A) because by the time instance t2, Pr (A) has been already executed.

In paper [2], the simulation plots are shown for the rejection ratio vs the execution time of algorithms with different strategies for the processor selection. Where rejection ratio is defined as the ratio of total requested tasks (rr*n*) vs total executed tasks (rr*c*). A discrete even simulator was used to simulate 100 sets of 1000 tasks. The simulation was done with an assumption that actually no fault has happened so all the backup copies can be deallocated. Window ratio in given simulation graphs is defined as *wr = drel/c,* where d*rel* is the relative deadline of the task. The decision deadline is assumed to minimum of 0.1 times the relative deadline or the slack available, i.e. *dd = min (0.1\*drel, drel-2\*c)*. The graph in figure 9 shows the simulation results for sequential search for processor selection. The graph in figure 9 also shows the simulation result for load based processor selection. And the graph in also shows the comparison between sequential search and load based algorithm. It is shown in the graph that sequential search performs upto 10% better with respect to rejected load.

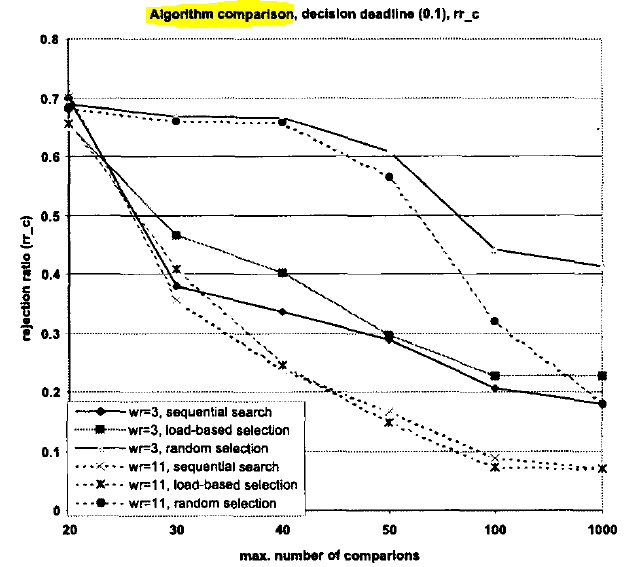
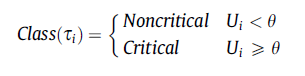


Figure 9: Sequential search vs load-based selection [2]

**Dynamic Fault Tolerance Scheduling (DFTS) Algorithm [3, 10]**

This algorithm uses task utilization to dynamically select the type of fault recovery method in order to tolerate the maximum number of multiple spatial and temporal faults. Each task is categorized into critical or noncritical based on the task utilization and the time at which scheduler allocates resources to the task. Noncritical tasks are scheduled on a single core, and check pointing with rollback recovery will be applied to them. Critical tasks will be replicated on separated cores to increase the probability of on time completion of the task in the presence of faults. Experimental results on several applications running on multi-core processors show that fault-tolerant scheduling feasibility rate of DFTS is higher than conventional methods for different fault rates and checkpoint costs. Moreover, the maximum fault-tolerance overhead is lower than the check pointing with rollback recovery method. The target is to fulfil the goals of reducing the overall execution time and increasing the scheduling feasibility.



The fault arrival rate depends on the environmental and operating conditions. Fault arrival rate can be in the range of 10-2 to 102 on the old processor architectures. On modern architectures, due to its complexities and dependencies, the fault arrival rates are much higher. This algorithm assumes a k-fault model, i.e. a maximum of k faults are allowed for each task. The problem of scheduling real time tasks in an NP-hard problem. Although EDF has a higher performance on single processors, the optimality of EDF reduces on a multiprocessor system.

There are trade-offs between applying frequent- and infrequent checkpoints for tasks in a system. Frequent *check-pointing* decreases re-execution time in the presence of faults, while task execution time is increased. On the other hand, infrequent check pointing has lower time overhead in the absence of faults, whereas the amount of re-execution will be increased if a fault is detected.

Although check pointing with rollback recovery methods have the advantage of reducing time overhead by re-executing only one part of the task in the presence of faults, they cannot utilize available spare resources (i.e. other processing cores in multicore systems) in order to reduce the schedule length. If the rate of fault occurrence on one processing core is high, a task needs more time to recover from faults, which means that the task is likely to miss its deadline. Hardware replication methods have the ability of parallel execution of the redundant copies of original tasks on the other processing cores. Hardware replication methods are generally classified into two distinct categories: (1) active replication in which all the task replicas are executed simultaneously and (2) passive replication in which the backup replicas are executed only if a fault occurs.

Dynamic fault-tolerant scheduling (DFTS), a hybrid method composed of time- and hardware-based redundancies is used; therefore, based on: (1) available hardware resources, (2) task utilization, and (3) expected number of faults for each task, the scheduler selects an appropriate fault-tolerant method.

As shown in figure, the hybrid or dynamic fault tolerance method helps to meet the deadline even in the presence of faults.

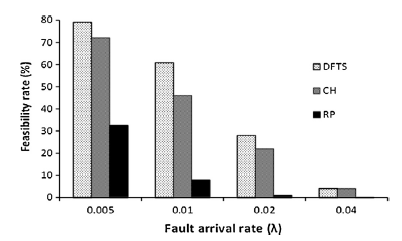


Figure 10: Fault arrival rate vs feasibility rate for hardware redundant, time redundant and hybrid (DFTS) approach. [10]

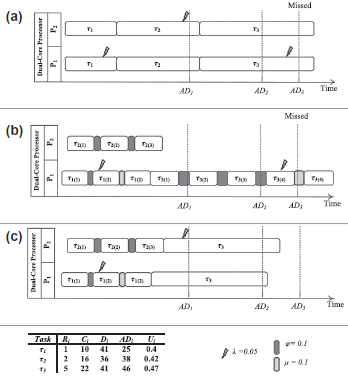


Figure 11: (a) Hardware redundancy and (b) time redundancy missing deadlines; (c) DFTS (Hybrid) approach meeting the deadlines [10]

**CONCLUSION**

Referring to Guarantee Algorithm that mentioned in *fault tolerance on* *multiprocessors*, task accepted is guaranteed to meet the deadline or it would be rejected. Notification of rejection is crucial, since this allowed application in decision making under fault circumstance that including providing warning to user, creating logs of diagnostic or reducing high precision calculation of existing task.

Referring to DFTS algorithm for multiprocessors, it was proven that simply increasing the hardware i.e. number of processor doesn’t guarantee to meet the deadlines. Time-redundant approach has better results compared to hardware redundancy. Hardware redundancy is also an expensive approach and not suitable for most commercial embedded system where price difference of even few hundred dollars has a drastic effect on profit.

Fault tolerance features is important for electronic device with the ability to self-recover and gracefully degradation due to transient or hardware fault. This guarantee the usefulness of military, medical grade devices. Besides that, for the application of industrial product e.g. boiler and furnace, fault detection mechanism should be enhanced with guaranteed functionalities and yet handling fault recovering should be disabled. With the “Design Idea” of stop the operation when fault is detected, this is to stop catastrophic disaster e.g. “explosion, chemical leakage”. With the intention to maintain On-time functionality and safety, redundant hardware is required with the architecture of primary-backup approach.

Referring to EDF, DM and RM algorithms on *uniprocessors*, EDF is comparatively *difficult to implement* compared to the latter two, because in EDF, decision has to be taken at each task activation. EDF has higher *runtime overhead*. While number of pre-emption will be higher in RM algorithms. Schedulability analysis is O(n) for EDF and pseudo-polynomial for RM. EDF has lower jitter compared to RM. Some sort of resource reservation has to be implemented for each algorithm if the resources are shared between tasks. EDF outperforms RM if the CPU usage is less than 100%. EDF exploits the maximum usage of computational elements and hence improving the *responsiveness* of the system. For future work, new algorithms can be implemented to switch between various algorithms dynamically depending on the type of faults and overloaded or under-loaded conditions.

**REFERENCES**

1. Mehta, Mr. Upasna, “Real Time System Fault Tolerance Scheduling Algorithms.”
2. Martin Naedele, “Fault-Tolerant Real-Time Scheduling under Execution Time Constraints.”
3. Reza Ravanmehr, “A Survey on Scheduling Approaches for Hard Real-Time Systems”
4. Viacheslav, Paul, Petru, Zebo, “Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints.”
5. Abhaya, Rajib, Chittaranjan, “Fault tolerant scheduling of hard real-time tasks on multiprocessor system using a hybrid genetic algorithm.”
6. Sunondo, Rami, Daniel, “Fault-Tolerance through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems.”
7. C.M. Krishna, “Fault-Tolerant Scheduling in Homogeneous Real-Time Systems.”
8. Risat Mahmud Pathan, “Scheduling Algorithms For Fault-Tolerant Real-Time Systems”
9. Risat Mahmud Pathan, “Fault-tolerant and real-time scheduling for mixed-criticality systems”
10. Mohammad M. Mottaghi, Hamid Zarandi, “DFTS: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors”

|  |  |
| --- | --- |
| Name | Work |
| Rogatiya Mohmad Aspak Arif | Survey of the fault tolerant scheduling algorithm for multiprocessors [1, 3]. Discuss algorithms with Chee Vui to compare the results with uniprocessor algorithms and come to a conclusion. |
| Chee Vui Wong | Survey of the fault tolerant scheduling algorithm for uniprocessor [2, 4]. Discuss algorithms with Aspak to compare the results with multiprocessor algorithms. |