Fault-Tolerant Real-Time Fair Scheduling on Multiprocessor Systems with Cold-Standby

Piyoosh Purushothaman Nair, Arnab Sarkar, *Member, IEEE*, and Santosh Biswas, *Member, IEEE*

Abstract—The ability to maintain functional and temporal correctness in the presence of faults is a key requirement in many safety-critical embedded systems. This work proposes an efficient fault recovery mechanism for real-time multiprocessor systems scheduled using a low overhead, semi-partitioned optimal proportional fair scheduling technique. We assume a system that can handle a single permanent processor fault at any time, using cold back-ups (with pre-specified activation / recovery time subsequent to the detection of a fault). As a result of the fault, the system may suffer transient overloads during such recovery periods, potentially leading to unacceptable fairness deviations and consequent rejections / early terminations of critical jobs. The proposed fault-tolerant scheduler, called *Fault Tolerant Fair Scheduler* (*FT-FS*), attempts to minimize such job terminations / rejections during recovery, by judiciously redistributing slacks accumulated by a subset of jobs, delivering more sustainable performance in the process. Experimental results reveal that the proposed *FT-FS* algorithm performs appreciably even under high system loads. Practical applicability of our proposed scheme has been illustrated using a case study on aircraft flight control system.

Index Terms—Deadline partitioning, fault-tolerance, proportional fairness, scheduling, real-time

1 Introduction

Real-time Safety-critical Systems. Real-time systems are characterized by their ability to respond to events that may happen in their operating environment, within stipulated temporal constraints. Thus, the correctness of these systems depends not only on the value of the computation but also on the time at which the results are produced [1]. In some real-time systems, failure to deliver correct outputs within stipulated deadlines can result in catastrophic consequences such as loss of life or property. Such real-time systems are designated to be safety-critical. Examples of safety-critical systems include fly-by-wire in aircrafts, pacemakers in health-care, anti-lock braking systems in automobiles, reactors in nuclear plants, etc. Reliability of a safety-critical system is typically guaranteed by enforcing stringent timing and fault-tolerance requirements. Given the availability of sufficient resources, successfully satisfying all temporal as well as fault-tolerance constraints of a set of real-time applications is essentially a scheduling problem.

Real-Time Scheduling on Multiprocessor Systems. Traditionally, scheduling of real-time applications (termed as tasks) on multiprocessors make use of either a partitioned or global approach [2]. In a partitioned approach, each task is assigned to a single designated processor on which it executes for its entire lifetime. This approach has the advantage of transforming the multiprocessor scheduling problem to a uniprocessor scheduling one. Hence, well known optimal uniprocessor scheduling approaches such as Earliest Deadline First (EDF), Rate Monotonic (RM) [1], etc. may be used. However, a major drawback of partitioning is that in the worst case, no more than half the system capacity may be utilized in order to ensure that all timing constraints are met [3]. Unlike partitioning, global and semi-partitioned scheduling schemes allow the migration of tasks from one processor to another during execution. Over the last three decades, a few global optimal schemes such as Pfair, ERfair, etc. and more recently, semi-partitioned optimal techniques like DP-Fair, have been proposed. All these scheduling approaches allow the possibility of utilizing the entire capacity of all processors in the system, resulting in high resource utilization.

Proportional Fair Scheduling. Most of these global and semi-partitioned scheduling strategies are based on the idea of proportional rate based execution progress for all tasks. Typically, such proportional fairness can be achieved by providing guarantees of the following form for each task: *complete X units of execution for application A out of every Y time units*. *Proportionate fair* (*Pfair*) scheduling introduced by Baruah et al. [4] is known to be the first optimal global scheduler for real-time repetitive tasks with implicit deadlines, on a multiprocessor system. Later, Anderson et al. [5] presented a work-conserving version of *Pfair*, called *Early-Release fair* (*ERfair*) scheduler, which never allows a processor to be idle in the presence of runnable / ready tasks. Since these global schemes attempt to maintain fair proportional progress for all tasks at all time slots, they may incur unrestricted preemption / migration overheads. More recently DP-Fair [6], an approximate proportional fair scheduler with a more relaxed execution rate constraint, was proposed.

DP-Fair Scheduling. DP-Fair, like ERfair is an optimal algorithm and enables full resource utilization. That is,

• *The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Assam 781039, India. E-mail: {piyoosh, arnabsarkar, santosh_biswas}@iitg.ac.in.*

Manuscript received 6 Sept. 2018; revised 4 July 2019; accepted 1 Aug. 2019. Date of publication 14 Aug. 2019; date of current version 9 July 2021. (Corresponding author: Santosh Biswas.)
Digital Object Identifier no. 10.1109/TDSC.2019.2934098

given n tasks and m processors, schedulability is ensured provided,

$$\sum_{i=1}^{n} e_i/p_i \leq m, \quad (1)$$

where, e_i and p_i denote the worst case execution time and period of a task T_i, respectively. In DP-Fair, time is partitioned into slices, demarcated by the deadlines of all jobs in the system. Within a time slice, each task is allocated a workload equal to its proportional fair share and assigned to one or two processors for scheduling. Job subtasks within a slice are typically scheduled using variations of traditional fairness ignorant schemes such as Earliest Deadline First (EDF [7]). Through such a scheduling strategy, DP-Fair is able to deliver optimal resource utilization while enforcing strict proportional fairness (ERfairness) only at period / deadline boundaries. DP-Fair is a semi-partitioned scheduling technique which allows at most $m-1$ task migrations and $n-1$ preemptions within a time slice and thus incurs much lower overheads compared to ERfair. In this work, we use a discrete approximation of DP-Fair as the underlying scheduling mechanism.

Fault-Tolerant Real-time Systems. In addition to satisfying timing constraints, today we observe an increasing emphasis towards fault-tolerant real-time systems which need to ensure functional correctness in the presence of *permanent / transient faults* [8], [9], [10], [11]. Two major approaches for achieving fault-tolerance are *time* and *hardware redundancies* [9], [12]. *Checkpointing* is an important time redundancy based fault-tolerance scheme [13], [14] where the current state of the task is saved periodically. On the occurrence of a failure,[1] the saved internal state of the task is restored and execution resumes from this saved state. On the other hand, important hardware redundant strategies include *N-modular redundancy* and use of *standby spares* [9], [15]. In N-modular redundancy, multiple units running in parallel execute redundant copies of the same workload and mask errors by voting on their outputs. In the standby spares approach (also called *backup redundancy*), the fault affected primary unit is replaced by an identical secondary unit subsequent to a fault.

Hot and Cold Standby Sparing. Two major standby sparing techniques used to achieve fault-tolerance in real-time systems are *hot-standby* and *cold-standby* [16], [17]. In hot-standby, the backup unit runs concurrently with the primary unit and so, there is no delay in replacing the faulty primary with the backup. However, the resource demands for such a system could be about twice or even more (depending on the degree of fault-tolerance desired) compared to a system without hot-standby sparing. On the other hand in cold-standby, the backup becomes operational only after a fault in the primary is detected. As the primaries and backups do not execute concurrently, additional resource demands that are necessary to achieve fault-tolerance in cold-standby systems are typically far lower compared to systems with hot-standbys. However in this case, recovery to the nominal system state after failure requires a finite amount of time called *recovery time*, to replace the faulty primary [16], [17]. In addition to lower resource demands, another important advantage of cold-standby sparing is that fault-tolerance do not necessitate extra power for running spares, during normal operation. Hence, this scheme may be useful for systems where power consumption is an important design constraint. Moreover, a system equipped with cold-standby spares has lower overall operational costs and higher life times compared to a hot-standby system.

Multi-Criticality Systems. Many real-time systems support the execution of applications with different relative importance values (some times called *criticality levels*) on a common platform [18]. For example, in modern avionics systems, flight control tasks (responsible for the vehicle's safety) are considered to be more safety-critical than military mission tasks (responsible for say, firing a missile at the enemy target). Scheduling decisions in these scenarios must consider the criticality levels of applications. Particularly, in times of overload the objective is usually to allow the execution of the highest critical tasks in the system while rejecting the least critical ones, such that the overload condition can be mitigated [18], [19].

Motivation of the Proposed Work. In spite of the additional flexibility and considerably higher resource utilizations that cold-standby sparing can potentially achieve within a global scheduling scenario, currently there does not exist any significant research that attempts at such a design approach. This is primarily because of the following two major challenges related to global scheduling with cold-standby: i) Efficiently managing task executions both during nominal system operation as well as during the recovery interval, so that all timeliness constraints can always be guaranteed over the entire schedule length, ii) Efficiently handling the execution progress of tasks while having an upper bound on both preemptions and migrations so that their overheads can be accounted even in the face of faults. In this paper, we propose a novel combined time-cum-hardware redundancy based fault-tolerant semi-partitioned scheduling strategy called *Fault Tolerant Fair Scheduler* (*FT-FS*) for real-time multiprocessor systems. In order to appropriately handle both the design challenges just mentioned, a work conserving version of the DP-Fair algorithm has been used as the underlying scheduling methodology. Thus, the proposed scheduler is able to effectively combine the benefits of high resource utilization and bounded context switches of DP-Fair along with all the advantages of cold-standby sparing as discussed above.

Proposed Scheduling Methodology. In our work, we assume a system that can handle at most one permanent processor fault at any given time using a single *cold-standby* spare that takes a fixed *recovery time* (say, t_r) to attain operational state after the detection of a fault. Thus, during any recovery period subsequent to a failure, the system is forced to work with one less processor resource. The primary objective of our fault-tolerance mechanism is to satisfy all DP-Fairness based timeliness constraints during any recovery interval, t_r, given the total workload to be handled. To achieve this, *FT-FS* utilizes the slacks of already over-allocated tasks in an effort to overcome possible transient overloads caused by the fault. However, if *FT-FS* estimates the possibility of fairness violations at any task deadline boundary within t_r, it rejects a minimal number of the least important tasks in

1. For simplicity, and with a slight abuse of terminology, the terms fault and failure have henceforth been used interchangeably.

order to keep the rest of the system operational with graceful performance degradation (called *fail-operational* [20], [21]) during the recovery period. For this mechanism to work, we assume that two successive processor failures are at least separated by t_r, the recovery interval.

Paper Organization. Rest of the paper is organized as follows. In the next section, we present important state-of-the-art works related to this paper. The system model under consideration along with the problem formulation are presented in Section 3. Section 4 presents the proposed Fault Tolerant Fair Scheduling algorithm with separate subsections detailing i) the working of the scheduler in normal mode, ii) fault model considered, iii) the scheduling methodology followed during the recovery period subsequent to fault, and iv) the complexity analysis of *FT-FS*. The experimental framework along with a discussion on the obtained results are presented in Section 5. Section 6 discusses a case study using an automated flight control system to illustrate the applicability of our fault recovery mechanism in real world scenarios. We conclude in Section 7.

2 Related Works and Contributions

Fault-tolerant scheduling approaches for handling transient as well as permanent processor failures in multiprocessor systems have received a lot of attention in the last two decades [10], [22], [23], [24], [25], [26]. Time and hardware redundancies as well as a combination of both, are the major approaches towards achieving fault-tolerance in real-time systems. A detailed survey on different fault-tolerant scheduling schemes for homogeneous real-time multiprocessor systems may be found in [10]. In this section, we discuss a few important fault-tolerant scheduling approaches in detail.

Time Redundancy Based Scheduling Schemes. Time redundancy based approaches use the slack capacity available in an underloaded system to achieve fault-tolerance [10]. Important time redundancy based scheduling strategies include re-execution [24], [25] and checkpointing with rollback recovery [13], [14]. In the re-execution scheme, whenever a fault is detected, the faulty task is either re-executed from the beginning or a different version of the task, called recovery block, is executed in order to recover from the fault. Pathan and Jonsson [24] presented a re-execution based time redundant fault-tolerance scheme for scheduling a set of fixed-priority sporadic tasks on multiprocessors, to tolerate multiple permanent as well as transient failures. They also presented a feasibility test that can be used to ensure satisfaction of all deadlines even in the presence of processor failures and task errors. Recently, Pathan [25] extended the fault-tolerant framework presented in [24] to incorporate probabilistic schedulability guarantees, resulting in the probabilistic satisfaction of individual task deadlines. It may be noted that the works presented in [24], [25] use a fixed priority scheduling scheme which may possibly result in significantly lower resource utilizations compared to dynamic priority schemes. In addition, the preemptive nature of these global multiprocessor scheduling policies make them potentially susceptible to unrestricted preemptions and migrations. Unlike the re-execution based approach, checkpointing involves periodically saving the intermediate states of a task during its execution. On the occurrence of a failure, the latest saved internal state of the task is restored and execution resumes from this saved state. However, saving checkpoints has an associated cost in terms of both time and space, and hence, checkpoints when taken too frequently may lead to significant overheads. In [14], El-Sayed and Schroeder provided an extensive analysis of the performance, energy and I/O costs associated with a wide array of checkpointing policies. Checkpointing schemes must not only maintain an account of overheads, but also be aware of the available slack capacity at all times so that the increase in overall execution cost do not lead to deadline violations.

Hardware Redundancy Based Scheduling Schemes. Hardware redundancy based approaches incorporate extra hardware into the design to either detect or override the effects of failed components [8]. Important hardware redundancy based scheduling strategies include N-Modular Redundancy (NMR), Primary/Backup (PB), and Standby Sparing (SS) [10]. In NMR, multiple copies of a hardware resource running in parallel execute redundant copies of the same workload and mask errors by voting on their outputs. Since NMR requires multiple hardware units, it is expensive and used only in very critical fault-tolerant systems. In the PB approach, each task is considered to have one primary copy and one or more backup copies. A backup copy may be active or passive. An active backup always executes along with its primary while a passive backup is activated only after the primary fails. In the SS approach, one or more spare processors are maintained as standby and the fault affected primary processor is replaced by an identical secondary unit subsequent to a fault. A majority of the PB based approaches often utilize partition-oriented scheduling schemes to assign the primary and backup copies of tasks onto distinct processors. Whenever a transient or permanent processor failure is detected at run-time, the outputs of the backup copies assigned on non-faulty processors are considered as the correct outputs. As a consequence of partitioning, achievable resource utilizations with these approaches may be considerably lower compared to global schemes [3]. In [22], Al-Omari et al. proposed an adaptive primary-backup based fault-tolerant scheme to schedule soft real-time aperiodic tasks on multiprocessor systems. By considering the dynamics of faults and task parameters in the system, they provided a mechanism which controls the degree of overlap between the primary and backup versions of tasks within the schedule. Kim et al. [23] presented a fault-tolerant task allocation strategy called R-BFD (Reliable Best-Fit Decreasing) which allocates active backups in such a way that a primary and its active backups are not assigned on the same processor. In their work, they extended R-BFD by proposing another task allocation algorithm called R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) which reduces the resource over-provisioning costs using passive backups. Recently, Bhat et al. [26] presented a system model in which each task is characterized by an application-specific constraint called *recovery time requirement* (RTR). RTR specifies the number of consecutive deadlines of the primary task that a backup can afford to miss without the system being considered to have failed. The authors then use this RTR based model and extended the fault-tolerant task allocation problem discussed in [23]. They presented

TABLE 1
A Few Important Notations

Symbol	Meaning
t_p	Periodic safety check point interval
t_r	Recovery interval
F_{OT}	Fault occurrence time
F_{DT}	Fault detection time
F_{RT}	Fault recovery time
ts_l	lth time slice
tsl_l	Length of lth time slice
wt_i	Weight of a task T_i
ewt_i	Effective weight of a task T_i
sh_i^l	Share of a task T_i in lth time slice

different task allocation strategies which satisfy the recovery time requirements of all tasks while attempting to optimize resource utilization. However, determining the optimal resource over-provisioning required to efficiently implement a PB based scheme is a complex problem and requires careful design.

Time-Cum-Hardware Redundancy Based Scheduling Schemes. A few fault-tolerant scheduling mechanisms based on a combination of time and hardware redundancies may also be found in the literature. Mottaghi et al. [27] presented a fault-tolerance mechanism which dynamically selects either hardware redundancy or checkpointing with rollback recovery based on the criticalities of tasks. In [13], Kang et al. presented a fault-tolerant scheme which utilizes dual modular redundancy (DMR) and checkpointing to detect and correct transient faults that may corrupt the result for a given application modelled as a task graph. They determined the optimal points in the schedule where checkpoint should be placed while taking into account the associated overheads.

It may be noted that most of the existing fault-tolerant approaches make use of *either a fully partitioned or fully global scheduling policy.* Partition oriented schemes typically tend to suffer from *low resource utilizations* while the use of global policies may lead to *unrestricted preemptions and migrations.* Furthermore, the liberal use of hot-standby based hardware redundancy makes many of the existing works *susceptible to high resource over-provisioning costs and added power consumption.* Therefore, hot-standby based schemes may not be useful for systems where power consumption is an important design constraint.

Contributions. In this paper, we propose a *novel time-cum-hardware redundancy* based fault-tolerant scheduling strategy called *FT-FS*, for real-time multiprocessor systems. Unlike most of the state-of-the-art works which are either fully global or fully partitioned, *FT-FS follows a semi-partitioned scheduling approach.* This enables it to deliver high resource utilization (like global approaches) while being able to provide bounds on context switch overheads (like partition-oriented schemes). Being based on DP-Fair, *FT-FS also guarantees at most* $m-1$ *task migrations and* $n-1$ *preemptions within a time slice.* By employing cold-standbys, *FT-FS is able to achieve significantly better resource usage and power efficiencies* compared to hot-standby based techniques. However like any cold-standby based scheme, *FT-FS* must also deal with a recovery period subsequent to a fault, when one less processor resource is available. In its strive to remain fail-operational during recovery, *FT-FS* employs a *novel slack donation scheme* from overallocated to underallocated tasks with the objective of maximizing resource utilization. In times of uncontrollable overloads during recovery when task rejection becomes inevitable, the proposed framework takes care to reject a minimum number of the least critical tasks. The scheduler also ensures that all tasks which execute through the recovery period have progressed by their prescribed amounts at the end of the recovery period when the spare processor gets activated. This empowers the framework to handle fresh faults immediately after recovery. Hence in this work, two consecutive processor faults only need to be separated by just the recovery interval, and this duration is typically small especially in closely coupled systems. Being based on DP-Fair, *FT-FS also allows dynamic task arrivals during its nominal mode of operation.* The generic applicability of the proposed fault-tolerant scheduling mechanism in real world scenarios has been illustrated using a case study on an automated flight control system. Finally, the overall efficacy of the presented work has been exhibited through detailed analysis and simulation based experimental evaluation.

3 System Model and Problem Formulation

We consider a real-time multiprocessor system consisting of a set of n periodic tasks $T = \{T_1, T_2, \ldots, T_n\}$, to be executed on a set of m homogeneous processors $V = \{V_1, V_2, \ldots, V_m\}$. We assume a discrete time line where the interval $[t, t+1)$ is referred to as a time slot t ($t \in \mathbb{N}$). Each task T_i in set T is defined by a 3-tuple (e_i, p_i, cr_i), where, e_i denotes the *worst case execution time requirement* (WCET) of each instance/job of T_i, p_i denotes the fixed inter-arrival time between consecutive instances (referred to as *period*) and cr_i measures the relative importance of T_i with respect to other tasks (denoted as the *criticality level* of T_i; cr_i takes an integer value in the range [1,100]). Each task T_i is associated with a *weight* (wt_i), which is defined as the ratio of its execution requirement (e_i) and period (p_i). Each processor V_j ($\in V$) has unit capacity.

In the nominal mode of operation, the system periodically checks for a processor fault, every t_p time slots. There is a cold standby processor which is activated if and when there is a permanent processor fault. The standby processor requires a finite time, called *recovery time* t_r to become operational. Let, t_i be the time instant in the past at which the system last checked for a fault. Let, $F_{OT}(= t_i + t_f)$, be the instant at which the fault actually occurs before the next periodic check at $F_{DT}(= t_i + t_p)$ and so, the system recovers at the instant $F_{RT} = F_{DT} + t_r$. We assume that the system can handle at most one permanent processor fault at any given time; no further faults are assumed to occur during the recovery period $[F_{DT}, F_{RT})$. A few important notations used in the later sections have been listed along with their meanings in Table 1.

Problem Formulation. Given a set of n real-time periodic tasks and m homogeneous processors, design an efficient fault recovery mechanism which attempts to satisfy all DP-Fairness based timeliness constraints during any recovery period subsequent to a permanent processor failure.

4 Fault Tolerant Fair Scheduler (FT-FS)

In this section, we describe the overall working of proposed *FT-FS* scheduler.

Algorithm 1. Fault Tolerant Fair Scheduler: *FT-FS*

Input: T: Task set; m: number of processors; t_p: periodic safety check point interval; t_r: recovery interval; tsl_l: length of the time slice (ts_l) currently being scheduled
Output: Generate and execute schedules of task instances

1 Initialize $t = next\_slice = 0$; $check\_point = t_p$;
2 **while** *TRUE* **do**
3 **if** $t = check\_point$ **then**
4 **if** a fault is detected at time t **then**
5 Interrupt all processors;
6 $next\_slice = t + t_r$;
7 $check\_point = next\_slice + t_p$;
8 Call function *FT-FS_Faulty()* and wait until completion;
9 **else**
10 $check\_point = check\_point + t_p$;
11 **if** $t = next\_slice$ **then**
12 Call function *FT-FS_Normal()*;
13 Execute the generated schedule on each processor in parallel;
14 $next\_slice = next\_slice + tsl_l$;
15 $t = min(check\_point, next\_slice)$;

The pseudocode of the *FT-FS* scheduler is presented in Algorithm 1. Similar to DP-Fair, *FT-FS* partitions time into slices, demarcated by the periods / deadlines of all tasks in the system. The duration between any two consecutive deadlines (say, the $(l-1)$th and lth deadlines) is referred to as a time slice ts_l and tsl_l denotes the length of ts_l. The entire *FT-FS* scheduler essentially executes within a while loop (lines 3-16) which continues until the system stops. Within the loop, *FT-FS* checks whether the value of timer t coincides with the next check pointing instant (variable $check\_point$). If it does and a fault is detected, all processors in the system are interrupted and the scheduler enters into *fault mode* (by calling the function *FT-FS_Faulty()*; refer Algorithm 3) for a duration t_r which denotes the recovery interval. In the absence of a fault, it simply updates $check\_point$ to next periodic check point instant.

If timer t coincides with the time slice boundary (variable $next\_slice$), *FT-FS* first invokes *FT-FS_Normal()* (refer Algorithm 2) to generate a *work conserving schedule* for the ensuing time slice. Then it executes tasks in the time slice, based on the generated schedule. Finally, before proceeding to the next iteration of the loop, *FT-FS* updates timer t to the earlier between the next check point instant and the next time slice boundary. We now discuss the function *FT-FS_Normal()* in detail. Function *FT-FS_Faulty()* is discussed in Section 4.3.

4.1 FT-FS: Normal Mode of Operation

The pseudocode of the function *FT-FS_Normal()* is presented in Algorithm 2. At the beginning of time slice ts_l (say, at time t), each task T_i is allocated a share sh_i^l (proportional to its weight) to be executed within ts_l and is calculated as follows:

Algorithm 2. Function *FT-FS_Normal()*

Input: R_l: Active task set, $\bar{e}_i$: the remaining execution requirement of each task T_i ($\in R_l$)
Output: Generate schedule for time slice ts_l

1 **for** *each task* T_i in R_l **do**
2 $sh_i^l = min(ewt_i \times tsl_l, \bar{e}_i)$ {refer Equation (3)};
3 $\bar{e}_i = \bar{e}_i - sh_i^l$;
4 $scap = m \times tsl_l - \sum_{i=1}^{|R_l|} sh_i^l$ {refer Equation (6)};
5 **if** $scap > 0$ **then**
6 **for** *each task* T_i in R_l **do**
7 **if** $\bar{e}_i > 0$ **then**
8 Determine T_i's urgency factor uf_i, and update its share sh_i^l and $\bar{e}_i$ {refer Equation (7)};
9 Recalculate $scap$ using Equations (4) and (6);
10 **if** $scap > 0$ **then**
11 Create a list lt of tasks for which $\bar{e}_i > 0$, sorted in non-increasing order of their $lag(T_i, t + tsl_l)$ values;
12 **for** *each task* T_i *in* lt **do**
13 $sh_i^l = sh_i^l + 1$, $\bar{e}_i = \bar{e}_i - 1$, $scap = scap - 1$;
14 **if** $\bar{e}_i = 0$ **then**
15 Remove T_i from list lt;
16 **if** $scap = 0$ **then**
17 exit;
18 After finalizing the shares of all tasks in R_l generate schedule for time slice ts_l using McNaughton's wrap-around rule [28].

Let, R_l denote the set of active tasks at time t. The total workload within time slice ts_l becomes: $L = \sum_{i=1}^{|R_l|} wt_i$. Given L, the *effective* execution rate (also referred to as *effective weight*) of each task T_i at time t is calculated as:

$$\forall T_i \in R_l, ewt_i = min\left(\frac{\mathcal{M}}{L} \times wt_i,\ 1\right), \tag{2}$$

where, $\mathcal{M} = m$, is the number of available processors. *FT-FS_Normal()* first determines an initial share for each task T_i in R_l as:

$$\forall T_i \in R_l, sh_i^l = \lfloor min(ewt_i \times tsl_l,\ \bar{e}_i) \rfloor, \tag{3}$$

where, $\bar{e}_i$ is the currently remaining execution requirement of T_i's current instance. Let, sum_shr^l denote the sum of shares of all tasks within time slice ts_l. Thus,

$$sum\_shr^l = \sum_{i=1}^{|R_l|} sh_i^l. \tag{4}$$

Given sum_shr^l, there exists a *feasible* schedule within ts_l only if,

$$sum\_shr^l \leq \mathcal{M} \times tsl_l. \tag{5}$$

If $sum\_shr^l < \mathcal{M} \times tsl_l$, there exists some spare capacity (denoted as *scap*) within time slice ts_l:

$$scap = \mathcal{M} \times tsl_l - sum\_shr^l. \tag{6}$$

Now, *FT-FS_Normal()* proportionally distributes this residual capacity $scap$ among all tasks in R_l for which $\bar{e}_i > 0$. Thus, the modified task shares become:

$$\forall T_i \in R_l, sh_i^l = sh_i^l + \lfloor min(scap \times uf_i,\ \bar{e}_i) \rfloor, \tag{7}$$

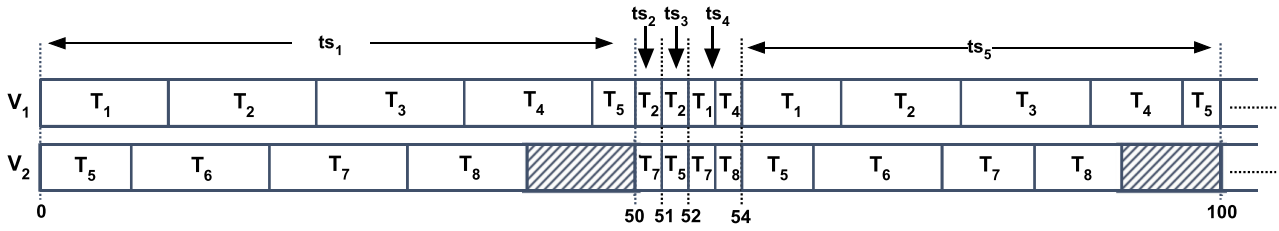

Fig. 1. *FT-FS* schedule for first 100 time slots.

where, uf_i is termed as the *relative urgency factor* of T_i and is defined as:

$$uf_i = \bar{e}_i/\bar{p}_i \Big/ \sum_{i=1}^{|R_l|} \bar{e}_i/\bar{p}_i. \tag{8}$$

Here, $\bar{p}_i$ is the currently remaining time before the completion of T_i's period. Equation (7) ensures that the fraction of the residual capacity allocated to task T_i is proportional to its relative execution urgency uf_i. If there still remains some residual capacity after updating task shares, *FT-FS_Normal()* creates a list lt of tasks for which $\bar{e}_i > 0$, sorted in non-increasing order of their $lag(T_i, t + tsl_l)$[2] values. Here, $lag(T_i, t + tsl_l)$ denotes the lag of task T_i at the end of ts_l (at time $t + tsl_l$), assuming T_i to have executed its share sh_i^l in ts_l. Thus, $lag(T_i, t + tsl_l) = \frac{e_i}{p_i} \times (t + tsl_l) - (allocated(T_i, t) + sh_i^l)$.

Now, *FT-FS_Normal()* sequentially chooses the next task in lt and increases its share by 1 until the residual capacity $scap$ is exhausted. After this, a schedule for all tasks in R_l is generated corresponding to time slice ts_l, using McNaughton's wrap-around rule [28]. Based on this generated schedule, tasks allocated to each processor are executed in parallel until completion of the time slice.

Example 1. Consider a set of eight periodic tasks, T_1 $(11, 52, 1)$, T_2 $(13, 50, 2)$, T_3 $(13, 54, 3)$, T_4 $(11, 52, 4)$, T_5 $(10, 51, 5)$, T_6 $(11, 54, 6)$, T_7 $(11, 50, 7)$ and T_8 $(10, 52, 8)$ to be executed on two unit capacity processors ($m = 2$) using the *FT-FS* scheduling scheme. Fig. 1 depicts the *FT-FS* schedule for the first 100 time slots.

The time slice boundaries corresponding to the above task set occur at time instants $50, 51, 52, 54$ and 100. Therefore, $tsl_1 = 50, tsl_2 = 1, tsl_3 = 1, tsl_4 = 2$ and $tsl_5 = 46$. So, at the beginning of ts_1, the initial allocated shares of currently active tasks $T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8$ become: $sh_1^1 = sh_4^1 = sh_6^1 = sh_7^1 = 11, sh_2^1 = sh_3^1 = 13$ and $sh_5^1 = sh_8^1 = 10$, (see Equations (2) and (3)). Here, $sum\_shr^1 (= 90) < m \times tsl_l (= 2 \times 50 = 100)$ (see Equation (5)) and all active tasks are able to satisfy their required execution requirements. Therefore, there exists a feasible schedule within ts_1 (refer Algorithm 2) and these tasks are executed in ts_1, based on the generated schedule. At the beginning of the second time slice ts_2, i.e., at time $t = 50$, the currently active tasks in the system are T_2 and T_7. The final allocated shares of these tasks to be executed within ts_2 are given by: $sh_2^2 = sh_7^2 = 1$. At the beginning of time slice ts_3, the currently active tasks are T_2, T_5, T_7, and their allocated shares are 1, 1, 0, respectively. Here, *lag* based priority order of these three tasks is: $T_5 > T_2 > T_7$, and total number of available time slots is: $m \times tsl_3 = 2 \times 1 = 2$. Therefore, tasks T_2 and T_5 acquire a share of 1 time unit each, in time slice ts_3. In the fourth time slice ts_4, the currently active tasks are $T_1, T_2, T_4, T_5, T_7, T_8$, and their allocated shares are 1, 0, 1, 0, 1, 1, respectively. Here, *lag* based priority order of these six tasks is: $T_1 > T_4 > T_8 > T_7 > T_5 > T_2$, and total number of available time slots is 4. Therefore, tasks T_1, T_4, T_7, T_8 acquire a share of 1 time unit each, in time slice ts_4. The rest of the schedule continues as shown in Fig. 1.

4.2 Fault Model

As discussed, a fault occurring at F_{OT} remains undetected up to F_{DT} (refer Section 3: System Model). Hence, the *FT-FS* scheduler continues allocating tasks on all m processors between F_{OT} and F_{DT}, although tasks assigned to the faulty processor during this interval cannot progress. Subsequent to the detection of the fault, *FT-FS* updates its information regarding the partial progress of each running task at F_{DT}, taking into account the schedule of tasks allocated to the faulty processor during $[F_{OT}, F_{DT}]$.

At F_{DT}, the scheduler moves to the *fault mode* of operation and transits back to its *normal operational mode* at the end of the recovery period at F_{RT}. Due to the unavailability of one processor during the recovery period, the tasks may be forced to execute at lower rates than their required execution rates, potentially leading to *transient overloads*. In this context, the objective of *FT-FS* in *fault mode* is to maintain DP-Fairness (that is, ERfairness at all deadline boundaries) of the system during the recovery period even under transient overloads, allowing judicious task rejections, if need be.

4.3 FT-FS: Fault Mode of Operation

In order to recover from a fault, Algorithm *FT-FS* calls the function *FT-FS_Faulty()*. This function first determines the schedule of tasks for the entire recovery period, at time F_{DT}, and then, executes tasks in accordance with the generated schedule until the system recovers, at time F_{RT}. The pseudocode of *FT-FS_Faulty()* is presented in Algorithm 3. *FT-FS_Faulty()* proceeds in a time slice by time slice manner within the recovery interval. The first time slice starts at time F_{DT} while the last slice ends at F_{RT}. We use the term ts_l to denote the lth time slice within the recovery period. The term tsl_l is used to denote the length of ts_l.

Let R'_l denote the set of active tasks at the beginning of ensuing time slice ts_l (say, at time t) within the recovery period. *FT-FS_Faulty()* first determines tsl_l, which is given

2. *lag* [29] defines the fairness deviation for each task T_i at time t and is given as: $lag(T_i, t) = (e_i/p_i) \times t - allocated(T_i, t)$. Here, $allocated(T_i, t)$ is the amount of execution actually completed by T_i subsequent to its start at time 0.

by the earliest deadline among the deadlines of all currently active task instances. The *effective* execution rate of each task T_i in R'_l at time t, denoted as ewt_i^f is calculated using Equation (2), where $\mathcal{M} = m - 1$ (the number of available processors during the recovery period). The state of the system is considered to be *safe* at time t, if the actual effective rates of execution for all tasks are at least equal to their required execution rates. That is,

$$\forall T_i \in R'_l, ewt_i^f \geq \bar{e_i}/\bar{p_i}. \tag{9}$$

The system is considered to be in an *unsafe* state, otherwise. The tasks T_i for which the condition: $ewt_i^f < \bar{e_i}/\bar{p_i}$, is true, are referred to as *needy* tasks. Similarly, tasks T_i for which: $ewt_i^f > \bar{e_i}/\bar{p_i}$, is true, are referred to as *affluent*. Lastly, when: $ewt_i^f = \bar{e_i}/\bar{p_i}$, we refer the tasks to be in *balanced* condition. Inability to steer the system to *safety* from an *unsafe* state makes the system susceptible to a *failure event* in the future, in which *DP-Fairness constraints* for one or more *needy* tasks may be violated. Here, *DP-Fairness constraint* refers to the necessity to meet ERfairness only at time slice boundaries.

Algorithm 3. Function *FT-FS_Faulty()*

Output: Task schedule for recovery period: $[F_{DT}, F_{RT})$
1 Initialize: Current time $tt = F_{DT}$;
2 **while** $tt < F_{RT}$ **do**
3 Determine tsl_l, the length of the ensuing time slice;
4 Calculate required rates of execution $\bar{e_i}/\bar{p_i}$ and effective weights ewt_i^f, for all tasks in R'_l;
5 **if** *system is in unsafe state {refer Equation (9)}*
6 Partition task set R'_l into three disjoint subsets $A1, A2$ and $A3$ based on whether the tasks are *needy*, *affluent* or *balanced*, respectively;
7 $G^f = \sum_{i=1}^{|A1|} U_i$; $H^f = \sum_{j=1}^{|A2|} O_j$;
8 **if** $H^f < G^f$ **then**
9 Reject T_i, the least critical task instance with the highest U_i value, from $A1$;
10 $tt = max(s_i, F_{DT})$ {backtrack schedule};
11 Update R'_l back to its contents at time tt;
12 Update $\bar{e_i}, \bar{p_i}$ of all tasks in R'_l back to their values at tt;
13 Remove T_i from R'_l;
14 **else**
15 Call function *Weight_Donation()*;
16 **if** system is in safe state **then**
17 Call function *FT-FS_Normal()*;
18 $tt = tt + tsl_l$;
19 Execute schedule for the interval $[F_{DT}, F_{RT})$;

The algorithm checks for the safety of the system at the beginning of each time slice ts_l using Equations (2) and (9). If the state of the system is *safe*, the schedule generation proceeds as in the *normal* operational mode, and *FT-FS_Faulty()* calls *FT-FS_Normal()* in this case. In an *unsafe* situation, *FT-FS_Faulty()* performs a set of corrective actions in the endeavor to drive the system to *safety*. *FT-FS_Faulty()* first divides the set of tasks R'_l into three disjoint subsets $A1, A2$ and $A3$ based on whether the tasks are *needy*, *affluent* or *balanced* respectively, such that $R'_l = A1 \cup A2 \cup A3$. The *overallocation rate* corresponding to an *affluent* task T_j in $A2$ is given by:

$$O_j = ewt_j^f - \bar{e_j}/\bar{p_j}. \tag{10}$$

Similarly, the *underallocation rate* of a *needy* task T_i in $A1$ is denoted by:

$$U_i = \bar{e_i}/\bar{p_i} - ewt_i^f. \tag{11}$$

Now, if $O_j \geq U_i$, it is clear that both tasks T_i and T_j will be able to complete their execution requirements in a time slice, provided T_j is allowed to *donate* a portion U_i from its effective weight ewt_j^f, to T_i. After T_i *accepts* this donation, the effective execution rates of T_i and T_j become: $\bar{e_i}/\bar{p_i}$ and $ewt_j^f - U_i$, respectively.

As an illustration, let us consider two tasks T_1 (*affluent*) and T_2 (*needy*) with effective and required weights ($\{ewt_i^f, \bar{e_i}/\bar{p_i}\}$) $\{1/4, 1/5\}$ ($O_1 = 1/20$) and $\{1/10, 1/8\}$ ($U_2 = 1/40$), respectively. Given the parameters, in the absence of weight donation, T_1 and T_2 will complete 10 and 4 time slots of execution, respectively within a time slice ts_l of length $tsl_l = 40$. However, according to required rates, T_1 and T_2 need to complete 8 and 5 time slots of execution, respectively. Subsequent to weight donation, the effective weights of T_1 and T_2 become: $9/40$ and $1/8$, respectively. T_1 now completes ($9/40 \times 40 =$) 9 time slots, while T_2 is able to complete ($1/8 \times 40 =$) 5 time slots in ts_l, and so, both tasks are able to satisfy at least their required execution requirements.

Now, *FT-FS_Faulty()* calculates the aggregate (denoted by H^f) over the *overallocation rates* of tasks in $A2$: $H^f = \sum_{T_j \in A2} O_j$. Similarly, the aggregate *underallocation rate* G^f over U_i's is: $G^f = \sum_{T_i \in A1} U_i$. When $H^f \geq G^f$, *FT-FS_Faulty()* initiates weight donation (refer Algorithm 4), which allows each *needy* task T_i in $A1$ to receive the necessary boost to its effective weight so that it can execute at its required execution rate $\bar{e_i}/\bar{p_i}$ in time slice ts_l. For this, *FT-FS_Faulty()* selects the first *needy* task T_i and the first *affluent* task T_j from sets $A1$ and $A2$, respectively. If $O_j \geq U_i$, T_j donates a weight equivalent to U_i from its effective weight ewt_j^f to T_i. As a result of this donation, the effective execution rates of T_i and T_j become: $\bar{e_i}/\bar{p_i}$ and $ewt_j^f - U_i$, respectively. Since the requirement of T_i is satisfied, it is removed from $A1$ and added into the *balanced* set $A3$. In the special case, when $O_j = U_i$ T_j is also moved to $A3$. On the other hand, if $O_j < U_i$, T_j can only partially satisfy T_i's need. The effective execution rates of T_i and T_j subsequent to donation become: $ewt_i^f + O_j$ and $\bar{e_j}/\bar{p_j}$. T_j is moved to $A3$. This donation of weights continues until $A1$, the set of needy tasks, becomes empty.

If $H^f < G^f$, it means that the aggregate slack weight (H^f) of $A2$ is not sufficient to satisfy the total additional need (G^f) of $A1$, and so, even weight donation cannot guarantee DP-Fairness of all tasks. Therefore, one or more task instances (jobs) in $A1$ must be rejected so that the rest of the system remains *fail-operational*. At any scheduling point if $H^f < G^f$, *FT-FS_Faulty()* rejects the least critical task instance T_i having the highest underallocation value (U_i). This strategy attempts to ensure the rejection of the least number of high criticality jobs. Then, the algorithm backtracks the schedule generation back to the time $tt = max(s_i, F_{DT})$, where s_i is the arrival time of T_i. Now, *FT-FS_Faulty()* attempts to regenerate the schedule from time tt, this time, without task T_i in R'_l. Subsequent to a rejection, the actual effective execution rates of the

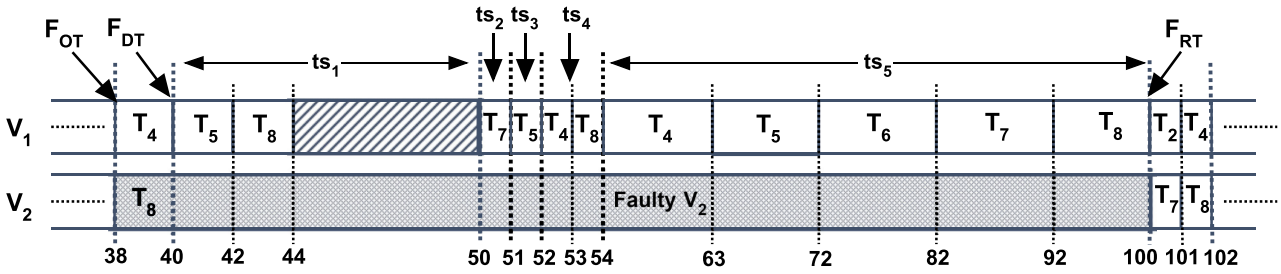

Fig. 2. The schedule generated by *FT-FS_Faulty()* during the interval $[40, 100]$.

tasks increase relatively due to reduced workload effected by the task rejection. This increases the possibility of the system becoming *safe* (checked using Equation (9)) or at least reduces the total additional requirement of *needy* tasks during the weight donation process. Further rejections may be required if the weight donation process fails in satisfying the requirement of all needy tasks. After the entire schedule for the interval $[F_{DT}, F_{RT}]$ is generated, *FT-FS_Faulty()* executes the schedule and returns back to the main function *FT-FS()* at time F_{RT}.

Algorithm 4. Function *Weight_Donation()*

Input: Needy set: $A1$, Affluent set: $A2$, Balanced set: $A3$
Output: Updated $A1, A2, A3$

```
1  while A1 ≠ ∅ do
2    Select first needy task T_i and first affluent task T_j from sets
     A1 and A2, respectively;
3    if O_j ≥ U_i then
4      ewt_j^f = ewt_j^f − U_i;  ewt_i^f ← ē_i/p̄_i;
5      O_j ← O_j − U_i; U_i ← 0;
6      A1 ← A1 \ T_i; A3 ← A3 ∪ T_i;
7      if O_j = 0 then
8        A2 ← A2 \ T_j; A3 ← A3 ∪ T_j;
9    else
10     ewt_j^f = ewt_j^f − O_j;  U_i ← U_i − O_j;  O_j ← 0;
11     A2 ← A2 \ T_j; A3 ← A3 ∪ T_j;
```

Example 2. Let us continue with the same system scenario discussed in Example 4, with the exception that processor V_2 now suffers a permanent fault at time $F_{OT} = 38$. The fault is detected at $F_{DT} = 40$ and the recovery interval t_r being 60, we get $F_{RT} = 100$. After detecting the fault at $F_{DT} = 40$, the main function *FT-FS* calls *FT-FS_Faulty()*, which proceeds in a time slice by time slice manner within the interval $[F_{DT}, F_{RT}]$. Therefore, the recovery duration consists of five slices $ts_1, \ldots, ts_5$ ($tsl_1 = 10, tsl_2 = tsl_3 = 1$, $tsl_4 = 2, tsl_5 = 46$) based on task deadlines. Fig. 2 depicts the schedule generated by *FT-FS_Faulty()* during the interval $[F_{DT}, F_{RT}]$.

At the beginning of time slice ts_1, the active task set is $R'_1 = \{T_4, T_5, T_8\}$. The effective and required weights ($\{ewt_i^f, \bar{e}_i/\bar{p}_i\}$) of these active tasks are {0.3526, 0.6666}, {0.3268, 0.1818} and {0.3205, 0.1666}, respectively. Here, $ewt_4^f < \bar{e}_4/\bar{p}_4$ (refer Equation (9)) and the system is in *unsafe* state. Partitioning R'_1 we get, $A1 = \{T_4\}$ (needy), $A2 = \{T_5, T_8\}$ (affluent) and $A3 = \emptyset$ (balanced). Since, $H^f (= 0.2989) < G^f (= 0.31405)$, the system cannot be driven to safety without incurring any rejection. Hence, task T_4 (only needy task in R'_1) is rejected and the system becomes safe subsequent to its removal. *FT-FS_Faulty()* therefore calls *FT-FS_Normal()* to generate the schedule for ts_1 as depicted in Fig. 2. Similarly, *FT-FS_Faulty()* generates the schedules for time slices ts_2 and ts_3.

At time $tt = 52$, the next instances of T_1, T_4 and T_8 arrive and the active task set R'_4 becomes $\{T_1, T_2, T_4, T_5, T_7, T_8\}$. Here, the system is in an *unsafe* state with $A1 = \{T_1, T_2, T_4, T_5, T_7, T_8\}$ and $A2 = A3 = \emptyset$. Since $H^f < G^f$, the least critical task T_1 in $A1$ is rejected and *FT-FS_Faulty()* reattempts to generate a feasible schedule (now, without T_1) for ts_4 (T_1 has arrived at the beginning of ts_4). However, schedule generation fails again and the system is still observed to be *unsafe* with $H^f < G^f$ ($A1 = \{T_2, T_4, T_5, T_7, T_8\}$ and $A2 = A3 = \emptyset$). *FT-FS_Faulty()* now rejects T_2, the least critical task, and attempts to regenerate a feasible schedule starting from the beginning of ts_2, the stipulated time for the arrival of T_2. After generating the schedules for ts_2 and ts_3 (now, without T_2) in a similar manner as before, a feasible schedule for ts_4 (without tasks T_1 and T_2) can be constructed with the task shares for T_4, T_5, T_7, T_8 being $sh_4^4 = 1, sh_5^4 = 0, sh_7^4 = 0, sh_8^4 = 1$. Proceeding further, at the beginning of ts_5 at time $tt = 54$, the system again becomes *unsafe* due to the arrival of the next instances of tasks T_3 and T_6 ($A1 = \{T_3, T_4, T_5, T_6, T_7, T_8\}$ and $A2 = A3 = \emptyset$). As $H^f < G^f$, the least critical task T_3 is rejected and the schedule generation for ts_5 is reattempted without T_3. This time, although the system is still unsafe, $H^f > G^f$ and so, system safety may be achieved through weight donation alone with no further task rejection being required. Subsequent to successful weight donations (refer Algorithm 4), the effective weights of the tasks T_4, T_5, T_6, T_7, T_8 in R'_5 are updated from 0.20665 to 0.2, 0.19155 to 0.19103, 0.1990 to 0.2037, 0.21492 to 0.21739, 0.18788 to 0.18788, respectively. At time $t = F_{RT} = 100$, the backup processor becomes operational and the system recovers from the fault.

Example 3. Let us continue with the same system scenario discussed in Example 4, with the exception that system has three unit capacity processors and processor V_2 suffers a permanent fault at time $F_{OT} = 38$. The fault is detected at $F_{DT} = 40$ and the recovery interval t_r being 60, we get $F_{RT} = 100$. Fig. 3depicts the *FT-FS* schedule for this system in the fault mode of operation. It may be observed from Fig. 3 that even in the presence of the faulty processor (V_2), the system incurs no task rejections. This is because FT-FS effectively schedules the given workload on the remaining two nonfaulty processors.

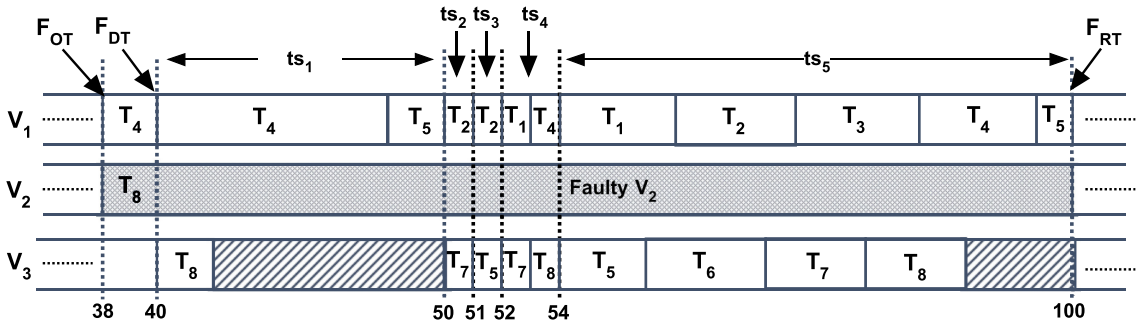

Fig. 3. The schedule generated for a 3 processor system under fault mode of operation.

4.4 Complexity Analysis

In this section, we present the time complexity analysis of the proposed FT-FS scheduler, with the objective of accounting the scheduling overheads associated with the algorithm.

Lemma 1. *Complexity of function* FT-FS_Normal() *(Algorithm 2) is* $O(nlogn)$ *per time slice.*

Proof. The loops in lines 1-3, 6-8 and 12-17 run once for each task with each step inside the loop taking constant time. So, the complexity of these loops become $O(n)$. Steps 4 and 9 which involves updation of the spare capacity $scap$ and takes $O(n)$ time. Step 11 which creates a modified task list, consumes $O(nlogn)$ time. Step 18 involves generation of a schedule within a time slice and takes $O(n)$ time. Therefore, function *FT-FS_Normal()* has an overall complexity of $O(nlogn)$ per time slice. □

Theorem 1. *Amortized complexity of function* FT-FS_Normal() *(Algorithm 2) is* $O(1)$ *per processor per time slot.*

Proof. From Lemma 1, the worst case time complexity of function *FT-FS_Normal()* is obtained as $O(nlogn)$ per time slice. The length of a time slice may be considered to be roughly proportional to the average task period length. Further typically, it may be realistically assumed that: *#tasks* $n << $ *average period size* $\times$ *#processors* m. Hence, *#tasks* $n << $ *time slice length* $\times$ *#processors* m. So, the amortized complexity of *FT-FS_Normal()* may be considered to be $O(1)$ per processor per time slot. □

Lemma 2. *Complexity of function* Weight_Donation() *(Algorithm 4) is* $O(n)$.

Proof. Algorithm 4 performs weight donation from affluent tasks in set $A2$ to needy tasks in $A1$, until the number of needy tasks in $A1$ reduces to zero. The maximum number of iterations of the while loop (Steps 1 to 11) is upper bounded by the initial cardinality of $A1$ which is $O(n)$. Therefore, this loop takes $O(n)$ time. Steps 6, 8 and 11 involve insertion and deletion operations at the beginning of sets $A1, A2, A3$ and each such operation takes constant time. The other steps in the while loop also consume $O(1)$ time. Therefore, the worst case time complexity of *Weight_Donation()* is $O(n)$. □

Lemma 3. *Average complexity of function* FT-FS_Faulty() *(Algorithm 3) is* $O(nlogn)$.

Proof. Function *FT-FS_Faulty()* is used to generate a schedule for the recovery duration $[F_{DT}, F_{RT})$. Step 3 involves determination of the length of the ensuing time slice, at each time slice boundary within $[F_{DT}, F_{RT})$. This length is given by the earliest among the deadlines of all currently active instances and takes $O(n)$ time for its computation. Each of the Steps 4 to 7 involves operations over sets of tasks where the cardinality of each of these sets are upper bounded by the total number of tasks, n. So, each of these steps takes $O(n)$ time. Step 9 which involves finding the least critical *needy* task with the highest utilization value, say T_i (from set $A1$), and consumes $O(n)$ time to complete. Step 10 updates tt to determine the number of time slices by which the schedule should be backtracked and this takes $O(1)$ time. Step 11 involves updation of the active task set R'_l, with respect to the backtracked schedule, and this can be done in $O(n)$ time. Step 12, which updates the $\bar{e}_i$ and $\bar{p}_i$ values of all tasks in the modified set R'_l, takes $O(n)$ time. Step 13 involves deletion of task T_i from R'_l and incurs an overhead of $O(n)$. Steps 15 and 17 call functions *Weight_Donation()* (Lemma 2) and *FT-FS_Normal()* (Lemma 1), respectively. These steps take $O(n)$ and $O(nlogn)$ times, respectively. Step 18 increments tt and takes $O(1)$ time. Therefore, the complexity of executing the single iteration of the *while* loop (steps 2 to 18) is $O(nlogn)$ in the worst case. The number of iterations of the while loop is determined by the number of time slices for which the schedule needs to be generated. This number must include backtracked re-executions of certain time slices (the time slice intervals during re-execution may possibly be different due to task rejections). Let us represent k to be the number of time slices generated by the active tasks (excluding the rejected tasks) within the recovery interval and k' to be the total number of time slices by which the schedule was backtracked due to all task rejections. Now given k and k', the complexity of *FT-FS_Faulty()* may be obtained as $O((k+k')nlogn)$. In this work, we have assumed a closely-coupled system in which the cold-standby processor may be activated within a short recovery duration (varying between say, tens of milliseconds to at most a few seconds) subsequent to the detection of a fault. Further, as time slices are demarcated by task period boundaries, average length of a time slice can be considered to be approximately equal to the mean period length of the given task-set. We have also observed that for a large class of real-time systems including automotive and avionic systems, most process control systems, satellites etc., task period lengths typically vary between tens to hundreds of milliseconds, or even more. Therefore, the number of time slices k, within the recovery duration may be reasonably considered to be bounded by a small constant.

Average number of time slices by which the schedule needs to be backtracked (say, k'') at each task rejection is upper bounded by k, and so, k'' may also be considered to assume small values. Moreover, the algorithm attempts to choose tasks with high utilization values for rejection. Due to this, transient overloads during recovery are typically mitigated by only suffering a small number of rejections (say, n_r), in most cases. Based on these arguments, k' ($=k'' \times n_r$) can also be assumed to be upper bounded by a small constant. Hence, assuming both k and k' to be small constants, the average complexity of *FT-FS_Faulty()* becomes $O(nlogn)$. □

Theorem 2. *Amortized complexity of function* FT-FS_Faulty() *(Algorithm 3) is* $O(1)$ *per processor per time slot.*

Proof. At the beginning of the recovery interval, *FT-FS_Faulty()* takes $O(nlogn)$ time to generate a schedule for the entire recovery period (Lemma 3). Therefore, the amortized complexity of *FT-FS_Faulty()* may be obtained by determining the average overhead incurred by the function per processor per time slice over the recovery interval $[F_{DT}, F_{RT})$. In our work, we have represented $[F_{DT}, F_{RT})$ as an integral number (say, k) of time slices with the duration of a typical time slice being assumed to be equal to average task period length. Therefore, amortized complexity of *FT-FS_Faulty()* becomes, $O(nlogn/(\ k \times$ *average period size* $\times$ *#processors* $m))$. Finally, with the realistic assumption that: *#tasks* $n \times logn \leq k \times$ *average period size* $\times$ *#processors* m, amortized complexity of *FT-FS_Faulty()* may be considered to be $O(1)$ per processor per time slot. □

Example 4. Let us consider a typical fully loaded system consisting of 8 processors with 40 tasks having an average execution requirement of 50 ms. With average task utilization being 1/5 (=8/40), the average task period length becomes 250 ms. Thus in this case, *#tasks* (40) $<<$ *average period length* (250) $\times$ #processors (8). These numbers thus validate the amortized time complexity results for *FT-FS_Normal()* and *FT-FS_Faulty()* obtained in Theorems 1 and 2, respectively.

Theorem 3. *Amortized complexity of the FT-FS Scheduler (Algorithm 1) is* $O(1)$ *per processor per time slot.*

Proof. It follows from Theorems 1 and 2. □

The complexity analysis conducted above shows that FT-FS is a low-overhead algorithm which incurs $O(1)$ average amortized overhead. This theoretical analysis has also being supported by our obtained experimental results as discussed in Section 5.4. Before presenting the detailed results, we now present the experimental framework used in this work.

5 Experiments and Results

We have evaluated the performance of the *FT-FS* algorithm through an extensive set of simulation studies (conducted over an experimental framework which is described in the next subsection) and compared its performance against a basic fault-tolerant fair scheduling strategy called *Basic-FS*.

Basic-FS. In its normal mode of operation, *Basic-FS* is same as *FT-FS*. However, it follows a naive and much simpler fault recovery policy. To drive the system to safety during transient overloads in the recovery period subsequent to a fault, *Basic-FS* repeatedly rejects the least critical tasks from set $A1$ until the overload is mitigated. Therefore, neither does this strategy employ *weight donation* nor *backtracking* subsequent to a task rejection.

5.1 Experimental Setup

The experimental setup in this work consists of a scenario generation framework which provides input test datasets corresponding to different scenarios over which both the *Basic-FS* and the *FT-FS* algorithms are evaluated. Each result data point is the average obtained by running the algorithms over 100 different datasets corresponding to a given scenario. The schedule length in all simulations have been taken to be 100000 time slots with the length of a time slot being assumed as 1 millisecond. We now discuss the scenario generation framework in more detail.

Given the number of tasks to be generated (n) and the summation of weights of the n tasks (L), the individual task weights (wt_i) have been generated from a normal distribution with standard deviation $(\sigma) = 0.1$ and mean $(\mu) = L/n$. The summation of weights of these generated tasks is not constant. However, making the summation of weights constant helps in the comparison of the algorithms. Therefore, the weights have been scaled uniformly to make the cumulative weight of each task set constant and equal to L. Task execution periods (p_i) have been generated from another normal distribution with $\mu = 400$ and $\sigma = 40$. Each task T_i in the system is assigned a criticality level cr_i which is denoted by an integer generated randomly from a uniform distribution with in the range $[1, n]$. The framework also includes a fault injection mechanism which randomly generates fault occurrence instants using Poisson distribution with fault rate $\lambda = 1 \times 10^{-5}$ [27]. The mechanism take cares that two consecutive fault occurrence instants are separated by at least: *recovery interval* t_r + *periodic safety check interval* t_p. The value of t_p has been assumed to be 10 milliseconds in all experiments.

Now, various simulation scenarios have been generated by setting different values for the following parameters:

1) Number of processors m: This parameter is varied between 2 to 10.
2) Task set size n: The number of tasks in the system have been varied from 20 to 100.
3) Workload: Our experiments are conducted on 70, 75, 80, 85, 90, 95 or 100 percent loaded systems.
4) Recovery time t_r: In order to study the effect of t_r, the recovery interval has been varied from 50 to 200 (in milliseconds).

5.2 Performance Evaluation Parameters

Fault of a processor may lead to transient overloads in the system, which in turn may necessitate the rejection of one or more jobs in order to keep the system fail-operational. The performance of the proposed framework has been evaluated using four different metrics:

1) *#Job_Rejections*: This denotes the average number of jobs rejected over 100 runs (with distinct datasets) by the algorithms *FT-FS* and *Basic-FS* corresponding to a given scenario.

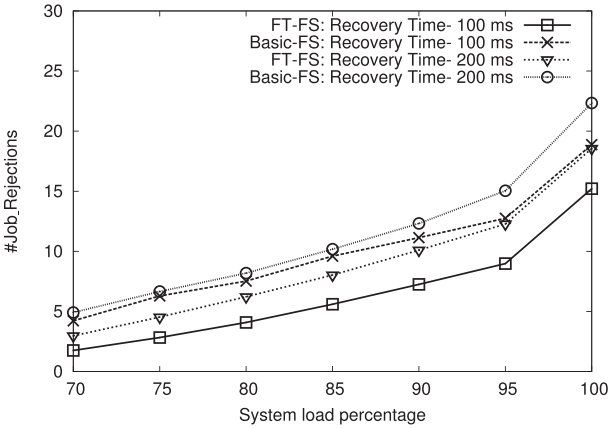

Fig. 4. #Job_Rejections versus System load: 2 processors, 40 tasks.

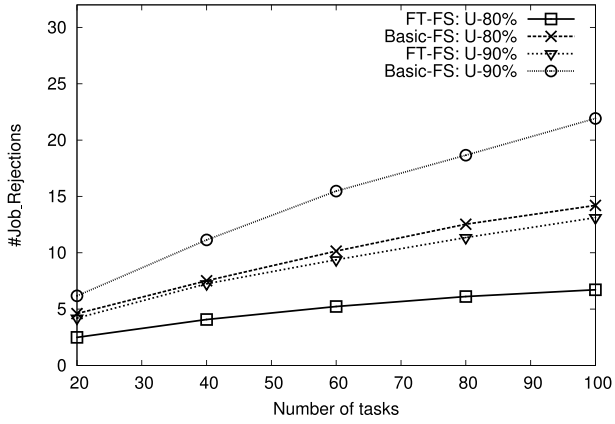

Fig. 6. #Job_Rejections versus #Tasks: 2 processors, t_r = 100 ms.

2) *#Penalty*: This parameter represents the average (over 100 runs) aggregate penalty suffered by the system due to job rejections. Here, penalty (corresponding to the single run of the experiment) is calculated by the summation of the criticality values of rejected jobs over the schedule length.
3) *Normalized context switch overhead*: It is the average preemption/migration overhead (in μs) per processor per time slot, incurred by FT-FS.
4) *Normalized scheduling overhead*: It is the average overhead (in μs) per time slot, incurred by FT-FS towards making scheduling decisions.

5.3 Results: Performance

Figs. 4, 5, 6 show the comparison of *FT-FS* and *Basic-FS* based on the performance parameter #Job_Rejection. The plots shown in Fig. 4 are obtained for two distinct values of recovery period t_r (100 and 200) and varying system workloads (U) corresponding to systems consisting of two processors and 40 tasks. It may be observed from Fig. 4 that for all cases, as expected, number of job rejections increases as system workload becomes higher due to progressive reduction in slack capacity within the system. Also, rejection rates are seen to increase very steeply when the system becomes almost fully loaded (for, $U > 95\%$). In addition, as recovery interval represents the time for which the system must deal with a sub-nominal capacity, plots for which $t_r = 200$ exhibit higher rejections compared to plots with $t_r = 100$. Finally, the plots clearly show the superiority of the proposed *FT-FS* algorithm over *Basic-FS* in terms of the ability to control job rejections. Empowered with the weight donation and post rejection backtracking mechanisms embedded with *FT-FS*, it incurs significantly lower rejections with respect to *Basic-FS*.

Fig. 5 considers scenarios containing 40 tasks, 80 or 90 percent loaded systems with varying number of processors. As is obvious, for any given number of processors, 90 percent loaded systems suffer higher rejections compared to systems where load $U = 80\%$. It may also be observed that job rejections decrease with increase in the number of processors when the workload, recovery time and number of tasks remain unchanged. This may be attributed to the fact that fractional loss in system capacity, during recovery, due to the failure of a single processor, decreases with increase in the available number of processors. Also as discussed above, *FT-FS* being more efficient, is seen to always perform better than *Basic-FS*.

Fig. 6 shows the variation in #Job_Rejections as the number of tasks is varied in a two processor, 80 or 90 percent loaded system with 100 ms as the recovery interval. Here, we observed that #Job_Rejections increases with growth in the number of tasks. This is because, as the number of tasks increases in a scenario with fixed total workload (U = 80% or 90%), the number of jobs which suffer under-allocation at time slice boundaries (leading to rejections) during recovery, also increases. However, a closer observation shows that although #Job_Rejections increases, the rejection ratio (#Job_Rejections : #Tasks) decreases with larger number of tasks. This is due to the fact that individual task weights decrease as tasks increase in scenarios with a fixed system load and such smaller weights have higher probability of fitting into a given available slack capacity in the system. As for the other two figures, *FT-FS* performs consistently better than *Basic-FS*.

Fig. 7 compares *FT-FS* and *Basic-FS* against the performance parameter #Penalty. As expected, *FT-FS* being equipped with weight donation and backtracking capabilities incurs far low penalties compared to *Basic-FS*. Table 2 summarizes the comparison results obtained for *FT-FS* and *Basic-FS* on two processor systems.

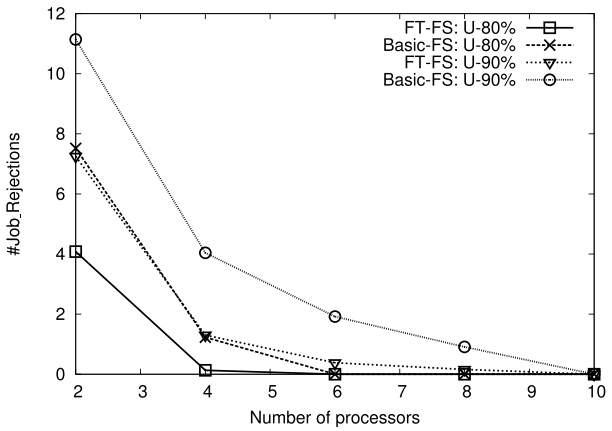

Fig. 5. #Job_Rejections versus #Processors: 40 tasks, t_r = 100 ms.

5.4 Results: Overheads

Fig. 8 shows plots for the normalized context switch overheads (refer Section 5.2) incurred by FT-FS, as the system load

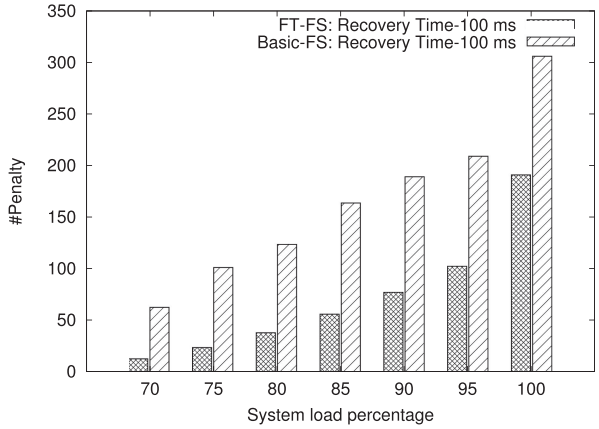

Fig. 7. #Penalty versus System load: 2 processors, 40 tasks, t_r = 100 ms.

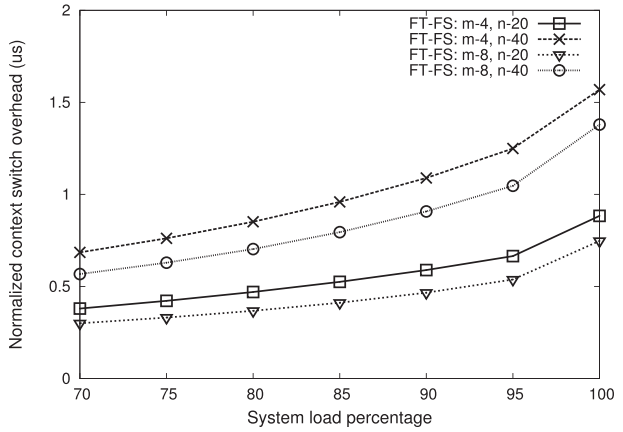

Fig. 8. Normalized context switch overheads of FT-FS.

TABLE 2
FT-FS versus *Basic-FS*: Average Number of Jobs Rejected

U	t_r	n = 20		n = 40		n = 60	
		FT-FS	Basic-FS	FT-FS	Basic-FS	FT-FS	Basic-FS
75%	50	1.53	3.36	2.3	5.96	2.83	7.57
	100	1.83	3.5	2.83	6.29	3.54	7.87
	150	2.4	3.76	3.77	6.37	4.66	8.43
	200	2.84	4.06	4.53	6.66	5.47	8.78
85%	50	2.99	4.97	4.97	8.65	6.38	11.85
	100	3.34	5.42	5.6	9.6	7.18	12.8
	150	4.1	5.72	6.88	9.76	8.97	13.07
	200	4.73	6.05	8.03	10.18	10.55	14.03
95%	50	4.88	6.97	8.43	12.35	11.28	17.01
	100	5.23	7.38	8.98	12.75	11.98	18.22
	150	6.1	8.21	10.65	14.12	14.29	19.72
	200	7.1	8.76	12.28	15.04	16.83	21.13

n: Total number of tasks; U: Total system load percentage; t_r: Recovery time #processors: 2; t_p: 10 ms; Average period length of tasks, P_{avg}: 400 ms

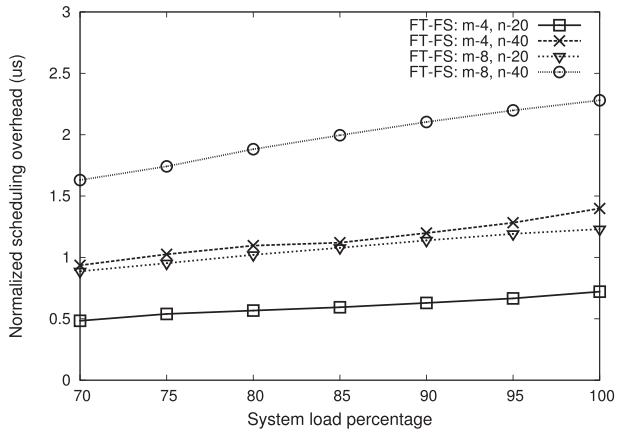

Fig. 9. Normalized scheduling overheads of *FT-FS_Normal()*.

is varied on four and eight processor systems. In this experiment, we assume the delay corresponding to a single context switch to be 5.24 μs, which is the actual average context switch overhead on a 24-core Intel Xeon L7455 system under typical workloads [30]. The normalized context switch overhead (in μs) is determined as follows: we first counted the average number of context switches per processor per time slot for a given simulation run and then multiplied it with the cost of a single context switch (5.24 μs). It may be observed that for a given task set (n = 20 or 40) and number of processors (m = 4 or 8), the overhead increases as system load increases from 70 to 100 percent. This may be attributed to the fact that as system load increases, individual task weights also increase and such larger weights increase the execution times of tasks. As a result, individual task shares within time slices become larger and residual spare capacities in the system reduce. Consequently, a higher number of context switches must be incurred to feasibly accommodate and execute the tasks on the available processors. It may also be observed that, for a given workload, overhead increases with increase in the number of tasks. This is because, the number of time slices within the schedule increases proportionately with increase in the number of tasks. Consequently, the number of migrations across time slices also increases proportionately. Additionally, the number of preemptions within the time slice increases as the number of tasks increases. It may also be observed that for a given task set (n = 20 or 40) and workload, the normalized context switch overhead decreases with an increase in the number of processors. This is due to the fact that the spare capacity in the system increases as the number of processors become higher with the system workload remaining the same. Due to such additional spare capacity, the tasks are able to execute continuously on the same processor for longer durations on average, without incurring migrations/preemptions. Finally, it may be observed from Fig. 8 that the maximum normalized context switch overheads for the considered scenarios is about 1.56 μs per processor per time slot (for four processor, 40 tasks, fully loaded systems). Therefore, considering 1 millisecond as the time slot length, about 0.156 percent of a slot duration may be considered to be wasted due to context switch related overheads.

Fig. 9 shows plots for the normalized scheduling overheads (refer Section 5.2) incurred by *FT-FS_Normal()*, as the system load is varied on four and eight processor systems. In this experiment, the normalized scheduling overhead is determined by first finding out the average scheduling overhead for the entire schedule length incurred by *FT-FS_Normal()* over 100 simulation runs. This average scheduling overhead is then divided by the length of schedule to obtain the normalized overhead per time slot. As expected,

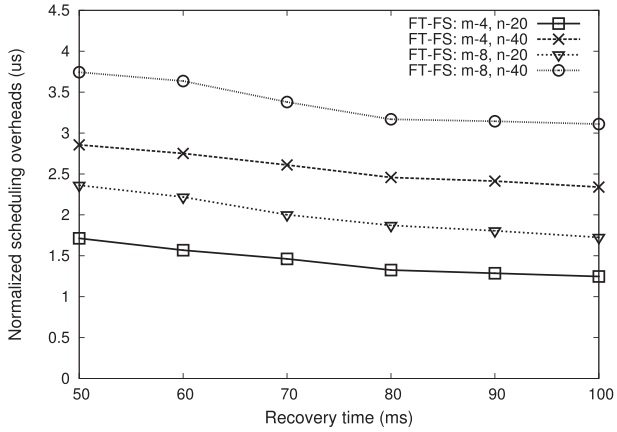

Fig. 10. Normalized scheduling overheads of *FT-FS_Faulty()*.

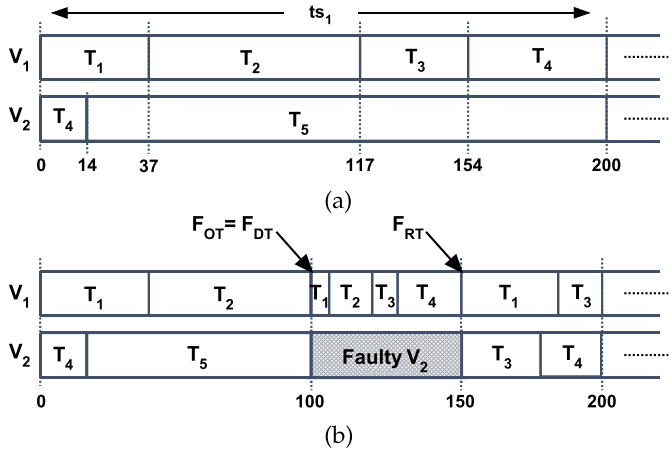

Fig. 11. Aircraft Flight Control: (a) & (b) Schedules generated by FT-FS under both nominal and fault modes of operation, respectively, for the first 200 time slots.

the overhead increases when both the number of tasks and system load increases. Moreover, for a given task set (n = 20 or 40) and workload, normalized scheduling overhead increases with an increase in the number of processors. Finally, it may be observed from Fig. 9 that the maximum normalized scheduling overheads for the considered scenarios is about 2.27 μs per time slot (for eight processor, 40 tasks, fully loaded systems). This normalized overhead is experienced by tasks on all processors at any time slot. Therefore, considering 1 millisecond as the time slot length, about 0.227 percent of a slot duration may be considered to be wasted due to scheduling related overheads.

Fig. 10 shows plots for the normalized scheduling overheads incurred by *FT-FS_Faulty()*, as the recovery time is varied on four and eight processor systems. In this experiment, the normalized scheduling overhead is determined by first finding out the average scheduling overhead incurred by *FT-FS_Faulty()* at the beginning of the recovery interval and then dividing it by the recovery duration. It may be noted that during the recovery period, the numbers of available processors in four and eight processor systems become three and seven, respectively. In addition, these systems are assumed to be hit by a transient overload of 5 percent, subsequent to the failure of a processor. So, the total system utilization has been fixed at 78.75 percent for four processor systems and 91.875 percent for eight processor systems, such that the workloads rise to 105 percent in both these systems, during the recovery period. As expected, for a given processor (m = 4 or 8), the normalized scheduling overhead increases as the number of tasks become higher. Moreover, for a given task set (n = 20 or 40), normalized scheduling overhead increases with an increase in the number of processors. It may also be observed that for a given task set (n = 20 or 40), processor (m = 4 or 8) and transient overload, normalized scheduling overhead decreases with an increase in the recovery time. This is due to the fact that the proportional increase in average scheduling overhead decreases as recovery time increases. It may be observed from Fig. 10 that the maximum normalized scheduling overheads for the considered scenarios is about 3.74 μs per time slot (for eight processor, 40 tasks systems with recovery time, t_r = 50 ms). This normalized overhead is experienced by tasks on all available processors at any time slot during the recovery period. Finally, it may be noted that the overall normalized scheduling overhead of the FT-FS scheduler may be obtained as the maximum of the overheads incurred by *FT-FS_Faulty()* and *FT-FS_Normal()*. Finally, the total overhead of FT-FS per time slot is given by the sum of normalized context switch and scheduling overheads. For example, for an eight processor, 40 task system, with t_r = 50 ms, the normalized scheduling overheads of *FT-FS_Faulty()* and *FT-FS_Normal()* are 2.27 μs (refer Fig. 9) and 3.74 μs (refer Fig. 10), respectively. Additionally, the normalized context switch overheads of FT-FS corresponding to these system parameters is 1.379 μs (refer Fig. 8). From these values, the total normalized overhead of FT-FS may be obtained as 1.379 μs + max(2.27, 3.74) μs = 5.119 μs. Therefore, considering 1 millisecond as the time slot length, only about 0.5119 percent of a slot duration may be considered to be wasted due to overheads. This overhead can be easily incorporated within the schedule by inflating the execution requirement of each task by 0.5119 percent.

6 Case Study: Aircraft Flight Control

In this section, we present a case study using an automated flight control system to illustrate the generic applicability of our fault recovery mechanism in real world scenarios. The Flight Management System (FMS) in an aircraft performs several flight control functions, including navigation, guidance, control, etc [31]. FMS requires four separate tasks to control the aircraft during flight: Guidance, Controller, Slow Navigation and Fast Navigation. The *Guidance* task (say T_1) sets the reference trajectory of the aircraft in terms of altitude and compass heading. Based on the reference trajectory and navigation sensor values, the *Controller* task (say T_2) executes closed loop control functions that compute actuation commands for components including, elevator, ailerons, rudder and throttle, to achieve a desired reference altitude and heading for the aircraft. The elevator, ailerons and rudder generate aerodynamic forces that alter aircraft heading and airspeed. The engine throttle generates a force along the aircraft fuselage which is used in combination with the aerodynamic forces to alter aircraft airspeed and altitude. The job of both *Slow* (say T_3) and *Fast* (say T_4) *Navigation* tasks is to read sensors at low and high sampling

frequencies, respectively. While slow navigation task is used to feed data to the less critical Guidance task, fast navigation task feeds data to the high critical Controller task. Now, we consider the case of an F-16 fighter aircraft which performs an additional function, *launch a missile at the enemy target during military operation*. Therefore, in addition to the basic flight control tasks (Guidance, Controller, Slow Navigation, Fast Navigation), the fighter aircraft requires a *Missile Control* task (say T_5) to monitor the aircraft radar, detect enemy targets and fire a missile if a target is detected.

Now, consider these five tasks T_1 (100, 1000, 2), T_2 (80, 200, 3), T_3 (100, 1000, 2), T_4 (60, 200, 3), T_5 (500, 1000, 1) to be executed on two unit capacity processors ($m = 2$) using the FT-FS scheduling scheme. The tasks and their associated parameters used for this case study have been taken from [31]. Each of these tasks may be assigned a relative criticality value based on the importance of their usage in flight control performance and military mission operation. In this case study, we assume that flight control tasks have higher relative criticality values compared to the military mission task. Fig. 11a depicts the FT-FS schedule for this system for the first 200 time slots (time slice ts_1).

All these tasks are real-time and periodic in nature, whose timing constraints have to be satisfied even in the presence of faults. Let us consider the following faulty scenario: t_p = 10 ms, t_r = 50 ms, $F_{OT} = F_{DT}$ = 100 and F_{RT} = 150. Due to the failure of any one of the two processors (say, processor V_2 fails), tasks have to execute on the remaining functional processor (V_1) until the system recovers. Fig. 11b depicts the schedule generated by FT-FS under fault mode of operation. At time t = 100, the system is observed to be *unsafe* and FT-FS attempts to regenerate a feasible schedule by rejecting the least critical task T_5. After this rejection, it becomes possible to generate a feasible schedule through weight donation alone with no further task rejection being required during the recovery interval [100, 150] (thereby, displaying the ability of the algorithm to remain fail-operational as far as possible). It may also be observed that under both nominal and fault modes of operation FT-FS is able to deliver optimal resource utilization. Fig. 11a shows that FT-FS incurs only 1 task migration and 4 preemptions, thus being able to effectively control context switching related scheduling overheads.

7 Conclusion

This work presents a fault-tolerant proportional fair scheduling mechanism called *FT-FS*, for periodic real-time multiprocessor systems containing cold-standby spares. Subsequent to the detection of a permanent processor fault, the system requires a fixed recovery interval to boot up the spare processor to the operational state. Equipped with two novel features namely, *weight donation* and *post rejection backtracking*, the proposed scheduler *FT-FS* attempts to minimize rejections of critical jobs, during transient overloads within recovery intervals. The objective is to maximize the possibility of keeping the system fail-operational even in the presence of faults. The underlying scheduling structure being based on DP-Fair, *FT-FS* is able to ensure high resource utilization and fair rate-based execution progress while incurring low scheduling related overheads through controlled migrations and context-switches. Experimental results reveal that the *FT-FS* algorithm performs appreciably over an extensive sets of system scenarios pointing to the practical effectiveness of the scheme.

In this paper, we have considered a system with identical processors. As part of our future work, we plan to refine this work to incorporate heterogeneous processing platforms, where the same task may need different amounts of execution time on different processors. Moreover, we plan to extend our framework to handle parallel applications with precedence constraints. In an underloaded system, the proposed work allows a processor to be idle in the absence of runnable/ready tasks, and this causes a considerable amount of energy wastage in the system. Therefore, we also aim to develop efficient energy-aware scheduling strategies for this fault-tolerant framework.

References

[1] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, vol. 24. New York, NY, USA: Springer Science & Business Media, 2011.

[2] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surveys*, vol. 43, no. 4, 2011, Art. no. 35.

[3] B. Andersson and J. Jonsson, "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%," in *Proc. 15th Euromicro Conf. Real-Time Syst.*, 2003, pp. 33–40.

[4] S. K. Baruah, et al., "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[5] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proc. 12th Eur. Conf. Real-Time Syst.*, 2000, pp. 35–43.

[6] G. Levin, et al., "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *Proc. 22nd Euromicro Conf. Real-Time Syst.*, 2010, pp. 3–13.

[7] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[8] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. New York, NY, USA: Elsevier, 2010.

[9] E. Dubrova, *Fault-Tolerant Design*. New York, NY, USA: Springer, 2013.

[10] C. Krishna, "Fault-tolerant scheduling in homogeneous real-time systems," *ACM Comput. Surveys*, vol. 46, no. 4, 2014, Art. no. 48.

[11] R. Samet, "Recovery device for real-time dual-redundant computer systems," *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 3, pp. 391–403, May 2011.

[12] G. Manimaran and C. S. R. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 11, pp. 1137–1152, Nov. 1998.

[13] S. H. Kang, H. w. Park, S. Kim, H. Oh, and S. Ha, "Optimal checkpoint selection with dual-modular redundancy hardening," *IEEE Trans. Comput.*, vol. 64, no. 7, pp. 2036–2048, Jul. 2015.

[14] N. El-Sayed and B. Schroeder, "Understanding practical tradeoffs in hpc checkpoint-scheduling policies," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 2, Mar./Apr. 2016.

[15] K. Han, G. Lee, and K. Choi, "Software-level approaches for tolerating transient faults in a coarse-grainedreconfigurable architecture," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 4, pp. 392–398, Jul./Aug. 2014.

[16] G. Levitin, L. Xing, H. Ben-Haim, and Y. Dai, "Effect of failure propagation on cold vs. hot standby tradeoff in heterogeneous 1-out-of-n: G systems," *IEEE Trans. Rel.*, vol. 64, no. 1, pp. 410–419, Mar. 2015.

[17] L. Xing, O. Tannous, and J. B. Dugan, "Reliability analysis of nonrepairable cold-standby systems using sequential binary decision diagrams," *IEEE Trans. Syst. Man Cybern.-Part A: Syst. Humans*, vol. 42, no. 3, pp. 715–726, May 2012.

[18] P. Mejía-Alvarez and D. Mossé, "A responsiveness approach for scheduling fault recovery in real-time systems," in *Proc. 5th IEEE Real-Time Technol. Appl. Symp*, 1999, pp. 4–13.

[19] R. M. Pathan, *Scheduling Algorithms for Fault-Tolerant Real-Time Systems*, *PhD Diss.*, Department of Computer Science and Engineering Chalmers University of Technology Göteborg, Sweden, 2010.
[20] R. Isermann, R. Schwarz, and S. Stolzl, "Fault-tolerant drive-by-wire systems," *IEEE Control Syst.*, vol. 22, no. 5, pp. 64–81, Oct. 2002.
[21] M. Blanke, M. Staroswiecki, and N. E. Wu, "Concepts and methods in fault-tolerant control," in *Proc. Amer. Control Conf.*, 2001, vol. 4, pp. 2606–2620.
[22] R. Al-Omari, A. K. Somani, and G. Manimaran, "An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 595–608, 2005.
[23] J. Kim, K. Lakshmanan, and R. Rajkumar, "R-batch: Task partitioning for fault-tolerant multiprocessor real-time systems," in *Proc. IEEE 10th Int. Conf. Comput. Inf. Technol.*, 2010, pp. 1872–1879.
[24] R. M. Pathan and J. Jonsson, "Ftgs: Fault-tolerant fixed-priority scheduling on multiprocessors," in *Proc. IEEE 10th Int. Conf. Trust Secur. Privacy Comput. Commun.*, 2011, pp. 1164–1175.
[25] R. M. Pathan, "Real-time scheduling algorithm for safety-critical systems on faulty multicore environments," *Real-Time Syst.*, vol. 53, no. 1, pp. 45–81, 2017.
[26] A. Bhat, S. Samii, and R. R. Rajkumar, "Recovery time considerations in real-time systems employing software fault tolerance," in *Proc. LIPIcs-Leibniz Int. Proc. Informat.*, 2018, vol. 106, pp. 23:1–23:22.
[27] M. H. Mottaghi and H. R. Zarandi, "Dfts: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors," *Microprocessors Microsystems*, vol. 38, no. 1, pp. 88–97, 2014.
[28] R. McNaughton, "Scheduling with deadlines and loss functions," *Manage. Sci.*, vol. 6, no. 1, pp. 1–12, 1959.
[29] J. H. Anderson and A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks," *J. Comput. Syst. Sci.*, vol. 68, no. 1, pp. 157–204, 2004.
[30] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proc. OSPERT*, pp. 33–44, 2010.
[31] T. Atdelzater, E. M. Atkins, and K. G. Shin, "QoS negotiation in real-time systems and its application to automated flight control," *IEEE Trans. Comput.*, vol. 49, no. 11, pp. 1170–1183, Nov. 2000.

Piyoosh Purushothaman Nair received the BTech degree in information technology from the University of Kerala, in 2010 and the MTech degree in computer science and engineering from the Indian Institute of Technology Guwahati, in 2014. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati. His research interests include real-time systems, fault-tolerance, discrete-event systems and embedded systems. He is partially supported through TCS research fellowship program.

Arnab Sarkar received the BTech degree in information technology from the University of Calcutta, Kolkata, India, in 2003, and the MS and PhD degrees from the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Kharagpur, India, in 2006 and 2012, respectively. After submitting the PhD thesis in 2011, he worked briefly as a visiting scientist with the Advanced Computing and Microelectronics Unit (ACMU), Indian Statistical Institute (ISI), Kolkata, India, before joining Samsung India Software Operations(SISO), Bangalore, where he worked for one year as a chief engineer with the Android Platforms Group. He is currently working as an associate professor with the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Guwahati, India. His current research interests include scheduling strategies in real-time and embedded systems and bandwidth allocation in wireless cellular networks. He is a member of the IEEE.

Santosh Biswas received the BE degree in computer science and engineering from NIT Durgapur, in 2001, the MS degree in electrical engineering and the PhD degree in computer science and engineering from the IIT Kharagpur, in 2004 and 2008, respectively. He is currently an associate professor with the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati. He has been involved in several Research Projects sponsored by Industry and Government agencies. He is engaged with academic as well as industry sponsored research related to VLSI Testing and Design for Testability. His research interests include VLSI testing, fault tolerance, network security, discrete-event systems and embedded systems. He has published about 170 research papers. He is a member of the IEEE.