Real-Time Scheduling under Fault Bursts with Multiple Recovery Strategy

Mohammad A Haque, Hakan Aydin
Department of Computer Science
George Mason University
Fairfax, Virginia 22030
e-mail: mhaque4@gmu.edu, aydin@cs.gmu.edu

Dakai Zhu
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas 78249
e-mail: dzhu@cs.utsa.edu

***Abstract*—In this paper, we consider the feasibility problem of a set of real-time jobs which may be subject to a fault burst during execution. A fault burst represents a time interval during which multiple jobs may incur faults; hence multiple recoveries may be needed. We show that determining the feasibility of a real-time system, which may be subject to a fault burst that may last at most Δ time units, is an NP-Hard problem even when the exact position of the fault burst is known a priori. However, in a practical system, the fault burst may occur at any arbitrary and unpredictable time. We develop feasibility analysis by assuming *multiple recovery* strategy where, in addition to the job at the end of which the fault is detected, all preempted tasks are also re-executed. We formally characterize the overhead that a scheduler incurs due to a fault burst and present a generic recovery strategy, called Δ-idling, that is shown to minimize the worst-case overhead for any priority-driven scheduling algorithm. Next, we analyze periodic task systems. We show that the preemptive EDF policy, when coupled with Δ-idling, provides the highest possible utilization bound $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$, where P_{min} is the smallest task period. We also present an empirical evaluation of the EDF policy with Δ-idling over synthetically generated task sets, and show that it offers a clear improvement over the naive EDF policy that triggers the recovery tasks as soon as an error is detected.**

I. Introduction

Real-time systems are often deployed for safety-critical applications. The system may encounter faults during execution, which may lead to resource unavailability or incorrect computations. Ensuring reliability for such applications is a crucial objective for real-time research. The fault recovery strategies typically take advantage of hardware [1] or temporal redundancy [2] to ensure successful completion of workload even in the presence of faults. In a single processor system, temporal redundancy is the only viable option. However, the fault management scheme must also take into account the inherent timing constraints.

The characteristics of the fault play an important role in determining the appropriate recovery strategy. Faults can be broadly categorized into two major categories - *permanent* and *transient*. Permanent faults can only be handled with hardware redundancy techniques such as replication [1]. Transient faults occur more frequently [3] and can be dealt with temporal redundancy as well [2]. In most existing fault tolerance studies, transient faults are considered to be instantaneous. As a result, only one job may be affected by a fault. Existing work assumes a known fault distribution [4] or a minimum inter-arrival time between faults [5], [6], [7].

More recently, there has been a growing interest in dealing with scenarios where faults follow a random pattern over a bounded time window and impose a potentially continuous disturbance to the computational activities [8]–[11]. In that new framework called the *fault burst* model, the fault distribution within the interval under consideration is not known. As a result, computations performed during the fault burst interval are potentially unreliable. For example, short voltage fluctuations due to power supply jitters or electromagnetic interference (EMI) caused by short-lived atmospheric effects (e.g., lightning strikes) may cause incorrect computations in the core logic and *bit flips* in architectural registers. Other examples may be found in automotive and avionics areas, where the real-time system controlling a vehicle or unmanned system may be exposed to higher EMI levels when passing near airport radar and communication facilities [8], [10].

Existing studies assume that the upper bound on the duration of the fault burst (Δ) is available in advance [8], [9]. This value is in general domain-specific and its derivation may require input from domain experts. In a study from the automotive domain, Ferreira et al. reported that 90% of errors that occurred in a CAN network resulted from bursts with an average length of 5μsec [12]. Voltage fluctuations are generally reported to be in the nanosecond to microsecond ranges [13], while temperature fluctuations may last for several milliseconds or more [14]. In addition, for a vehicle in motion, the duration of the disturbance may be a function of the distance to the EMI source and the vehicle's velocity.

Fault-burst settings pose serious challenges for hard real-time scheduling. During the fault-burst window, some continuous CPU time intervals become unavailable for computationally correct job executions, limiting opportunities to meet the deadlines. A primary objective of this paper is to contribute to the research efforts that seek to *characterize the conditions under which hard real-time scheduling might become feasible under fault bursts.* For example, if the analysis shows that the maximum fault burst duration (Δ) in the target operation environment exceeds the period/relative deadline of *any* task, obviously it is impossible to give absolute deadline guarantees – a task instance whose execution window entirely coincides with a potential fault burst will definitely miss its deadline.

However, it is not clear what extra conditions may be sought in order to provide feasibility guarantees for all task instances; and those conditions will depend on the recovery semantics as well as the scheduling policy. For example, the

jobs that are "in progress" (i.e., running or in preempted state) at the time of fault burst occurrence may need to be re-executed from the beginning, possibly significantly increasing the workload [8]. This further increases the extent of challenges to ensure feasibility. In fact a contribution of this paper is to show that finding a feasible schedule for a set of real-time jobs is NP-Hard even if the fault burst interval is known a priori. In a real system, the fault burst can occur at any time. As the existing efforts on settings with fault bursts emphasize [8], [9], [11] , when a fault is detected, the scheduler has no way of knowing when exactly the burst has started. This makes optimal recovery decisions very challenging. Timely recovery operations are needed to meet the deadlines; but the recovery operations might also be erroneous if they partially overlap with an ongoing fault burst.

In the presence of fault bursts, there are two major recovery strategies. The system can potentially invoke recovery/re-execution tasks as fault detection tests are performed at the end of each individual job, as needed. This is called the *single recovery* strategy in [8], [9]. This technique has the risk of detecting a fault too late and deadlines can be missed, as some jobs that are in the preempted state may have been also affected by the burst. Another approach is to conservatively estimate the set of jobs that are potentially affected by the fault burst and initiate multiple recovery/re-executions as soon as a fault is detected. This is called the *multiple recovery* strategy [8]. In this strategy, when a fault is detected, the system commits to re-execute the faulty job and all jobs in preempted state at that time. The multiple recovery strategy is also employed in [6] for the instantaneous fault model. The multiple recovery strategy typically reduces the worst-case response time and provides greater feasibility ranges, as shown in [8]. To the best of our knowledge, this is the first work that studies the impact of multiple recovery strategy for a dynamic priority (EDF) based system under a fault burst.

The main contributions of this paper are as follows.

- We show that determining the feasibility of a real-time job set is NP-Hard under *multiple recovery* strategy, even when the fault burst interval is known a priori.
- We formally characterize the computational overhead of a schedule subject to a fault burst at run-time. Based on this characterization, we provide a necessary condition for schedulability under fault bursts.
- We discuss how standard priority-driven algorithms can be adapted to fault burst settings. We propose a specific recovery mechanism, called Δ-idling, that is shown to minimize the worst-case overhead when used in conjunction with any priority-driven algorithm.
- We then turn our attention to *periodic tasks*. We obtain utilization bounds for both frame-based tasks and general periodic tasks. We show that no priority-driven algorithm, in general, can achieve a utilization bound higher than $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$ where P_{min} is the minimum task period, in settings with fault bursts and multiple recovery strategy. Finally, we show that the preemptive EDF-Δ algorithm, which is the EDF policy augmented by Δ-idling, can achieve this bound.
- As our utilization bound provides only a sufficient condition for schedulability, we also empirically evaluate the ratio of the task sets that are schedulable by EDF-Δ over a wider utilization spectrum, through extensive experiments. Our results indicate that EDF-Δ is able to schedule a significantly larger proportion of the task sets, compared to the (naive) EDF policy.

The rest of the paper is organized as follows. We list our workload, fault and recovery models in Section II. In Section III we discuss the challenges faced by a real-time scheduler in the presence of fault bursts. In Section IV we formally characterize the overhead due to a fault burst. Based on this result, we present a necessary condition for schedulability under fault bursts, and derive the worst-case overhead for any scheduler. In Section V, we discuss how priority-based fault-burst-sensitive scheduling algorithms can be designed, and show that the worst-case overhead can be minimized through the Δ-idling scheme for any priority-driven algorithm. In Section VI, we analyze periodic tasks. In Section VII, we present how the ratio of the schedulable task sets varies for EDF, with and without Δ-idling, over a wide range of system parameters. In Section VIII, we present a summary of the fault-tolerant real-time system research that is closely related to our work. Finally, we conclude in Section IX.

II. Preliminaries

A. Workload Model

We consider a set of n real-time jobs $\Psi = \{J_1, J_2, ..., J_n\}$ executing on a single processor system. The release time, deadline, and the worst-case execution time of a job J_i are denoted by r_i, d_i, and C_i respectively. We assume preemptive scheduling. In the first part of the paper, we build our formal framework based on this generic real-time execution model where specific arrival patterns (such as periodicity) are not considered. In Section VI, we focus on and analyze the periodic task model.

B. Fault and Recovery Models

In this paper, we consider faults that occur in bursts [8]. A *fault burst* is a continuous time interval during which the system is potentially subject to multiple *transient* faults whose exact distribution is not known. As a result, all computations performed during this interval are potentially unreliable and should be repeated. Our work is based on the following assumptions as in [8], [9].

- There is an upper bound (Δ) on the length of the fault burst, which satisfies the inequality:
$$\Delta \leq \min_{i=1}^{n}\{d_i - r_i - C_i\}$$
This assumption expresses that the length of the fault burst is less than the minimum laxity of any job in the system. This is essential because, otherwise there is no way to guarantee the feasibility of a job with the smallest $(d_i - r_i - C_i)$ value, in case that a fault burst entirely overlaps with its execution interval.
- The system may be subject to at most one fault burst during execution [8], [9]. In Section VI, we assume that the periodic task set may be subject to multiple fault bursts as long as their temporal separation is at least equal to the hyperperiod. Tightening this result and deriving the bounds that allow shorter intervals between consecutive fault bursts is left as future work.

We consider only the transient faults that affect the core logic (e.g., bit flips in architectural registers or timing errors in CMOS circuits) and result in incorrect computation. For such faults, a typical mechanism is to use *acceptance* or *sanity tests* at the end of job executions [1], [8], [9]. The acceptance tests exploit some expected features of the correct output, such as whether it falls in a reasonable pre-determined range, and for some mathematical control law computations, whether it satisfies certain invariants [1]. Such acceptance tests typically involve simple checks/computations and the running time can be incorporated in the tasks WCETs. Thus we focus on transient faults that can be detected by such acceptance tests. We assume that memory subsystem is protected through separate mechanisms (such as ECCs) and they are out of the scope of this effort. In case that a fault is detected, the output of the job is not committed to, and the system prepares for recovery operations.

When a fault is detected, the system may have already partially executed a number of jobs that may be currently in preempted state. Consequently, the recovery action we assume in this paper consists of *re-executing the faulty job along with all partially completed (preempted) jobs at the time of fault detection*, as in [6], [8], [9]. This recovery strategy is called *multiple recovery* in [8].

Even though full re-execution is assumed as the mode of recovery, the presented results remain valid when the recovery involves executing some alternate job as long as the worst-case execution time of the alternate does not exceed that of the original. Also observe that, the timing of initiating the recovery/re-executions and the order of re-executions depend on the specific scheduling algorithm. Finally, as the fault burst may also affect the recovery jobs, acceptance tests are performed at the end of their execution as well.

C. Fault Burst Feasibility

A set of real-time jobs is said to be Δ-*Fault-Burst-Feasible* (FB-feasible) if there exists a schedule such that all jobs (or their potential re-executions) can complete successfully before their respective deadlines, even in the presence of a fault burst of length not exceeding Δ, occurring at any arbitrary time during execution.

III. Challenges in the Presence of Fault Bursts

Real-time scheduling in the presence of fault bursts presents peculiar difficulties. To start with, by the time a fault is detected, all computations that may have taken place during a potential fault burst need to be considered as unreliable. Instead of resuming preempted jobs and trying to discover their status at the end of their execution (which may be too close to the deadline), we adopt the *multiple recovery* strategy and re-execute all the preempted jobs along with the job at the end of which an error is detected. As a result, the fault burst settings may increase the actual workload significantly at run-time, as several jobs may be in preempted state and completing all the re-executions before the deadlines may be quite difficult unless certain guarantees are made available (e.g., based on the upper bound on the burst length and characteristics of the set of jobs).

Second, re-executions triggered by the recovery mechanism may be also subject to the faults. Consider a system where the upper bound on the fault burst length is given by $\Delta = 10$. Figure 1 shows a re-execution scenario after a fault is first detected at time $t = 28$, the completion time of job J_C with $C = 8$. Suppose the system immediately re-executes J_C. If the actual fault burst has started at $t = 22$, it only affects the original job J_C and its first re-execution. As a result, the second re-execution will be successful. However, if the actual fault burst has just started at $t = 27$, it spans up to the beginning of the second re-execution, potentially causing it to fail as well. In that case, the system may have to initiate a third re-execution, potentially leading to deadline miss.

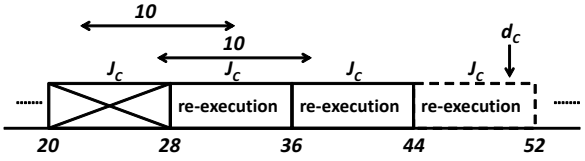

Fig. 1. Possible failure of re-executions

The difficulty of the scheduling problem is not entirely due to the uncertainty about the exact position of the fault burst. Moreover, the existence of a continuous interval during which all computations may potentially fail put a fundamental constraint on the real-time schedulers. In fact, the scheduling problem is intractable even if the exact position of the fault burst is known in advance.

Theorem 1: The problem of determining whether a set of real-time jobs are Δ-FB-feasible is NP-Hard, even when the fault burst is known to occur exactly in the interval $[t_a, t_a + \Delta]$, a priori.

Proof: The proof is presented in the appendix.

We note that our result points only to the weakly NP-Hard nature of the problem – showing whether it is NP-Hard in the strong sense, and if not, investigating polynomial-time approximation algorithms is left as future work.

IV. Overhead of a Fault Burst

The job executions performed during a fault burst interval create additional computational overhead as they are essentially unreliable. In this section, we first characterize the *computational overhead* of a fault burst known to occur in the interval $[t_a, t_b)$ in an actual schedule S. Next, we provide a necessary condition for feasibility using the formalism that we developed. Then, considering that an actual scheduler will not know the exact fault interval at run-time, we characterize the *worst-case overhead* of a fault burst that is manifested at the first fault detection point t_x.

A. Characterizing the Fault Burst Overhead

The overhead of a fault burst essentially captures the computation time wasted. According to the *multiple recovery* strategy the jobs that execute during the fault burst, as well as those that are in preempted state at time t_a need to be re-executed [8]. Let $\alpha_k^S(t_1, t_2)$ represent the CPU time allocated to a job J_k during the interval $[t_1, t_2)$. First, we consider all the jobs that execute (partially or completely) during the fault burst interval $[t_a, t_b)$. We use the set $Y^S(t_a, t_b)$ to denote those jobs:

$$Y^S(t_a, t_b) = \{J_k \mid \alpha_k^S(t_a, t_b) > 0\}$$

Also we denote the set of jobs that are in preempted state at time t_a and that do not execute in $[t_a, t_b)$, by $H^S(t_a, t_b)$:

$$H^S(t_a, t_b) = \{J_k \mid J_k \notin Y^S(t_a, t_b) \; and \; 0 < \alpha_k^S(r_k, t_a) < C_k\}$$

With *multiple recovery* strategy, we need to re-execute all jobs in $Y^S(t_a, t_b)$ and $H^S(t_a, t_b)$. Hence, the set of all the jobs that require re-execution is the following:

$$X^S(t_a, t_b) = Y^S(t_a, t_b) \; \cup \; H^S(t_a, t_b)$$

The *overhead* of a fault burst is defined as the total execution time consumed by these jobs *outside* the actual fault burst interval. We can formally define the overhead due to a fault burst occurring at $[t_a, t_b)$ as:

$$O^S(t_a, t_b) = \sum_{J_k \in X^S(t_a, t_b)} \alpha_k^S(r_k, t_a) + \sum_{J_k \in X^S(t_a, t_b)} \alpha_k^S(t_b, d_k) \quad (1)$$

Observe that, the overhead consists of two terms for each job affected by the fault burst. The first term is the CPU time used by the jobs before the fault burst starts and the second term represents the CPU time used after the end of the fault burst. Note that, we consider re-execution of a job as a separate job.

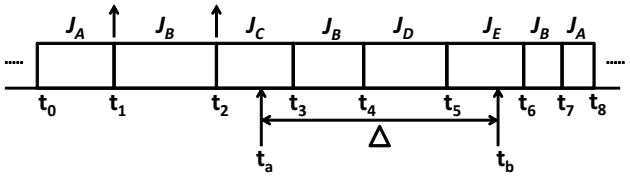

Fig. 2. Overhead of a fault burst

Figure 2 shows an example with a fault burst occurring in time interval $[t_a, t_b)$. J_B, J_C, J_D and J_E are executed during the fault burst. As a result they will all eventually fail. According to our notation, $Y(t_a, t_b) = \{J_B, J_C, J_D, J_E\}$. J_A is preempted at time t_1, but did not get a chance to execute until $t = t_7$, when the fault burst has already ended. So, $H(t_a, t_b) = \{J_A\}$. The total CPU time allocated to these jobs before t_a and after t_b will contribute to the fault overhead. As a result the overhead will consist of the following:

$$O(t_a, t_b) = (t_1 - t_0) + (t_8 - t_7) + (t_2 - t_1) + (t_7 - t_6) + (t_a - t_2) + (t_6 - t_b)$$

B. Impact of Overhead on Feasibility

A job's execution is considered successful if its first invocation, or one if its re-executions, completes without a fault before its deadline. Obviously, the fault burst reduces the effective computation time available for job execution. The overhead makes the situation even more critical. We now analyze the impact of the overhead on the feasibility.

We denote by $S(A, t_a, t_b)$ the schedule generated by any arbitrary algorithm *A* when the fault burst occurs in interval $[t_a, t_b)$. Next, we define the overhead of a fault burst for a specific subset of jobs. The overhead of a fault burst for a subset of jobs Γ is defined as:

$$O_\Gamma(t_a, t_b) = \sum_{J_k \in \{X(t_a, t_b) \; \cap \; \Gamma\}} \alpha_k(r_k, t_a) + \sum_{J_k \in \{X(t_a, t_b) \; \cap \; \Gamma\}} \alpha_k(t_b, d_k) \quad (2)$$

Observe that, in Equation (2), we only consider the computational overhead incurred by the jobs in Γ. In (2) and below, we omit the superscript from the related variables when the involved schedule is clear from the context. Finally, we define the following:

$$\gamma(t_1, t_2) = \{J_k \mid r_k \geq t_1 \; and \; d_k \leq t_2\} \quad (3)$$

$\gamma(t_1, t_2)$ denotes the jobs that must be completed successfully within $[t_1, t_2)$. Using this definition, we provide the following necessary condition for feasibility of a job set.

Theorem 2: A job set Φ is Δ-FB-feasible, only if there exists an algorithm *A* such that for every $(t_a, t_b = t_a + \Delta)$ fault burst interval and for every (t_1, t_2) time interval, the following holds for the generated schedule $S(A, t_a, t_b)$:

$$\sum_{J_k \in \gamma(t_1, t_2)} C_k + O_{\gamma(t_1, t_2)}(t_x, t_y) + (t_y - t_x) \leq (t_2 - t_1) \quad (4)$$

where, $[t_x, t_y)$ denotes the time interval where the fault burst overlaps with the (t_1, t_2) interval under consideration, namely, $t_x = \max\{t_1, t_a\}$ and $t_y = \min\{t_b, t_2\}$

Proof: The proof is presented in the appendix.

C. The Worst-Case Overhead of a Fault Burst

Theorem 2 provides only a necessary condition for feasibility under any arbitrary occurrence time of the fault burst, given that the schedule up to the fault burst interval is given. Moreover, the feasibility condition depends on the overhead of the fault burst. The overhead can be precisely computed for any deterministic scheduling algorithm using Equation (1), if the exact location of the fault burst is known. However, at the time of fault detection during actual execution, the scheduler has no way of knowing the exact start time of the fault burst. Therefore, to ensure feasibility, *the scheduler must satisfy the feasibility condition for all possible fault burst scenarios* and must schedule the recovery jobs appropriately.

Notice that, the overhead of a fault burst also depends on the decision made by the scheduler *after* the fault has been detected. An immediate re-execution may be also subject to potential new faults. On the other hand, if the system is kept idle beyond the fault burst, the system will lose valuable execution time. Therefore, we define a new metric, called the *Worst-Case Overhead* (WCO) to denote the maximum overhead that a scheduling algorithm may incur for any arbitrary fault burst interval, when a fault is detected at a given time.

Assume that at run-time, a scheduling algorithm A detects a fault at the completion of a certain job J_k at time t_x. We denote the schedule generated by *A* is $S(A)$. Then we define the worst-case overhead of the algorithm A by:

$$WCO_A(t_x) = \max_{t_a \in T}\{O^{S(A)}(t_a, t_a + \Delta)\} \quad (5)$$

Here T is all possible time points where the fault burst may have started, between $t = r_k - \Delta$ and $t = t_x^-$.

Example: Consider a preemptive priority-based scheduling policy which employs the multiple recovery strategy. When a fault is detected, the system immediately starts re-executing the failed jobs. Figure 3 shows a fault detection scenario for such a system. J_A is preempted at $t = 5$ by a high priority job J_B, which, in turn, is preempted at $t = 10$ by a higher priority job J_C. A fault is detected when J_C completes at $t = 20$. The length of the fault burst is 12. Notice that, at that time, the scheduler has no way of knowing the exact start time of the fault burst. In Figure 3, we show three possible scenarios. The fault burst may have ended before detection, as shown in the cases when the fault burst starts at $t = 3$ or $t = 8$. The overhead for the fault burst for these cases is 8. On the other hand, if the fault burst starts at $t = 15$ the first re-execution will fail as well. In that case, the overhead will be 18.

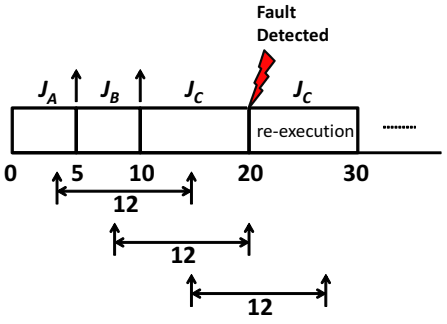

Fig. 3. Possible overheads for different fault burst intervals

Now assume that the deadline of J_C is 48 and the fault burst starts at time 20 right before the completion of J_C as shown in Figure 4. Then the fault burst will last until $t = 32$, causing two re-executions to fail. The overhead for the fault burst will be 28. The third re-execution will be fault-free, but a deadline will be missed. Hence it is imperative to characterize the worst-case scenarios in the feasibility analysis.

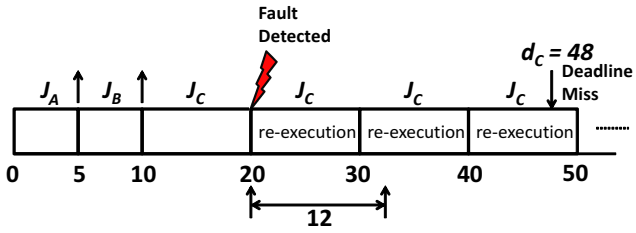

Fig. 4. Deadline miss due to overhead of a fault burst

V. Fault-Burst Sensitive Scheduling

A. Adapting Existing Scheduling Frameworks to Fault Burst Settings

The precise definition of the (worst-case) overhead for an arbitrary scheduling algorithm is critical for real-time guarantees. However the *design* of fault burst sensitive real-time scheduling algorithms requires additional considerations; in particular, it requires the specification of the algorithm's actions *before* and *after* the detection of the fault. Notice that the multiple recovery settings imply that at the time of fault detection (at the end of a job J_k), J_k along with all preempted jobs at that time, must be re-executed from scratch. We refer to this additional workload as "recovery load" (or recovery jobs) in the rest of the paper.

Priority-Driven Real-Time Scheduling Algorithms [15], are well known and widely used. A priority driven algorithm, at any time, schedules the ready job with the highest priority, and does not leave the CPU idle as long as there are pending jobs. Common examples are the earliest-deadline-first (EDF) and fixed-priority algorithms.

In this work, we use existing priority-driven real-time scheduling algorithms as the basis for designing effective schemes to tolerate fault bursts. In particular, we assume that,

i. The priority assignment policy is deterministic and does not change at run-time.

ii. The job that is scheduled at any time has the highest priority among all ready jobs. This applies to both original jobs and recovery jobs. That is, an original job may have higher priority with respect to a recovery job as in [16], [17].

iii. On the other hand, in the post-detection part, the algorithm *may* choose to idle even when there are ready jobs, which may include recovery jobs.

We call an algorithm that satisfies all three conditions above a *Fault-Burst Sensitive Priority Driven* scheduling algorithm. Obviously a natural fault-burst sensitive priority driven algorithm would be based on simply using a well-known priority driven algorithm (such as EDF), and use it in non-idling fashion, even in the post-detection part. While this strategy is optimal in settings where faults are *instantaneous* and affect *only one job* [16], it is clearly sub-optimal in fault burst settings. The main reason is that with the fault bursts the re-execution of the recovery jobs may be still subject to faults, further increasing the recovery load.

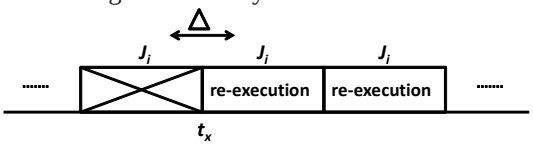

Fig. 5. Failure of re-execution for a short fault burst

In fact, a quick analysis shows that even when Δ is of short duration, it may be necessary to execute a job 3 times (once for original and two recoveries), in case that fault burst overlaps partially with the first re-execution as shown in Figure 5. A similar analysis shows that $\left\lceil \frac{\Delta}{C_i} \right\rceil + 1$ re-executions of a faulty job J_i may be needed for *any* non-idling algorithm. As shown in Figure 6, assume that the fault burst starts right before t_x, the completion time of J_i. Then, when $C_i < \Delta < 2C_i$, we will need 3 re-executions for successful completion.

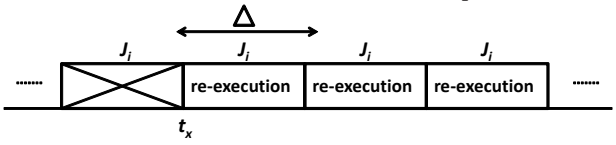

Fig. 6. Maximum re-executions required for a fault burst

Note that, considering possible re-execution of the pre-empted jobs in the potential fault burst region would even further increase the recovery load. Using our terminology, we can see that *non-idling strategies have the potential of significantly increasing the worst-case overhead*.

B. The $\Delta-$*idling Scheme*

We now propose a generic fault burst sensitive priority driven algorithm satisfying all three conditions given in Section V-A. The algorithm, called $\Delta-$*idling*, is generic in the sense that it can be used in conjunction with any standard priority driven scheduling algorithm.

The $\Delta-$idling scheme essentially consists of *deliberately leaving the CPU idle for* Δ *units of time* (the known upper bound on the length of the fault burst), *upon detection of the fault. At the end of this period, the scheduler continues with the execution of the pending load (including recovery) according to the original priority assignment.*

The main rationale for this scheme is to prevent the system from executing additional jobs in a region which may be still affected by the fault burst. With Δ-idling, the re-executions are never subject to faults, hence each job may need at most one re-execution before successful completion. Although leaving the system idle for Δ time units results in loss of some computation time, as we show shortly, it minimizes the worst case overhead, and allows us to derive tight bounds (not achievable by any non-idling algorithm) for schedulability guarantees.

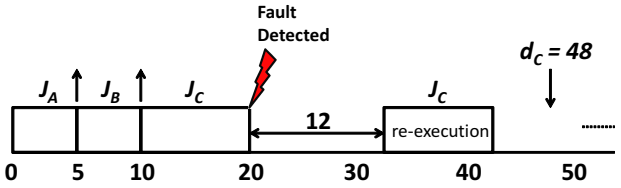

Fig. 7. Overhead for the $\Delta-$idling scheme

Example: Let us consider the same example as in Section IV-C. Figure 7 shows the schedule with $\Delta-$idling. A fault is detected at $t = 20$ and the length of the fault burst is 12. In that case, the fault burst is guaranteed to end by $t = 32$. In fact, $\Delta-$idling will keep the system idle until $t = 32$. As a result, all future re-executions will be successful. Therefore, irrespective of the actual position of the fault burst, the overhead will not exceed 20 and the deadline will be met.

We now prove the optimality of $\Delta-$idling for minimizing the WCO for a fault burst detected at time t_x. Assume that Algorithm *A* is a standard priority based scheduling algorithm. The recovery is performed with multiple recovery strategy. At the time of fault detection, the system can use $\Delta-$idling or any arbitrary recovery strategy. We denote the Δ-idling variant of the algorithm by $A - \Delta$. The high level idea of the proof is as follows. The overhead depends on the jobs that are in preempted state at the start of the fault burst and the jobs that execute during the fault burst. Prior to fault detection, the schedule is the same for all recovery strategies. Therefore, the set of jobs in preempted state at the start of the fault burst will remain the same for all recovery strategies. By idling in the interval $[t_x, t_x + \Delta)$, $\Delta-$idling prevents further increase of the overhead.

Theorem 3: For any arbitrary scheduling algorithm *A* with multiple recovery strategy, $\Delta-$idling minimizes the worst-case overhead, $WCO(t_x)$ when a fault burst is detected at time t_x, among all algorithms that use *A* up to $t = t_x$.

Proof: The proof is presented in the appendix.

VI. Schedulability and Utilization Bounds for Periodic Tasks

In this section, we consider periodic task sets that represent many real-time applications in practice. Specifically, we have a set of n periodic real time tasks $\Psi = \{T_1, T_2, ..., T_n\}$. Associated with each task T_i, there is a worst-case execution time C_i, period P_i, and the relative deadline D_i. We assume implicit-deadline systems where $D_i = P_i$. The j^{th} job of T_i, denoted by $J_{i,j}$, is released at $t = P_i \cdot (j-1)$ and must be completed by its deadline at $t = P_i \cdot j$. The least common multiple (LCM) of the task periods is called the *hyperperiod*.

The utilization of a task T_i is defined as $u_i = \frac{C_i}{P_i}$. Similarly, the total utilization of the system is given by $U_{tot} = \sum_{i=1}^{n} u_i$.

For the schedulability analysis of periodic tasks with fault bursts, we make the following assumptions.

- $\Delta < P_{min}$. This assumption ensures that the length of the fault burst is less than the smallest period in the system. This assumption is necessary to analyze the conditions for hard real-time scheduling. Otherwise, the task with smallest period will definitely miss deadline, when the fault burst completely overlaps with its period.
- The two consecutive fault bursts are separated by at least one hyperperiod.

We define utilization bound for fault burst settings as follows.

Δ-Sensitive Utilization Bound: A scheduling algorithm *A* is said to have the Δ-*sensitive utilization bound* (U_b^{Δ}), if it can generate a feasible schedule for any periodic task set with utilization $U_{tot} \leq U_b^{\Delta}$, for any fault burst of length not exceeding Δ, occurring at any arbitrary time.

A. Frame-Based Systems

We now consider a special case of periodic tasks, called frame-based systems, where all tasks are released at the same time and have a common period/deadline, P. In this setting, a natural solution is to execute all tasks one after another according to a pre-speficied order. We call this algorithm the *Chain-Execution* algorithm. Clearly, in the absence of faults, the task set is feasible if and only if $\Sigma C_i \leq P$.

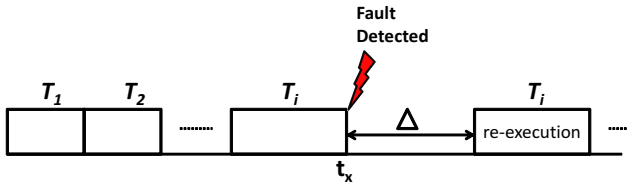

Fig. 8. $\Delta-$idling for frame-based system

Note that, since all fault detections occur at the end of task executions and there are no preemptions, the chain-execution has a clear advantage in fault burst settings: only the faulty task T_i will need to be re-executed. Suppose a fault is detected at $t = t_x$ at the completion of T_i. We can now apply Equation (5) and obtain the worst-case overhead for any chain-execution algorithm as:

$$C_i \leq WCO_{chain}(t_x) \leq C_i + Q \tag{6}$$

where $Q \geq 0$ is the additional overhead due to task execution (or re-executions) in $[t_x, t_x + \Delta)$.

Algorithm Chain-Δ: The $\Delta-$idling scheme can be used in conjunction with the chain execution algorithm. The resulting algorithm is called the *chain-Δ* algorithm. In accordance with the $\Delta-$idling scheme, tasks are executed according to the chain algorithm until a fault is detected. Then the system is kept idle for Δ time units. After that, the system resumes the re-execution of the failed task and the rest of the tasks. Figure 8 shows a schedule segment generated by the chain-Δ algorithm. The worst-case overhead for chain-Δ algorithm is

$$WCO_{chain-\Delta}(t_x) = C_i \tag{7}$$

as it does not execute any task in the interval $[t_x, t_x + \Delta)$ and there are no preemptions. According to Theorem 3, this is the lower bound for the worst-case overhead for any chain-execution based real-time scheduling algorithm with multiple recovery strategy.

Theorem 4: A frame-based system scheduled by the Chain-Δ algorithm is Δ-FB-feasible, if and only if,

$$\Sigma_i C_i + C_{max} \leq P - \Delta \tag{8}$$

Proof: The proof is presented in the appendix.

The result above enables us to derive the Δ-Sensitive Utilization Bound for the Chain-Δ algorithm as well. Let us define $u_{max} = max\{\frac{C_i}{P}\}$. By simple algebraic manipulation on Equation (8), we get:

$$U_{tot} \leq 1 - \frac{\Delta}{P} - u_{max}$$

Since the condition is necessary and sufficient, for any task set with total utilization not exceeding $(1-\frac{\Delta}{P}-u_{max})$, there exists a feasible schedule for any arbitrary fault burst occurrence time. Conversely, if the total utilization of a task set exceeds $(1 - \frac{\Delta}{P} - u_{max})$, there is no feasible schedule for some fault burst arrival pattern(s).

Corollary 1: $U_b^{\Delta} = (1 - \frac{\Delta}{P} - u_{max})$ for the Chain-Δ algorithm and the bound is tight.

B. General Periodic Tasks

In this section, we consider the well-known preemptive Earliest-Deadline-First (EDF) scheduling policy for general periodic tasks, where tasks may have different periods. EDF is known to be optimal for preemptive scheduling of real-time tasks on a single processor system in a fault-free scenario. However, preemptions can occur and more than one task may be affected by a single fault burst.

We now present our analysis of the utilization bound for periodic tasks. We first show that no scheduling algorithm can achieve a Δ-sensitive utilization bound higher than $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$.

Theorem 5: For any arbitrary scheduling algorithm *A*, $U_b^{\Delta} \leq \frac{1}{2}(1 - \frac{\Delta}{P_{min}})$.

Proof: Assume that the Δ-sensitive utilization bound for an arbitrary algorithm *A* exceeds $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$. So there must be also a single-task system with computation time C and $P = P_{min} = D$ and with utilization greater than $\frac{1}{2}(1 - \frac{\Delta}{P_{min}}) = \frac{1}{2}(1 - \frac{\Delta}{D})$ that can be feasibly scheduled by *A* for any arbitrary fault burst occurrence time. According to our assumption,

$$\frac{C}{D} > \frac{1}{2}(1 - \frac{\Delta}{P_{min}}) \tag{9}$$

Now assume that the fault burst occurs right before the completion of the task. Then all re-execution attempts prior to the completion of the fault burst will also fail. Therefore, the re-executions must be performed after the fault burst has ended. So, $C + \Delta + C \leq D = P = P_{min}$ must hold

By simple algebraic manipulation we obtain,

$$\frac{C}{D} \leq \frac{1}{2}(1 - \frac{\Delta}{P_{min}}) \tag{10}$$

Equations (9) and (10) contradict. Therefore, the utilization bound for *A* can not exceed $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$. □

Now let us consider the $\Delta-$idling scheme with the preemptive EDF scheduling policy, called EDF-Δ. Tasks are executed according to the preemptive EDF policy. When a fault is detected, the system is kept idle for Δ units of time. When the execution resumes, the re-executions are performed according to the EDF policy along with the original tasks.

Theorem 6: For a periodic task set scheduled by the EDF-Δ policy, $U_b^{\Delta} = \frac{1}{2}(1 - \frac{\Delta}{P_{min}})$.

The proof of Theorem 6 is presented in the appendix.

Combining this result with Theorem 5, we obtain the following corollary:

Corollary 2: EDF-Δ provides the highest utilization bound achievable for periodic task sets.

Remark 1. While Theorem 6 is presented for periodic tasks, the proof remains valid for *sporadic* tasks [18], when consecutive instances of a given task are separated by *at least* the period value. Therefore, for sporadic tasks the number of task instances in a specific time duration, can not be higher than the case with the strictly periodic tasks. As a result, the sum of computational demand in any interval will be still bounded by that of a strictly periodic task set in that interval, and the reasoning will follow the same lines.

Remark 2. Even though EDF-Δ gives the highest possible utilization bound, we underline that this does not imply that it can schedule any task set under a *given (specific)* fault burst pattern whenever it is possible to do so. One could imagine a scenario where the fault burst ends just before the fault detection time, and EDF-Δ misses a deadline due to the idling period, whereas a non-idling algorithm could meet the deadline. Due to the nature of uncertainty involved in the fault burst settings, an "adversary" can always change the actual position of the fault burst to demonstrate that the action taken by a specific algorithm was suboptimal. EDF-Δ, on the other hand, provides the best/highest utilization bound that can be obtained by any deterministic priority driven fault burst scheduling algorithm. The bound for any non-idling algorithm is strictly smaller than $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$. To verify this, one can consider a single-task scenario with $C = 40, D = P = 100$ and $\Delta = 20$. For EDF-Δ, the utilization bound is 0.4. Therefore the task set can be scheduled feasibly by EDF-Δ under any fault burst pattern.

Now consider a non-idling schedule and suppose that the fault burst starts right before the completion of the task. As a result, the fault burst will affect the first re-execution as well. The task will complete at $t = 120$ at the earliest, giving a deadline miss.

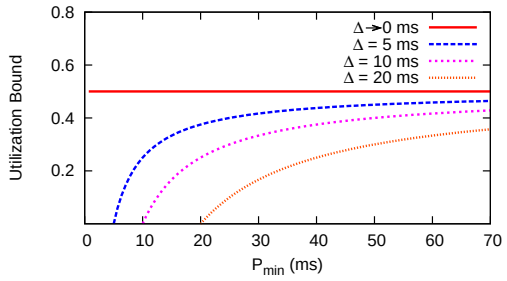

Fig. 9. Utilization Bound

Figure 9 shows how the utilization bound varies as a function of the minimum task period, for various fault burst lengths. The maximum utilization bound for any task set is 0.5. If the length of the fault burst is greater than the minimum period, no scheduling algorithm can guarantee feasibility. As $\Delta \rightarrow 0$, the settings converge to those of the instantaneous faults with multiple recovery strategy [6], giving the bound of 0.5. As the duration of the burst length Δ increases, a marked decrease in the bound is observed; however, for task systems with larger P_{min} values the reduction in the utilization bound is relatively minor.

VII. Experimental Evaluation

Theorem 6 in Section VI-B indicates that EDF-Δ provides the highest utilization bound to schedule periodic task sets. Yet, the bound provides only a sufficent condition for schedulability, and is naturally derived under worst-case scenarios over all task sets. We believe the following questions warrant an empirical evaluation:

- For task sets with utilization greater than the bound of $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$, how does the fraction of sets that are schedulable by EDF-Δ change, as a function of the system parameters?
- How does the average performance of EDF-Δ compare to that of the naive EDF (which triggers re-executions immediately when a fault is detected)?

For this purpose, we constructed a discrete event simulator to check the feasibility of synthetically generated task sets for various burst length parameters. Given a task set and a burst of length Δ, we assessed the feasibility for both EDF-Δ and EDF, i.e., *checked whether the task set remains feasible with these two strategies, for any burst length of length Δ that may start anywhere during the hyperperiod*. Obviously, simulating the execution for all possible fault burst start points is computationally prohibitive. Let X be the set of job completion points in the fault-free EDF schedule. We make the following observation:

Proposition 1: For the EDF-Δ policy, to assess the feasibility under a fault burst of length Δ, it is sufficient to consider the fault burst start points that immediately precede the set of job completion points X.

Proposition 1 may be justified through the following observation. All fault bursts will be detected at a subsequent job completion point. Note that there may be multiple fault burst start points $\{t_f\}$ that are all detected at the same job completion point $t_s > t_f$. For all such distinct fault scenarios, the schedule up to time t_s (the first fault detection point) will be identical. Moreover, the EDF-Δ algorithm will leave the CPU idle in the interval $(t_s, t_s + \Delta]$ for all such scenarios, effectively eliminating the possibility that some new faults will be incurred *after* the fault detection point t_s. Finally, for all such fault occurrence points $\{t_f\}$, the schedule after $t = t_s + \Delta$ will be identical in the EDF-Δ schedule. Thus, the feasibility impact of all those fault scenarios can be assessed by considering only one such scenario that leads to the fault detection at $t = t_s \in X$, in particular the one that starts at time $t = t_s^-$.

However, for naive (non-idling) EDF, the feasibility cannot be accurately assessed by considering only the fault bursts starting immediately before the job completion points in the fault-free EDF schedule. This is because, naive EDF will dispatch jobs immediately after fault detection, which may cause additional jobs to fail and trigger recurring fault detections. As a result, unlike EDF-Δ, the final schedule will depend on the subsequent jobs that are affected by the fault burst. So, we need to consider additional fault burst intervals which may end at different times after $t = t_s^-$, resulting in a different set of affected recoveries and schedules. Nevertheless, it is possible to obtain an *upper bound* on the feasibility performance of the naive EDF by considering the fault bursts that start at time $t = t_s^-$ $(\forall t_s \in X)$. We implemented that (optimistic) approximation in the interest of computational time needed for simulations. Our results below indicate that EDF-Δ outperforms even this optimistic approximation of the naive EDF with immediate recovery.

In our evaluation, for each data point, we conduct 100 experiment. Each experiment has at least 200 task sets to obtain simulation results within 97.5% confidence interval. For each experiment we obtained the fraction of task sets that are feasibly scheduled (called the *feasibility ratio*), by EDF-Δ and the naive EDF, separately. As discussed above, by limiting the fault burst start points to those immediately preceding the job completion points in the fault-free EDF schedule, we obtain the exact feasibility ratio for EDF-Δ and an (optimistic) approximation of the feasibility ratio for the (naive) EDF. The task utilization values are generated according to *UUnifast* scheme [19]. The minimum and maximum allowable utilization per task are set to 0.5 and 30% of total utilization, respectively. The task periods are generated using the log-normal distribution, between 25 and 1000 ms. Those values correspond to the minimum and maximum task periods in the well-known GAP task set from the avionics domain [20]. The execution time each task is obtained by multiplying its period and utilization value. By default, we considered 15 tasks in each set. For each task set, we consider fault bursts of length not exceeding the minimum period of the task set.

Figure 10 shows the feasibility ratio of the EDF-Δ scheme for different task utilization and fault burst length values on a 3-D plot. For small utilization ($U \leq 0.3$) values the feasibility ratio is very close to 1. As we increase the utilization the feasibility drops; but even for $U = 0.7$ in practice the

ratio exceeds 60% except for when the fault burst length Δ approaches $P_{min} = 25ms$. In fact for small Δ values (e.g., $\Delta = 5ms$) the ratio is significantly high even when $U = 0.9$.

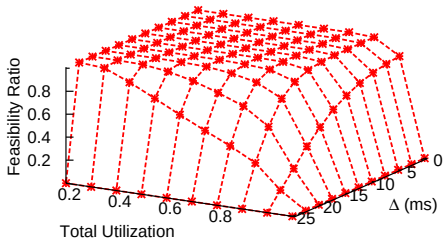

Fig. 10. Feasibility Ratio for EDF-Δ

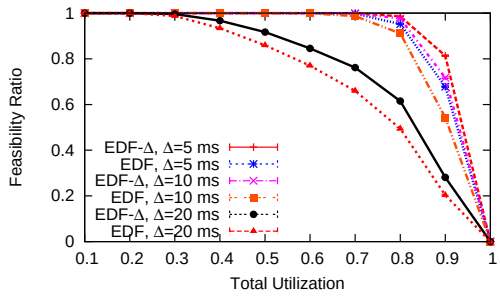

Fig. 11. Impact of Utilization

We next compare the performance of EDF-Δ and naive EDF. Fig. 11 shows how the feasibility ratios vary as a function of the utilization for different burst lengths. We observe that EDF-Δ outperforms EDF in the entire spectrum, for the same fault burst length. For example, when $\Delta = 5ms$, EDF-Δ maintains a feasibility ratio of at least 0.8 as long as $U < 0.9$; whereas EDF's performance can be as low as 0.75. Fig. 12 illustrates the same phenomenon over a continuous spectrum of the fault burst length Δ. We observe that for large Δ values, the feasibility performance rapidly drops, but EDF-Δ exhibits superiority over naive EDF for all values.

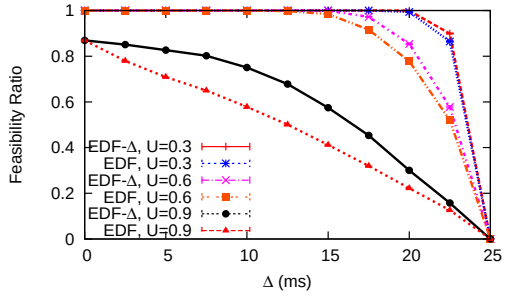

Fig. 12. Impact of Δ

Figure 13 shows the impact of number of tasks in each task set on the feasibility ratio. We set the utilization to 0.7 and present the feasibility ratio for different Δ values. Notice that the feasibility ratio increases with the number of tasks for both schemes. The main reason is that as the number of tasks increases for the same utilization value, we have more job completion points and acceptance tests, leading to faster fault detection. Again, EDF-Δ has superior performance.

In the presence of fault bursts, an important parameter is the *fault resilience* of a task set, which is defined as the

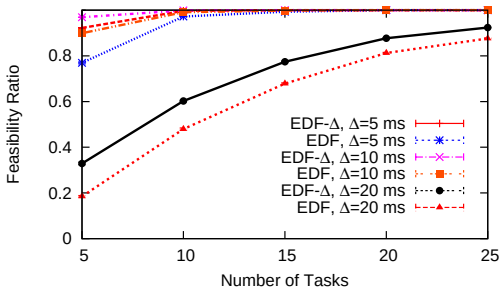

Fig. 13. Impact of Number of tasks

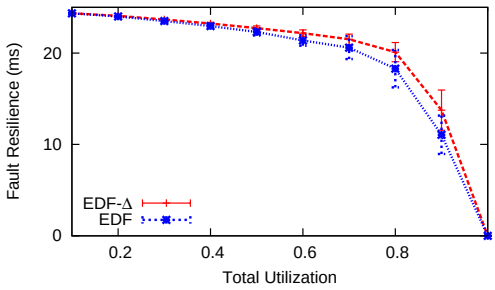

Fig. 14. Fault Resilience

maximum length of the fault burst the set can sustain while still maintaining the feasibility [8]. For every task set at various utilization values, we performed a binary search on the length of the fault burst values (between 0 and $P_{min} = 25$ ms) and check the feasibility for each, to find the fault resilience of the corresponding task set. The average of task fault resilience values is presented in Fig. 14. At very low utilization, there is ample slack in the system and therefore the fault resilience is close to $P_{min} = 25$ms. As we increase the utilization, there is less time available for task re-executions. As a result, the fault resilience drops and at higher utilization it approaches 0. Notice that the fault resilience values obtained for EDF-Δ are better than those of the naive EDF scheme.

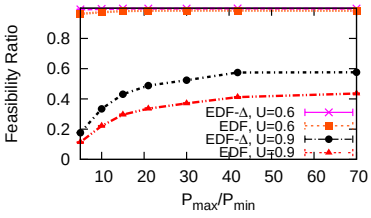

Fig. 15. Impact on feasibility

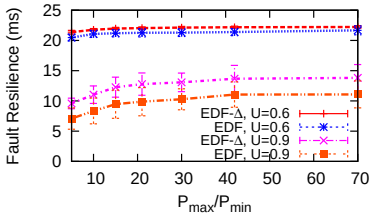

Fig. 16. Impact on fault resilience

In the experiments reported so far the ratio of the maximum period P_{max} to the minimum period P_{min} was $\frac{1000ms}{25ms} = 40$. We next consider the impact of varying P_{max} from 125 ms to 1750 ms, for both the feasibility ratio and fault resilience in Fig. 15 and Fig. 16, respectively. We observe only minor variations with increasing P_{max} values, as the most important parameters that determine feasibility are the smallest task period P_{min} and Δ values for a given total utilization. However, increasing P_{max} value gives some minor leverage for feasibility and fault resilience as the number of preemptions that may affect the execution of short-period tasks decreases.

VIII. Closely Related Works

Fault tolerant real-time scheduling is a well-studied research area. Since transient faults are more common [3], most of the work is dedicated to tolerating transient faults. In earlier works, faults are considered to be instantaneous and fault arrivals were assumed to follow some known pattern. For example in [4] faults are considered to follow Poisson or Weibull distribution. The works in [5], [6], [7] assumed a minimum inter-arrival time between faults. This is called the *pseudo-periodic* fault model in [21]. Later works considered tolerating a fixed number of faults during the entire execution [16], [17]. [16] is more generalized in that it allows recovery jobs with execution time different than the original jobs.

Many et al. proposed the *fault burst* model [8] to handle faults that occur in quick succession over a continuous time interval with an unknown distribution. In [8], the authors considered fixed-priority systems such as *Rate Monotonic* (RM) [22] or *Deadline Monotonic* (DM) [23] scheduling and computed the worst-case response time for a task in the presence of a fault burst. Using the worst-case response time the authors proposed a technique for evaluating the feasibility of a task set. In [11], the authors extended their work to determine the maximum length of the fault burst to guarantee feasibility, again for fixed-priority systems. Aysan et al. [24] considered a probabilistic fault burst model for the fixed priority systems and obtained probabilistic feasibility guarantees. Finally, in [9], the authors considered the *Earliest-Deadline-First* (EDF) policy. The work in [9] computes the worst-case increase in processor demand during an interval. Using this result, the authors presented the minimum resource augmentation required to ensure feasibility. In [9] the authors also presented a necessary condition for the feasibility of a periodic EDF system under fault burst. The work in [9] and our paper differ in that we consider *multiple recovery* strategy, while [9] considers *single recovery* strategy.

In [10], the authors considered a quantile-based approach to obtain stochastic schedulability guarantees for a system under a random burst of errors. Instead of considering a fault burst interval, the authors considered arbitrary arrival of multiple errors. Given a probabilistic error model, [10] provides a bound for the number of faults that a system should tolerate in order to achieve an arbitrary reliability target.

Utilization bounds provide efficient tests to check the feasibility of periodic tasks. For preemptive EDF, in the fault free scenario, the utilization bound is proved to be 1 [22]. The same utilization bound holds for sporadic tasks [18]. For RMS, the utilization bound approaches $\ln 2 \simeq 0.69$ for large number of tasks [22]. However, in the presence of a fault, the utilization bound decreases significantly. For RMS, even if the fault is instantaneous, the utilization bound reduces to 0.5 [6].

IX. Conclusions

In this paper, we considered the feasibility analysis of real-time jobs under fault burst with multiple recovery strategy. We first addressed a generic real-time system with a set of distinct jobs. We showed that determining the feasibility of such a system is NP-Hard even when the exact position of the fault burst is known. The difficulty of the problem only increases for a real system as the fault burst interval is not known precisely. We proposed a solution called the $\Delta-$idling scheme, which minimizes the worst-case waste of execution time (overhead) with multiple recovery strategy. Then we considered two special but common periodic task systems. We derived an utilization bound for the frame-based system, where all jobs have a common deadline. Finally, we considered general periodic tasks systems. We showed that the utilization bound to guarantee feasibility for EDF when augmented by Δ-idling scheme is $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$. We also proved that no scheduling algorithm for periodic tasks can achieve a higher utilization bound. Finally, we presented an empirical evaluation with synthetically generated task sets, showing that EDF with Δ-idling gives a significant improvement on the ratio of the schedulable task sets when compared to the naive EDF. In the future, we plan to extend this work by considering multiple fault bursts that can occur during the execution of the aperiodic job sets, and fault bursts that may be separated by an interval shorter than the hyperperiod, for periodic task sets. In addition, we plan to explore the problem on multicore systems where the individual cores or a group of cores are simultaneously affected by a fault burst.

Acknowledgments

This work was supported by US National Science Foundation awards CNS-1016855, CNS-1016974, and CAREER Award CNS-0953005.

References

[1] D. K. Pradhan, *Fault-tolerant Computer System Design*. Upper Saddle River, NJ: Prentice-Hall, 1996.

[2] H. Aydin and D. Zhu, "Reliability-aware energy management for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 58, no. 10, pp. 1382–1397, 2009.

[3] R. K. Iyer, D. J. Rossetti, and M.-C. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 3, pp. 214–237, 1986.

[4] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and calibration of a transient error reliability model," *IEEE Transactions on Computers*, vol. 31, no. 7, pp. 658–671, 1982.

[5] G. de A Lima and A. Burns, "An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1332–1346, 2003.

[6] M. Pandya and M. Malek, "Minimum achievable utilization for fault-tolerant processing of periodic tasks," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1102–1112, 1998.

[7] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Journal of Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[8] F. Many and D. Doose, "Scheduling analysis under fault bursts," in *Proc. of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[9] A. Thekkilakattil, R. Dobrin, S. Punnekkat, and H. Aysan, "Resource augmentation for fault-tolerance feasibility of real-time tasks under error bursts," in *Proc. of the 20th ACM International Conference on Real-Time and Network Systems*, 2012.

[10] M. Short and J. Proenza, "Towards efficient probabilistic scheduling guarantees for real-time systems subject to random errors and random bursts of errors," in *Proc. of the 25th IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[11] F. Many and D. Doose, "Fault tolerance evaluation and schedulability analysis," in *Proc. of the ACM Symposium on Applied Computing*, 2011.

[12] J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca, "An experiment to assess bit error rate in CAN," in *Proc. of 3rd International Workshop of Real-Time Networks*, 2004, pp. 15–18.

[13] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," in *Proc. of the 9th IEEE International Symposium on High-Performance Computer Architecture*, 2003.

[14] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 336–349, 2003.

[15] J. W. S. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice-Hall, 2000.

[16] H. Aydin, "Exact fault-sensitive feasibility analysis of real-time tasks," *IEEE Transactions on Computers*, vol. 56, no. 10, pp. 1372–1386, 2007.

[17] F. Liberato, R. Melhem, and D. Mossé, "Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems," *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 906–914, 2000.

[18] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proc. of 11th IEEE Real-Time Systems Symposium*, 1990, pp. 182–190.

[19] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Journal of Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[20] C. D. Locke, D. R. Vogel, and T. J. Mesler, "Building a predictable avionics platform in ada: a case study," in *Proc. of the 12th IEEE Real-Time Systems Symposium*, 1991.

[21] S. Punnekkat, "Schedulability analysis for fault tolerant real-time systems," Ph.D. dissertation, York University, UK, 1997.

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

[23] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Real-time scheduling: the deadline-monotonic approach," in *Proc. of IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[24] H. Aysan, R. Dobrin, S. Punnekkat, and R. Johansson, "Probabilistic schedulability guarantees for dependable real-time systems under error bursts," in *Proc. of the 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability*. New York: Freeman, 1979.

APPENDIX

A. Proof of Theorem 1

Theorem 1: The problem of determining whether a set of real-time jobs are Δ-FB-feasible (the problem FB-feasibility) is NP-Hard, even when the fault burst is known to occur in the exact interval $[t_a, t_a + \Delta]$ a priori.

Proof: We will reduce the the well-known NP-Hard problem Partition [25] to FB-feasibility. The input to the problem is a set of n integers $S = \{s_1, \ldots, s_n\}$ where $\sum_{i=1}^{n} s_i = 2X$. The problem is to determine whether there exist 2 disjoint subsets of S such that the sum of integers in each set is equal to exactly X. The union of two sets together must include all the elements in S.

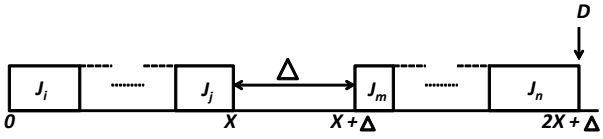

Fig. 17. Reducing Partition to FB-feasibility

We can reduce any given instance of the Partition problem to an instance of the FB-feasibility problem as follows. We construct a set of n jobs. The worst-case execution time of the i^{th}th job, C_i, is set equal to s_i . The release times of all the jobs are equal to 0. Similarly, all the deadlines are the same and equal $2X + \Delta$, where Δ is the length of the fault burst in the FB-feasibility instance. In the FB-feasibility instance, the fault burst is set to occur the interval $[X, X + \Delta)$. Figure 17 shows the position of the fault burst with respect to the common deadline. Jobs can be executed in any order as they have a common deadline. Notice that the selection of job parameters implies that there is a feasible schedule if and only if no job execution partially overlaps with the fault burst and the system does not remain idle at any time before and beyond the fault burst. A partial overlap with the fault burst would require the re-execution of the preempted job at the end of fault burst, for which there is no sufficient time.

Therefore the integers corresponding to the jobs executed prior to the fault burst will form the first subset and the rest will form the second subset in the Partition instance. Thus the outcome of the FB-feasibility instance will determine the outcome of the Partition instance, completing the proof. □

B. Proof of Theorem 2

Theorem 2: A job set Φ is Δ-FB-feasible, only if there exists an algorithm *A* such that for every $(t_a, t_b = t_a + \Delta)$ fault burst interval and for every (t_1, t_2) time interval, the following holds for the generated schedule $S(A, t_a, t_b)$:

$$\sum_{J_k \in \gamma(t_1,t_2)} C_k + O_{\gamma(t_1,t_2)}(t_x, t_y) + (t_y - t_x) \leq (t_2 - t_1) \quad (11)$$

where, $[t_x, t_y)$ denotes the time interval where the fault burst overlaps with the (t_1, t_2) interval under consideration, namely, $t_x = \max\{t_1, t_a\}$ and $t_y = \min\{t_b, t_2\}$

Proof: Suppose, Φ is Δ-FB-feasible. Then, there exists an algorithm *A* that generates a feasible schedule for any fault burst up to length Δ.

Assume that the fault burst occurs at $[t_a, t_b)$ and *A* generates the feasible schedule $S(A, t_a, t_b)$. We now show that, for any arbitrary $[t_1, t_2)$, the claim holds for $S(A, t_a, t_b)$. We first consider the case where the interval $[t_1, t_2)$ does not overlap with the fault burst. Then the overhead due to fault burst will be 0, and the feasibility condition becomes,

$$\sum_{J_k \in \gamma(t_1,t_2)} C_k \leq t_2 - t_1$$

which is trivially true for any feasible schedule S, as we must execute all jobs in $\gamma(t_1, t_2)$ before their deadlines.

Now, assume that the interval $[t_1, t_2)$ overlaps with the fault burst. Then $[t_x, t_y)$ denotes the time interval when the system is under fault burst in interval $[t_1, t_2)$. Job executions that partially overlap with the fault burst, as well as jobs that remain in preempted state, are unreliable as well. In schedule $S(A, t_a, t_b)$ if the execution of a job in $\gamma(t_1, t_2)$ is affected by the fault burst during $[t_x, t_y)$, the execution time consumed by such jobs outside the $[t_x, t_y)$ is given by $O_{\gamma(t_1,t_2)}(t_x, t_y)$. Therefore the effective computation time available during $[t_1, t_2)$ is,

$$(t_2 - t_1) - (t_y - t_x) - O_{\gamma(t_1,t_2)}(t_x, t_y) \quad (12)$$

Hence, the following must hold:

$$\sum_{J_k \in \gamma(t_1,t_2)} C_k \leq (t_2 - t_1) - (t_y - t_x) - O_{\gamma(t_1,t_2)}(t_x, t_y)$$

By simple algebraic manipulation we obtain

$$\sum_{J_k \in \gamma(t_1,t_2)} C_k + O_{\gamma(t_1,t_2)}(t_x, t_y) + (t_y - t_x) \leq (t_2 - t_1) \quad (13)$$

Therefore theorem 2 holds for any arbitrary interval $[t_1, t_2)$ in $S(A, t_a, t_b)$. □

C. Proof of Theorem 3

Theorem 3: For any arbitrary scheduling algorithm *A* with multiple recovery semantics, $\Delta-$idling minimizes the worst-case overhead, $WCO(t_x)$ when a fault burst is detected at time t_x, among all algorithms that use *A* up to t_x.

Proof: Suppose a fault burst is detected at time t_x at the completion of J_i. For any schedule $S(A)$ generated by algorithm *A*,

$$WCO_A(t_x) = \max_{r_i-\Delta \le t_a \le t_x} \{O^{S(A)}(t_a, t_a+\Delta)\}$$

We can rewrite $WCO_A(t_x)$ as follows,

$$WCO_A(t_x) = \max\{\max_{r_i-\Delta \le t_a < t_x-\Delta} \{O^{S(A)}(t_a, t_a+\Delta)\}, \max_{t_x-\Delta \le t_a \le t_x} \{O^{S(A)}(t_a, t_a+\Delta)\}\} \quad (14)$$

Equation (14) represents the worst case overhead as the maximum overhead among the maximum overhead of two intervals. The first interval considers all the scenarios when the fault burst completes prior to t_x. The second interval considers the cases when the fault burst includes t_x. We will consider each interval separately.

Case 1: $r_i - \Delta \le t_a < t_x - \Delta$. The fault burst has ended before t_x. Therefore, during the fault burst both *A* and *A*-Δ has the same schedule. Therefore, the overhead will be the same.

$$\forall_{r_i-\Delta \le t_a < t_x-\Delta}\ O^{S(A)}(t_a, t_a+\Delta) = O^{S(A-\Delta)}(t_a, t_a+\Delta)$$

Case 2: $t_x - \Delta \le t_a \le t_x$. The fault burst includes t_x and possibly spans beyond t_x. Note that, the set of jobs in preempted state will be the same for all recovery strategies. Due to the idle period, *A*-Δ will not execute any job during the fault burst after t_x, while *A* may execute some additional job. Therefore, $X^{S(A-\Delta)}(t_a, t_a+\Delta) \subseteq X^{S(A)}(t_a, t_a+\Delta)$ and for any job $J_i \in X^{S(A-\Delta)}(t_a, t_a+\Delta)$, J_i will not execute any time after t_x. Therefore, $\alpha_i^{S(A-\Delta)}(t_a+\Delta, d_i) = 0$. So,

$$\forall_{t_x-\Delta \le t_a \le t_x}\ O^{S(A)}(t_a, t_a+\Delta) \ge O^{S(A-\Delta)}(t_a, t_a+\Delta)$$

From case 1 and 2 we can conclude that,

$$WCO_{A-\Delta}(t_x) \le WCO_A(t_x)$$

for any scheduling algorithm based on *A* up to fault detection point t_x and any arbitrary occurrence of the fault burst. □

D. Proof of Theorem 4

Theorem 4: A frame-based system scheduled by the chain-Δ algorithm is Δ-FB-feasible, if and only if,

$$\Sigma_i C_i + C_{max} \le P - \Delta \quad (15)$$

Proof: If part: Assume that Equation (15) holds. Now, we construct a feasible schedule for the task set using the chain-Δ algorithm. All tasks are executed according to the pre-specified order until a fault is detected. The chain-Δ algorithm will idle for Δ units and then re-execute faulty task. All other tasks will be executed only once. As a result, the total execution time needed is bounded by $(\Sigma C_i + C_{max})$ which is no larger than the available execution time $P - \Delta$. Therefore, chain-Δ will be able to feasibly schedule the task set for any fault burst of length at most Δ.

Only if part: Since all tasks are released at the same time and has a common deadline, the necessary condition according to Theorem 2 can be written as,

$$\Sigma C_i + O^{S(chain-\Delta)}(t_x, t_x+\Delta) + \Delta \le P$$

For chain-Δ algorithm, the maximum overhead occurs when the task with maximum execution time encounters the fault burst. Further, there are no preemptions and the system deliberately avoids augmenting the overhead by remaining idle after the fault detection. So the worst-case overhead is C_{max}, giving:

$$\Sigma C_i + C_{max} \le P - \Delta$$

Consequently, Equation (15) is the necessary and sufficient condition for chain-Δ algorithm. □

E. Proof of Theorem 6

Theorem 6: For a periodic task set scheduled by the EDF-Δ policy, $U_b^{\Delta} = \frac{1}{2}(1 - \frac{\Delta}{P_{min}})$.

Proof: We use a proof by contradiction. Suppose, there exists a periodic task set with $U_{tot} \le \frac{1}{2}(1 - \frac{\Delta}{P_{min}})$ for which EDF-Δ misses a deadline, when subject to a fault burst of length at most Δ.

Consider the first deadline miss at time d_i, when a job $J_{i,j}$ of task T_i misses deadline. We can identify the earliest job release point $r_a \le r_i$ such that the system is continuously executing jobs with deadline equal to d_i or earlier. Note that in this interval EDF-Δ can idle only for at most Δ units upon the detection of a fault (if there is a fault burst in this interval). At all other times, it should be executing jobs with deadline d_i or earlier according to EDF priority. Hence, the sum of required computations (jobs with deadlines $\le d_i$) during this interval should be larger than the available computation time. The available CPU time is at least $(d_i - r_a) - \Delta$.

Note that, in EDF-Δ, the re-execution starts only after the delay. By that time, the fault burst is guaranteed to end. As a result no re-execution will fail. So, EDF-Δ executes each job at most twice. During interval $[d_i, r_a)$ each task T_k may have at most $\left\lfloor \frac{d_i - r_a}{P_k} \right\rfloor$ instances. Thus, the total computational demand is bounded by,

$$2 \times \sum_{k=1}^{n} \left\lfloor \frac{d_i - r_a}{P_k} \right\rfloor C_k \quad (16)$$

Since $J_{i,j}$ misses deadline, the following must hold,

$$2 \times \sum_{k=1}^{n} \left\lfloor \frac{d_i - r_a}{P_k} \right\rfloor C_k > (d_i - r_a) - \Delta \quad (17)$$

We can perform algebraic manipulation as follows.

$$\Rightarrow 2 \times \sum_{k=1}^{n} \frac{d_i - r_a}{P_k} C_k > (d_i - r_a) - \Delta \Rightarrow 2 \times \sum_{k=1}^{n} \frac{C_k}{P_k} > 1 - \frac{\Delta}{d_i - r_a}$$

$$\Rightarrow 2 \times U_{tot} > 1 - \frac{\Delta}{d_i - r_a}$$

Since $r_a \le r_i$, $d_i - r_a \ge P_{min}$. Thus we obtain,

$$\Rightarrow U_{tot} > \frac{1}{2}(1 - \frac{\Delta}{P_{min}}) \quad (18)$$

Equation (18) contradicts with our assumption. Therefore, the assumption is not valid. So, a feasible schedule exists for any task set with total utilization not exceeding $\frac{1}{2}(1 - \frac{\Delta}{P_{min}})$. By definition of the Δ-sensitive utilization bound, we find $U_b^{\Delta} = \frac{1}{2}(1 - \frac{\Delta}{P_{min}})$ for the EDF-Δ policy. □