Applied Computing and Informatics (2017) **13**, 130–139

Saudi Computer Society, King Saud University

Applied Computing and Informatics

(http://computer.org.sa)
www.ksu.edu.sa
www.sciencedirect.com

ORIGINAL ARTICLE

Timing analysis for embedded systems using non-preemptive EDF scheduling under bounded error arrivals

Michael Short

School of Science and Engineering, Teesside University, Middlesbrough TS1 3BA, UK

Received 8 June 2016; revised 22 July 2016; accepted 28 July 2016
Available online 4 August 2016

KEYWORDS

Real-time and embedded systems;
Timing analysis;
Non-preemptive scheduling;
Transient error protection;
Fault-tolerance

Abstract Embedded systems consist of one or more processing units which are completely encapsulated by the devices under their control, and they often have stringent timing constraints associated with their functional specification. Previous research has considered the performance of different types of task scheduling algorithm and developed associated timing analysis techniques for such systems. Although preemptive scheduling techniques have traditionally been favored, rapid increases in processor speeds combined with improved insights into the behavior of non-preemptive scheduling techniques have seen an increased interest in their use for real-time applications such as multimedia, automation and control. However when non-preemptive scheduling techniques are employed there is a potential lack of error confinement should any timing errors occur in individual software tasks. In this paper, the focus is upon adding fault tolerance in systems using non-preemptive deadline-driven scheduling. Schedulability conditions are derived for fault-tolerant periodic and sporadic task sets experiencing bounded error arrivals under non-preemptive deadline scheduling. A timing analysis algorithm is presented based upon these conditions and its run-time properties are studied. Computational experiments show it to be highly efficient in terms of run-time complexity and competitive ratio when compared to previous approaches.

1. Introduction

1.1. Motivation

Recent results quantifying the optimality gap between non-preemptive scheduling and its preemptive counterpart on uniprocessors have been very encouraging. Using processor speedup analysis as a quantification metric, it has been shown that any task set which can be successfully scheduled by a fully

E-mail address: m.short@tees.ac.uk
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.aci.2016.07.001

preemptive optimal scheduling algorithm on a uniprocessor can also be scheduled by its non-idling, non-preemptive counterpart if the processor is speeded up by a factor which is no more than a simple linear function of the timing requirements (and is usually quite small for realistic applications) [1,2]. Since timing requirements are typically fixed by application constraints and processor speeds are much faster than one or two decades ago, this has renewed interest in non-preemptive and limited-preemptive scheduling on low cost microcontroller and microprocessor platforms [1–5]. For embedded real-time applications such as multimedia, automation and control, increases in hardware parallelism (such as multicore processors, DMA, programmable ADCs/DACs and dedicated communication controllers) also favor the use of non-preemptive scheduling as the workload on the host CPU can be considerably reduced. Such applications are the focus of this paper.

There are several reasons to implement an application using a non-preemptive scheduler if it is possible to do so: these include an easier code implementation (no need to implement context switching and advanced stack management), lower CPU and memory overheads, exclusive access to shared resources by design, fewer cache/pipeline-related flushing events and potentially less susceptibility to transient errors [6–10]. Indeed, the principal advantage of preemptive software architectures is their comparative flexibility around timing requirements [6], which as mentioned has recently been explicitly quantified [1]. Among the available options for non-preemptive scheduling, the non-preemptive Earliest Deadline First (npEDF) dynamic priority-driven scheduling algorithm is known to be optimal among the class of non-preemptive schedulers that do not allow the use of inserted idle-time [8,11,12] (For recent developments in the area of priority-driven non-preemptive scheduling allowing the use of inserted idle-time, the interested reader is referred to Nasri and Kargahi [4] and Nasri and Fohler [5]. The results demonstrated by Abugchem et al. [1] and Thekkilakattil1 et al. [2] are directly applicable to the npEDF scheduler. Since very efficient run-time implementations (with effectively $O(1)$ CPU overheads) are relatively straightforward to create [13], npEDF becomes the focus of the current paper.

1.2. Problem statement

Non-trivial problems can arise with npEDF with respect to task failures and overloads, due to its single-tasking mode of operation. Such failures occur in situations when assumptions of finite and known worst-case execution times are violated and a task overrun its determined execution time, overloading the CPU. Although overloads are also problematic in preemptive architectures, Allworth [14] has noted:

"[The] main drawback with this [non-preemptive] approach is that while the current process is running, the system is not responsive to changes in the environment".

Unless evasive action is taken when a task fails, program flow will never be returned to the scheduler; the entire system will effectively 'freeze' indefinitely, or at least until the problem is cleared. Such a situation is depicted in Fig. 1 for two periodic tasks τ_1 and τ_2, both having a period and relative deadline of 50 ms. Supposing that task τ_1 fails during its third invocation (at $t = 105$), then both tasks will miss their deadlines indefinitely; or at least until the situation is detected and recovered.

Timeline breaks such as that depicted above are especially problematic in real-time control systems. Physical processes such as chemical reactors, aircraft and nuclear power plant are open-loop unstable, and the automatic control of such systems is safety-critical in nature as interruptions of the provided control service may lead to hazardous physical conditions [15]. Even with open loop stable systems, interruptions of control services can lead to deleterious or degraded performance because of the effect of introducing large time delays and jitters in the feedback loop [16]. Considering that experimental studies employing fault-injection have reported that approximately 58% of faults injected into a representative real-time operating system and its application tasks resulted in either complete system failures or multiple task 'crashes' [17], adding fault tolerance into npEDF would seem to be advisory. Fault-tolerance in this context refers to *fault detection and recovery* and also *temporal redundancy*, i.e., indirectly detecting a fault (caused by an error) through its effect on the execution time of a task, recovering the state of the CPU and attempting re-execution of the failed task at some appropriate future point in time.

1.3. Contributions and structure

In this paper, schedulability conditions to ensure that a set of fault-tolerant tasks (which can be periodic or sporadic, or both) under npEDF scheduling will meet their deadlines given some mild assumptions about the nature of errors and their arrival rate will be developed. The efficiency of fault-tolerant npEDF when compared to other fault-tolerant non-preemptive scheduling methods will also be investigated. The specific contributions include the following: (i) the development of efficient schedulability testing conditions for fault tolerant task sets under bounded error arrival rates and (ii) results

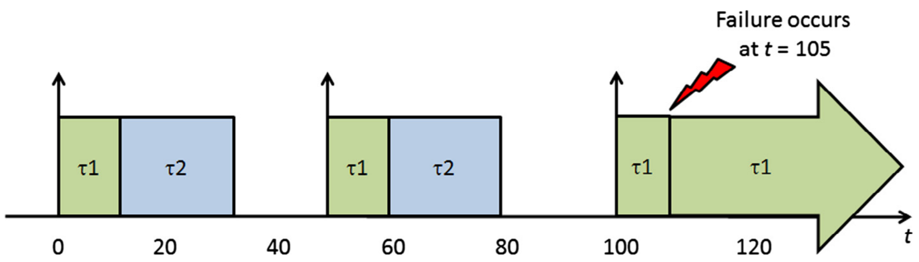

Figure 1 Task failure leading to a schedule ('timeline') break.

from a computational study which provide an evaluation of the relative effectiveness of the proposed techniques in comparison with previous works. The remainder of this paper is organized as follows. Section 2 reviews some relevant previous work in the area of fault tolerant real-time scheduling. Section 3 presents the system and error models while Section 4 derives the schedulability analysis. Section 5 presents the computational simulation results. Section 6 concludes the paper and presents some areas for future work.

2. Related work

A watchdog timer is arguably the simplest approach for error detection and correction in a computer system [18,19]. However the use of a watchdog can have several drawbacks with respect to a real-time control system, as outlined in Short and Sheikh [24]. Previous work has therefore considered more elaborate error/fault detection and recovery means for task failures for a variety of non-preemptively scheduled systems: the key related works are now elaborated.

Mosse and Melham [20], consider a real-time scheduler which is capable of recovering from task failures in an *aperiodic* task environment. The authors consider npEDF along with a 'suitable error detection' capability, such that failed tasks can be detected at or before the end of their allotted execution times. They propose an optimal scheduling algorithm (SFS) with time complexity that is quadratic in the number of active jobs to reserve idle (backup) slots in the aperiodic schedule at least every Δf time units to execute backup tasks should a fault occur. The authors also propose a linear-time heuristic (LTH) to reduce the overheads, and demonstrate that the degree of sub-optimality is small. However, for periodic (c. f. aperiodic) tasks, deciding schedulability becomes intractable due to strong NP-hardness [21]. Sporadic tasks cannot be supported directly in the scheduler. Broster and Burns [22] consider situations in which messages can fail due to electromagnetic interference in a Controller Area Network (CAN). A CAN supports fixed-priority non-preemptive scheduling of messages in a broadcast environment. The authors propose a technique to upper-bound the latest time that a particular message can be scheduled for retransmission using an estimate of message worst-case response times. For npEDF, response times require considerably more effort for their computation and no equivalent to the method has yet been proposed. Hughes and Pont have previously described a simple task guardian to be used with the TTC periodic task scheduler [23]. TTC is a compact form of 'cyclic executive' (see [9] or [14] for further details). Although the technique of Hughes and Pont [23] gives a basic form of error/fault tolerance, the underlying scheduler itself does not support sporadic tasks. For periodic tasks, schedulability testing involves a check across the least common multiple of the task periods that the timeline is free of overloads. This procedure is again strongly NP-hard; Short [10] proposed a re-design of the TTC approach which allows for an improved form of FIFO-based TTC scheduler. This reduces the schedulability test to pseudo-polynomial complexity [10]. Exploiting the predictability and lack of interference in the non-preemptive schedule, a simple means to detect task failures and execute a backup task in an npEDF scheduler was proposed by Short and Sheikh [24]. Although the technique was successfully applied to control an unstable system in the presence of errors, a major drawback in this work is that worst-case non-preemptive blocking was significantly increased; this in turn had a very negative effect on task schedulability and a loss of achievable CPU utilization.

Overload detection and recovery has also been investigated with respect to fully preemptive scheduling. As outlined by the survey paper of Gardner and Liu [25], these techniques typically rely upon the detection of a task exceeding its execution budget and the application of corrective action. An effective solution for preemptive EDF lies in the Constant Bandwidth Server (CBS) [26], which for the case of real-time control systems typically involves the dynamic elongation of the period(s) of the offending task(s) to maintain a constant utilization. The CBS, however, cannot deal with timeline breaks such as that depicted in Fig. 1. Although each of these schemes may be applied (to a certain extent) within the framework of npEDF, to the knowledge of the current author no schedulability conditions for the general case of fault-tolerant npEDF scheduling of recurring tasks under bounded error arrivals have previously been described. This is aimed to be addressed in the next section.

3. System and error model

3.1. System model

Let the single processor task system be represented by a set T of n recurring (periodic/sporadic) tasks, denoted as $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task is parameterized by a 3-tuple:

$$\tau_i = (p_i, c_i, d_i) \tag{1}$$

In which p_i is the task period (minimum inter-arrival time), c_i is the (worst-case) computation requirement of the task, d_i is the task (relative) deadline. Each invocation of a task is called a *job*. When a job of task i becomes ready at time t, its absolute deadline is set at time $t + d_i$ and the scheduling algorithm must allocate c_i units of CPU time to process the job in the interval $[t, t + d_i)$; otherwise, a deadline miss will occur. Jobs are scheduled according to the npEDF scheduling policy, in which the ready job with the earliest (absolute) deadline is selected to be run to completion. Ties between ready jobs sharing a common deadline are broken arbitrarily but consistently, typically by lowest task index. An executing job is said to be *blocking* another job, if this ready job has an earlier deadline and is waiting for the current job to complete before it can execute on the processor; note that in a fully preemptive system, the blocking task would have been preempted (suspended, paused) by the arrival of the blocked job. Under the npEDF scheduling policy, a job can be blocked if and only if it is invoked subsequently to the blocking job being selected for execution.

The utilization of an individual task is given by $u_i = c_i/p_i$, and the utilization of the task set $U = \Sigma u_i$. Successive job arrivals from sporadic tasks are always separated by *at least* p_i units of time; job arrivals from periodic tasks are always separated by *exactly* p_i time units. In this paper no specific relationships between task periods and deadlines are assumed: for any task i the deadline may be constrained ($d_i < p_i$), implicit ($d_i = p_i$) or unconstrained ($d_i > p_i$). A task set is considered to be schedulable if and only if every valid job sequence that could be generated at run-time meets all deadlines [11]. In order to facilitate schedulability analysis, two further functions related to the task set parameters will be defined as follows:

$$h(t) = \sum_{i=1}^{n} \left\lfloor \frac{t + p_i - d_i}{p_i} \right\rfloor \cdot c_i \tag{2}$$

$$b(t) = \max_{d_j > t} \{c_j - 1\} \tag{3}$$

The quantity $h(t)$ is the processor demand function, representing the worst-case execution requirement of jobs with release time and deadline in the interval $[0, t)$. The quantity $b(t)$ is the blocking function, representing the worst-case blocking due to non-preemption that a job may experience in the interval $[0, t)$. See, for example, Refs. [1,11,12] for details of the derivations of these functions.

3.2. Error model

In the current context, a job error can be defined as an incorrect state arising during the computation of a job (arising due to electromagnetic inference or other disturbances); a job fault is the inability of a job to function in a correct manner (due to the manifestation of an error or a software defect); and a job failure is a loss of service where the required computational results cannot be delivered on-time due to faults. The paper is concerned with temporal effects of errors and defects upon system timing. The terms job error, job fault and job failure will be used interchangeably on occasion; it should be taken that this refers to a job which has exceeded its assumed worst-case execution time. In this paper, it is assumed that job failures occur intermittently due to transient errors (EMI) and/or particle strikes. Any transient upset affecting the CPU (and its constituent components or peripheral devices) can potentially lead to control-flow or data errors occurring, which leads to job faults and failures – these job-level faults and failures will ultimately lead to timing failures (deadline misses). Previous estimates of bit corruption probability in an IC arrange from $\approx 10^{-9}$ to 10^{-7} bit failures per hour, varying upon altitude [27]. Other types of failure may also occur intermittently (pseudo-randomly) due to interactions between sensor signals and other inputs and latent software defects, see for example [19,24]. The arrival of errors (or the encountering of software defects) leading to timing faults and failures of jobs is therefore considered sporadic in nature, in the sense that the occurrence of two consecutive job failures is always separated by at least p_f time units. In other words, suppose that the last job failure affecting the system occurred at time t_1 (with probability λ), then the probability of a job failure occurring at some time $t_2 > t_1$ is either λ if $(t_2 - t_1) \geqslant p_f$ and zero otherwise. This assumption of pseudo-periodic error arrivals leading to job failures has been used in several previous works (e.g. [20,22]), and is thought to give a good approximation of reality.

In the analysis that follows, the following assumptions are made regarding the nature of errors, job faults and failures and the resulting behavior of the scheduler: (i) the arrival/occurrence of an error or fault during the execution of a job will cause it to fail; (ii) failures are detectable at or before the end of a job's execution (the job will generate a run-time exception or subsequently try to exceed its determined worst-case execution time); (iii) a period of lost computation time equal to c_f time units is experienced after every job failure (e.g. the time taken to execution of a fault handler or recovery mechanism) and (iv) a failed job is not *immediately* re-executed but is simply re-queued for execution using its original deadline. Assumption (ii) in the above can be enforced by employing suitable run-time error detection techniques (e.g. [19,24]). However, unlike the method proposed in Short and Sheikh [24], assumption (iv) requires that control be returned to the npEDF scheduler upon completion of a fault handler. This is very beneficial from the perspective of schedulability as will be described in the next Section. The schedulability guarantees that will be developed ensure that all jobs generated by all tasks will meet their deadlines in an environment in which error arrivals are repetitive (sporadic), but are bounded in the sense that they are separated by at least p_f time units. In the absence of any empirical data or specific knowledge regarding the error separation p_f, one may look to practical guidelines that have been developed within industry. For industrial measurement and process control systems, Electromagnetic Compatibility (EMC) testing according to IEC 61000-4-4 requires a system to be able to tolerate short bursts of interference (of duration $\approx$15 ms) with a repetition period of 300 ms. Therefore, setting $p_f = 300$ ms and $c_f = 15$ ms (in addition to the time actually required for any additional fault recovery mechanisms) seems a practical choice for most types of industrial embedded systems.

4. npEDF schedulability analysis of fault-tolerant task sets

4.1. Preliminaries

To include the effects of sporadic task failures in the schedulability analysis, it must be assumed that error load (along with processor demand and blocking) will manifest simultaneously in the worst-case manner. To begin, the worst-case sporadic fault manifestation pattern with respect to npEDF scheduling will be categorized. Although intuitively it may seem that the worst-case sporadic fault manifestation is the same as for sporadic tasks (the first job failure occurs at $t = 0$ and subsequent failures arrive with minimum separation), this does not seem to be the case as the following example will illustrate. Consider two implicit-deadline tasks $\tau_1 = \{11, 3\}$ and $\tau_2 = \{5, 2\}$ with $p_f = 20$ and $c_f = 0$. The tasks are schedulable under npEDF when no job failures occur, as the worst-case blocking that τ_1 can induce upon τ_2 is (3–1) = 2, leaving enough slack for τ_2 to meet its deadline. Consider the situation in which an error occurs at $t = 0$, causing τ_1 to fail (indicated by 'F') as shown in Fig. 2 (top). When the job failure is detected at $t = 2$, a scheduling decision is made and although τ_1 is re-scheduled for execution, this execution is suspended until a later time (indicated by 'R' in the figure) as τ_2 has become active and has the earlier deadline; τ_2 still meets its deadline. Suppose now that the occurrence of the error is delayed until $t = 2$, causing τ_2 to fail as shown in Fig. 2 (bottom). In this case, when the failure is detected at $t = 4$, a scheduling decision is made and although τ_2 is immediately re-executed, it now misses its deadline at $t = 5$.

Observation 1. Under npEDF, considering a job with a deadline at $t = d$, then the worst-case arrival of a single error leading to a job failure must be delayed until at least $t = b(d)$

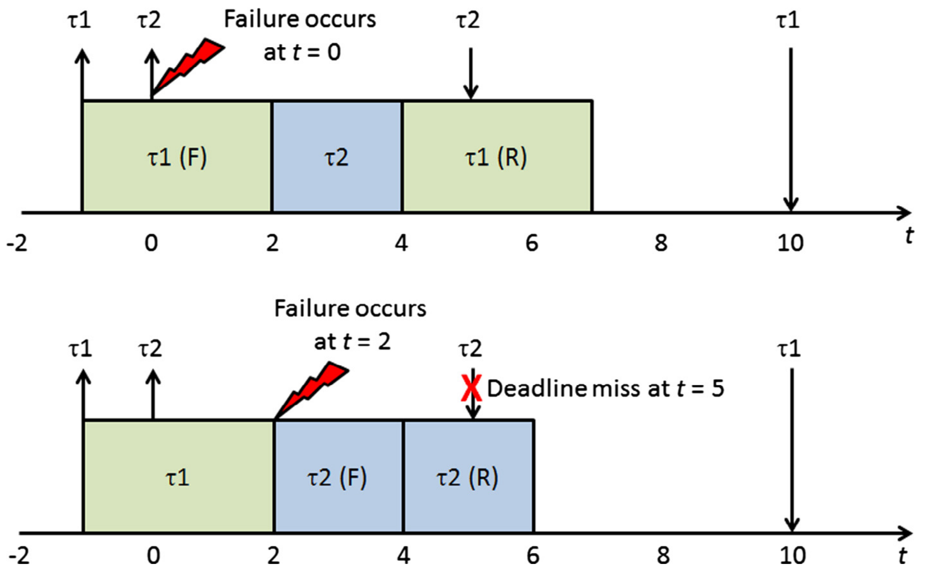

Figure 2 Illustration that a fault arrival occurring during blocking situations is not necessarily the worst-case behavior.

as a failed job with deadline $> d$ will not be eligible for re-execution under the npEDF scheduling policy until all jobs with deadlines at or before d have been successfully executed. □

Observation 2. Under npEDF, considering a job with a deadline at $t = d$ and a single error occurs in the interval $[b(d), d)$, then the worst-case behavior is induced when the eligible job with the largest execution time fails. Under the npEDF scheduling policy, jobs which are eligible for execution in the interval $[b(d), d)$ are those which are generated by tasks satisfying the relationship $d_i \leqslant d$. □

Extending this analysis to the case in which multiple job failures may occur in a given interval is not a trivial matter, since there are situations in which the effect of multiple job failures (with each job having a short execution time) may not necessarily be as severe as the manifestation of only a single job failure (having a comparatively longer job execution time). In addition, there are situations in which the absence of blocking due to non-preemption can lead to worse failure behaviors that with blocking present; such a case may occur if more failures can be packed into the interval $[0, d)$ than $[b(d), d)$ and $b(d)$ is less than the largest execution time of the eligible jobs in $[b(d), d)$. Another complicating factor is that if a failure is detected immediately, then this may have the effect of inserting a preemption point in the schedule, which may reduce run-time blocking (although the worst-case blocking is clearly not effected). As such, although the exact categorization of the worst-case behavior could potentially be obtained, this may require consideration of an excessive number of situations and would be very difficult to compute. However, it is relatively easy to use the observations above to calculate a safe upper bound on the worst-case failure workload, which is presented in the following Lemma:

Lemma 1. *Following a synchronous arrival pattern of tasks at* $t = 0$, *for npEDF scheduling an upper bound on the cumulative worst-case workload due to failed job executions in an interval* $[0, t)$ *in an environment with a minimum error inter-arrival rate of* p_f *is given by the function* $f(t)$ *defined below*:

$$f(t) = \left\lceil \frac{t}{p_f} \right\rceil \cdot \left(c_f + \max_{d_j \leqslant t} \{c_j\} \right) \tag{4}$$

Proof. As the minimum fault inter-arrival rate is p_f, the largest number of errors that may occur in an interval $[0, t)$ is given by the smallest integer that is greater than or equal to the quantity t/p_f. If blocking due to non-preemption is present, re-execution of jobs before t does not need to be considered for errors affecting jobs with deadline $> t$, and since it holds that:

$$\left\lceil \frac{t - b(t)}{p_f} \right\rceil \leqslant \left\lceil \frac{t}{p_f} \right\rceil \tag{5}$$

We also have that the number of assumed error arrivals is always greater than or equal to the actual number of error arrivals, regardless of whether blocking is present or not. As it was assumed that an error arrival will cause the executing job to fail, and since any task with a relative deadline at or before t generates jobs which are eligible for execution in the interval $[0, t)$, taking the largest execution time among those tasks satisfying the relationship $d_i \leqslant t$ gives an upper bound on the execution time of the largest valid job executed; hence adding this value to the worst-case execution time of the fault handler c_f gives the worst-case computational demand due to any *single* error arrival. The function $f(t)$ in Eq. (4) multiplies this upper bound on the number of error arrivals (and hence job failures) by an upper bound on the worst-case computational demand of any single job failure; it must therefore upper bound the cumulative computational demand due to failed job executions regardless of the presence or absence of blocking or the actual arrival pattern of any valid set of errors. □

As with processor demand and blocking, the actual run-time load induced by faults may be much lower than that obtained by the function $f(t)$ since it represents a worst-case value. Building upon this Lemma, it is also possible to make one further observation that will prove useful.

Observation 3. Defining $c_{\max} = \max\{c_i\} + c_f$, the following inequality holds as an upper bound on the value of $f(t)$ $\forall t \geqslant 0$:

$$f(t) \leqslant c_{\max} + t \cdot \frac{c_{\max}}{p_f} \tag{6}$$

$$= c_{\max} + t \cdot u'_f \tag{7}$$

with $u'_f = c_{\max}/p_f$ being the utilization of an equivalent sporadic task to capture the worst-case effects of job failures on the CPU over its lifetime. □

4.2. Schedulability conditions

It is now possible to incorporate the effects of task failures into a sufficient schedulability analysis in a relatively straightforward way, which is summarized in the Theorem and Corollary below:

Theorem 1. *A set of fault-tolerant tasks scheduled using npEDF in an environment with a minimum fault inter-arrival rate p_f and fault handler execution time c_f is schedulable if*:

$$U' < 1.0 \tag{8}$$

And:

$$\forall t,\ d_{\min} \leqslant t < t_{\max}: \quad h(t) + b(t) + f(t) \leqslant t \tag{9}$$

where $d_{\min}$ is the smallest relative deadline among the tasks, $h(t)$, $b(t)$ and $f(t)$ are as given by (2)–(4), $U' = U + u'_f$ and the bound $t_{\max}$ is given by

$$t_{\max} = \max\left\{\max_{1\leqslant i\leqslant n}\{d_i - p_i\}, \frac{\sum_{i=1}^{n} u_i(p_i - d_i) + 2c_{\max} - c_f}{(1 - U')}\right\} \tag{10}$$

with $c_{\max} = \max\{c_i\} + c_f$.

Proof. Lemma 1 has established the validity of the upper bound $f(t)$, and it is straightforward to see that whether the summation of the processor demand $h(t)$, non-preemptive blocking $b(t)$ and the upper bound on workload due to failed job executions $f(t)$ is always less than or equal to the CPU time available then the tasks will be schedulable. Condition (8) is sufficient to ensure that the CPU is not overloaded during the course of its lifetime (in the limit as $t \to \infty$), as we have from (7) that job failures will in effect manifest as a sporadic task requiring a fraction of the CPU utilization proportional to no more than u_f. Thus it remains to verify that no deadlines will be missed in some initial portion of the schedule under worst-case assumptions, and to show that the test interval for this can be bounded by the proposed value of $t_{\max}$ to complete the proof. We have that for $t \geqslant \max\{d_i - p_i\}$, it holds that $t \geqslant d_i - p_i$ $\forall i$ and hence $t - d_i \geqslant -p_i$ $\forall i$. Inspecting the individual terms of the processor demand function (Eq. (2)) for each task we have that

$$\left\lfloor \frac{p_i + t - d_i}{p_i} \right\rfloor > 0 \to \max\left\{0, \left\lfloor \frac{t + p_i - d_i}{p_i} \right\rfloor \cdot c_i\right\} = \left\lfloor \frac{p_i + t - d_i}{p_i} \right\rfloor \cdot c_i \leqslant \frac{p_i + t - d_i}{p_i} \cdot c_i \tag{11}$$

Thus $\forall t \geqslant \max\{d_i - p_i\}$, since $c_{\max} = \max\{c_i\} + c_f$ and using Eq. (7) the following relationship can be written:

$$\begin{aligned} h(t) + b(t) + f(t) &= \sum_{i=1}^{n} \left\lfloor \frac{p_i + t - d_i}{p_i} \right\rfloor \cdot c_i + \max_{d_j > t}\{c_j - 1\} \\ &\quad + \left\lceil \frac{t}{p_f} \right\rceil \cdot \left(c_f + \max_{d_j \leqslant t}\{c_j\}\right) \\ &\leqslant \sum_{i=1}^{n} \frac{p_i + t - d_i}{p_i} \cdot c_i + c_{\max} - c_f + c_{\max} \\ &\quad + t \cdot u'_f \end{aligned} \tag{12}$$

When $U' < 1.0$ and some deadline(s) will be missed, there will exist some t such that $t < h(t) + b(t) + f(t)$ corresponding to a deadline miss and we can write the following:

$$\begin{aligned} t &< h(t) + b(t) + f(t) \\ &\leqslant \sum_{i=1}^{n} \frac{p_i + t - d_i}{p_i} \cdot c_i + 2c_{\max} - c_f + t \cdot u'_f \end{aligned} \tag{13}$$

Simplifying and solving for t:

$$\begin{aligned} t &< \sum_{i=1}^{n} \frac{c_i}{p_i} t + \sum_{i=1}^{n} \frac{c_i}{p_i}(p_i - d_i) + 2c_{\max} - c_f + t \cdot u'_f \\ t &< t\left(U + u'_f\right) + \sum_{i=1}^{n} u_i(p_i - d_i) + 2c_{\max} - c_f \\ t(1 - U') &< \sum_{i=1}^{n} u_i(p_i - d_i) + 2c_{\max} - c_f \\ \therefore t &< \frac{\sum_{i=1}^{n} u_i(p_i - d_i) + 2c_{\max} - c_f}{(1 - U')} \end{aligned} \tag{14}$$

And since condition (8) requires that $U' < 1.0$ the denominator in Eq. (14) is non-zero. Thus if there exists some job deadline at which $t < h(t) + b(t) + f(t)$, this will occur before the larger of the value $\max\{d_i - p_i\}$ and the value for t given in (14), which yields the desired bound of $t_{\max}$ as originally stated in the Theorem. □

Condition (9) needs to be evaluated only at values of t corresponding to absolute job deadlines within the interval $[d_{\min}, t_{\max})$ [11]. If the CPU utilization U' is bounded to be less than some small fixed constant c, the worst-case complexity of evaluating the conditions of Theorem 1 is pseudo-polynomial with run-time $O(n \max\{p_i - d_i\})$, which follows from Theorem 3.1 in Stankovic et al. [11]. In the case where U' is exactly equal to unity, then the complexity of deciding schedulability can become exponential as the condition of (9) is required to be checked over the least common multiple (*lcm*) of the task periods/inter-arrivals. Since for many real-world task sets the CPU utilization can be bounded below the value of some constant close to unity (e.g. $c = 0.999$), the schedulability test can be made to be very efficient. In fact, for implicit deadline task sets the categorization of the run-time complexity can be further improved as will now be shown:

Corollary 1. If *the CPU utilization U' is bounded to be less than some small fixed constant c and all tasks have deadlines equal to their periods, the worst-case complexity of an algorithm to evaluate the conditions of* Theorem 1 *is $O(n^2)$.*

Proof. When $d_i = p_i$ $\forall i$, evaluation of $t_{\max}$ reduces to

$$t_{\max} = \max\left\{\max_{1\leqslant i\leqslant n}\{d_i - p_i\}, \frac{\sum_{i=1}^{n} u_i(p_i - d_i) + 2c_{\max} - c_f}{(1-U')}\right\}$$
$$= \max\left\{0, \frac{2c_{\max} - c_f}{(1-U')}\right\} = \frac{2c_{\max} - c_f}{(1-U')} \quad (15)$$

And since $U' \leqslant c < 1.0$ for some fixed constant c we can further write

$$t_{\max} \leqslant \frac{2c_{\max} - c_f}{1-c} \leqslant \frac{2c_{\max}}{1-c} \quad (16)$$

Now, since $h(d_{\min}) + b(d_{\min}) + f(d_{\min}) \geqslant c_{\max}$, the condition $c_{\max} \leqslant d_{\min} = p_{\min}$ is necessary for schedulability (and can easily be checked in $O(n)$ steps), and any task set failing this condition can immediately be flagged as unschedulable; hence, this quantity becomes a lower bound on the smallest period of any task in a schedulable task set. Thus, the maximum number of absolute deadlines d' for the jobs generated by n tasks in the interval $[0, t_{\max})$ is upper bounded by

$$d' \leqslant \frac{nt_{\max}}{c_{\max}} \leqslant \frac{2n}{1-c} \quad (17)$$

For fixed c, no more than $O(n)$ deadlines need to be checked, and as the evaluation of the functions $h(t)$, $b(t)$ and $f(t)$ takes time linear in n for each of these deadlines, the overall procedure requires not more than $O(n^2)$ iterations. □

Even in cases of relatively high CPU utilization the bound on the number of absolute deadlines given by Eq. (17) is useful. For example when $c = 0.999$, not more than $2000n$ deadlines ever need checking. This is in contrast to methods proposed by Mosse and Melham [20] and Hughes and Pont [23], in which evaluation of schedulability surmounts to checking over the task hyper-period. The following example illustrates this efficiency gain.

4.3. Illustrative example

To illustrate the efficiency of the proposed schedulability analysis with respect to previous work, an illustrative example is given below.

Example 1. Consider a task system consisting of three implicit-deadline periodic tasks with parameters $\tau_1 = \{11, 2\}$, $\tau_2 = \{15, 3\}$ and $\tau_3 = \{40, 4\}$ to be scheduled with npEDF operating in an environment with $p_f = 12$ and (for ease of exposition) $c_f = 0$.

Using the technique proposed by Mosse and Melham [20], it would have to be determined whether the equivalent aperiodic jobs generated by the periodic tasks over the system lifetime can be accommodated, by executing the LTH algorithm (with npEDF as the underlying rule for ordering the priority queue) for each and every new job occurring in L which represents the planning cycle (hyper-period) of the schedule. The length of L corresponds to the least common multiple of the task periods, which in this case is equal to 1320 and therefore ≈240 applications of LTH are required to verify schedulability of the tasks. Using the basic TTC scheduler technique with 'task guardians' [23] requires a number of checks exactly equal to the length of L (1320) to carry out a schedulability analysis. However, note that the tasks are not schedulable using the basic TTC technique without fault-tolerance as $gcd(11, 15, 40) = 1 < \max\{2, 3, 4\} = 4$.

Alternatively, using the schedulability analysis technique developed in this Section, it is first determined that $c_{\max} = 4$ and compute $U = 0.482$ and $u_f = 0.333$, and hence $U' = 0.815$. This allows the computation of $t_{\max} = 43.28$ and therefore each absolute task deadline in the interval $[1, 43]$ requires checking to verify schedulability. This requires evaluating the functions $h(t)$, $b(t)$ and $f(t)$ for values of t equal to 11, 15, 22, 30, 33 and 40. These values are tabulated in Table 1 below.

From this table it may be observed that the combined CPU load $h(t) + b(t) + f(t)$ is always less than the time available for its processing across the required test interval; therefore, schedulability is verified after checking only 6 absolute deadlines. □

5. Computational study

Recall that the motivation for the current paper was to develop an efficient mechanism to add fault-tolerance to npEDF with efficiently verifiable schedulability conditions. The analysis and illustrative example in the previous Section gave a promising indication of the suitability of the proposed method in this respect; computational experiments using randomly generated representative task sets will be used to give further evidence in this respect. Specifically, the competitive ratio and analysis complexity will be investigated in more depth. First, the methodology employed to generate the task parameters and the design of the experiment is described.

5.1. Task parameter generation/experiment design

Individual task sets were randomly generated for the experiments, and in each case the number of tasks n was varied between 5 and 30 in multiples of 5. For each value of n, the total CPU utilization U' was varied between 0.6 and 0.999 in multiples of 0.1 (or 0.099 for the last step), and for each combination of U' and n, the fault utilization u_f' was varied between 0.1 and 0.3 in steps of 0.1. Next, the individual utilization allowed for each task u_i was generated (without bias) using the UniFast algorithm [28], such that the CPU utilization for all tasks $U = U' - u_f'$. The individual task periods/inter-arrivals p_i were then uniformly selected from the interval $[10, 1000]$ in multiples of 10, and computation times c_i were computed using $c_i = p_i \cdot u_i$. Next, relative deadlines d_i were uniformly drawn from the interval $[0.7p_i, 1.3p_i]$. The minimum fault inter-arrival was computed according to $p_f = c_{\max}/u_f$.

Table 1 Illustration of the schedulability test.

t	$h(t)$	$b(t)$	$f(t)$	$h(t) + b(t) + f(t)$
11	2	3	2	7
15	5	3	6	14
22	7	3	6	16
30	10	3	9	22
33	12	3	9	24
40	16	0	16	32

Next, task sets were generated using this procedure and then filtered until 10^5 task sets satisfying conditions (8) and (9) had been obtained for each combination of n, U' and u_f', giving a total of 900,000 task sets covering a wide range of periods, relative deadlines, utilizations, computation times and fault arrival rates.

For each of the generated task sets, a measure of the competitive ratio of the proposed technique was obtained by applying a modified TTC schedulability test (described in Short [10]) to each set of tasks; note that the test was modified to include a fault bounding function (conceptually similar to (5)) to reflect that a failed task is immediately re-inserted into the head of the FIFO queue when using the 'task guardian' technique. Note that to maximize the chance of schedulability, 'tick overruns' were assumed allowed in this case for reasons discussed in Short [10]. Results are also reported for the measure of competitive ratio of the proposed technique with the technique described by Mosse and Melham [20] using the LTH algorithm with npEDF, with the caveat that the results presented may not be full accurate, due to the complexity of the analysis; in most cases the CPU running time was prohibitive. Instead, this measure was accurately estimated based on a limited range of examples with tractable running time. To obtain further insight into algorithm complexity as compared to other methods, the ratio of the length of the testing interval ($t_{\max}$) given by (10) with the duration of the synchronous FIFO busy period, which defines the number of iterations required for the test in Short [10], and also with the *lcm* of the task periods (as this defines the complexity of the test proposed in Mosse and Melham [20]) is also measured. These data are reported as average case values, as the worst-case value was always <1 and the best case effectively was 0.

Table 2 Competitive ratios and relative testing complexity.

Competitive ratio		Test interval length	
TTC	LTH	TTC	LTH
20.06	100.00	84.49	0.043

5.2. Results and observations

The results that were obtained from the experiment are as shown in Table 2. In the table, ratios are expressed as percentages. From these data it can be observed that only 20.06% of the task sets deemed schedulable by the proposed technique were found to be schedulable using the modified TTC scheduler. This provides further quantifiable evidence of the degree of sub-optimality of the TTC approach as compared with the proposed approach: less than a quarter of the task sets could be successfully scheduled. No task sets were schedulable with the modified TTC scheduler and not npEDF, which is to be expected given the optimality of the latter. Among those tasks which could be tested by LTH in a reasonable time, none were found that were not able to be scheduled; given that npEDF was used as the underlying scheduling heuristic, this was also not surprising. When comparing the test interval lengths, it can be observed that an 84.49% reduction in the number of deadline checks was required for proposed approach using npEDF than with the modified TTC. A more revealing figure is the ratio obtained for the LTH; on average, less than 0.043% of the deadline checks required by LTH were needed by the proposed approach, which is a significant improvement. Note that since the length of the test interval in the original TTC approach as described in Pont and Hughes (2008) is of identical length to LTH, this also gives an indication of the improvement in performance over this method.

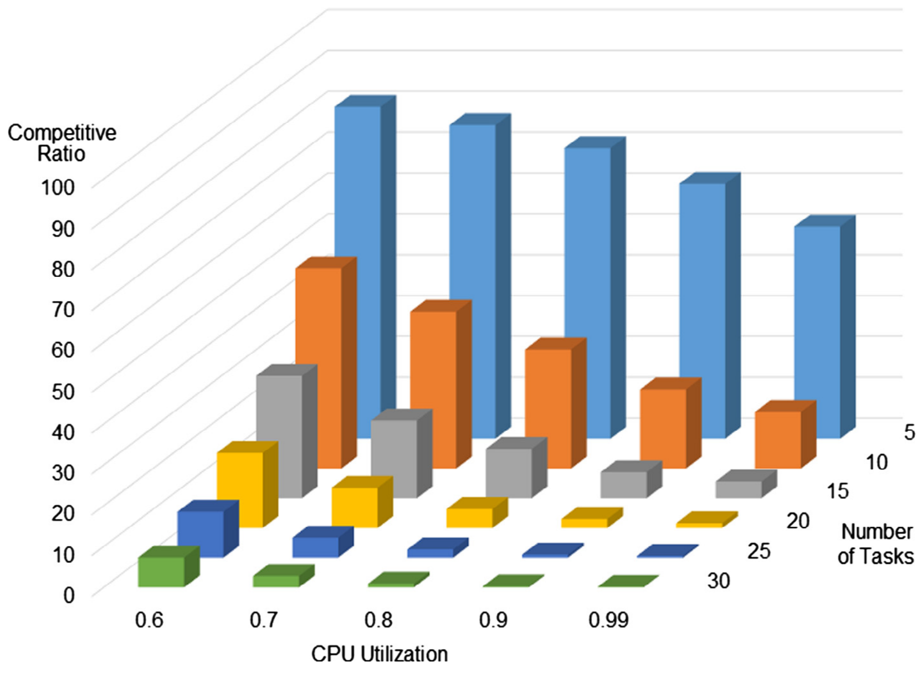

Figure 3 Competitive ratio as a function of both the number of tasks and CPU utilization.

In order to further investigate the sub-optimality of the modified TTC approach, an investigation was made of the parameters that influence the achieved competitive ratio. The competitive ratio was calculated (as a percentage) for each configuration of number of tasks and CPU utilization U'. Fig. 3 shows the obtained ratios for these two indices.

From the figure it is apparent that the competitive ratio varies considerably with changes to these parameters. The competitive ratio approaches 90% when 5 tasks are present with utilization $U' = 0.6$, but decreases to 50% in a close to linear fashion as the CPU utilization increases. Increasing the number of tasks for a given CPU utilization leads to a dramatic (exponential) drop in the competitive ratio. For 20 or more tasks and utilization of 0.8 or greater, the competitive ratio remained below the 5% level. These data give additional supporting evidence that the goal of creating a simple and flexible approach for adding fault tolerance to npEDF has been achieved, in that the technique has a good competitive scheduling ratio and comparatively low analysis complexity.

6. Conclusions and further work

Non-preemptive scheduling techniques can provide a simple and attractive option for meeting real-time constraints in embedded systems. In this paper, the fault-tolerant npEDF scheduling of periodic and sporadic tasks has been studied. Schedulability analysis techniques for task sets experiencing bounded sporadic error arrivals have been developed, and have been shown to provide an efficiency improvement over previous methods in terms of competitive ratio and/or analysis complexity. In conclusion, the proposed technique may be of interest to developers of fault-tolerant, non-preemptive embedded systems which may be exposed to interference and errors. Further work will concentrate upon better categorizations of the fault load, the application of probabilistic schedulability guarantees and extensions to multiprocessor environments, extending techniques such as those proposed in Andrei et al. [29].

Acknowledgment

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors. The author declares no conflict of interests regarding the publication of this article.

References

[1] F. Abugchem, M. Short, D. Xu, On the sub-optimality of non-preemptive real-time scheduling, IEEE Embed. Syst. Lett. 7 (3) (2015) 69–72, ISSN 1943-0663.
[2] A. Thekkilakattil1, R. Dobrin, S. Punnekkat, The limited-preemptive feasibility of real-time tasks on uniprocessors, Real-Time Syst. 51 (2015) 247–273.
[3] G.C. Buttazzo, M. Bertogna, G. Yao, Limited preemptive scheduling for real-time systems: a survey, IEEE Trans. Industr. Inf. 9 (1) (2013) 3–15.
[4] M. Nasri, M. Kargahi, Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks, Real-Time Syst. 50 (4) (2014) 548–584.
[5] M. Nasri, G. Fohler, Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions, in: Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS), Tolouse, France, July, 2016.
[6] G.C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Springer-Verlag, New York, 2005.
[7] M. Short, M.J. Pont, J. Fang, Exploring the impact of pre-emption on dependability in time-triggered embedded systems: a pilot study, in: Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS 2008), Prague, Czech Republic, 2008, pp. 83–91.
[8] K. Jeffay, D. Stanat, C. Martel, On non-preemptive scheduling of periodic and sporadic tasks, in: Proceedings of the IEEE Real-Time Systems Symposium, 1991, pp. 129–139.
[9] M.J. Pont, Patterns for Time Triggered Embedded Systems, Addison Wesley, 2001.
[10] M. Short, Analysis and redesign of the 'TTC' and 'TTH' schedulers, J. Syst. Architect. 58 (1) (2012) 38–47.
[11] J.A. Stankovic, M. Spuri, K. Ramamritham, G.C. Buttazzo, Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms, Kluwer Academic Publishing, 1998.
[12] L. George, N. Rivierre, M. Supri, Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling Research Report RR-2966, INRIA, Le Chesnay Cedex, France, 1996.
[13] M. Short, Improved task management techniques for enforcing EDF scheduling on recurring task sets, in: Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), Stockholm, Sweden, 2010, pp. 56–65.
[14] S. Allworth, Introduction to Real-time Software Design, Springer-Verlag, 1991.
[15] K.G. Shin, H. Kim, Derivation and application of hard deadlines for real time control systems, IEEE Trans. Syst. Man Cybern. 22 (1992) 1403–1413.
[16] C.R. Elks, J.B. Dugan, B.W. Johnson, Reliability analysis of hard real-time systems in the presence of controller faults, in: Proceedings of the Annual IEEE Reliability and Maintainability Symposium, 2000, pp. 58–64.
[17] N. Ignat, Y. Nicolescu, G. Savaria, G. Nicolescu, Soft-error classification and impact analysis on real-time operation systems, Proceedings of the Design Automation & Test in Europe Conference (DATE), vol. 1, 2006, pp. 47–52.
[18] M.J. Pont, R.H.L. Ong, Using watchdog timers to improve the reliability of single-processor embedded systems: seven new patterns and a case study, in: Proceedings of the First Nordic Conference on Pattern Languages of Programs, Otaniemi, Finland, 2003.
[19] M. Short, Development guidelines for dependable real-time embedded systems, in: Proceedings of the 6th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2008), Doha, Qatar, 2008, pp. 1032–1039. April.
[20] D. Mosse, R. Melham, A nonpreemptive real-time scheduler with recovery from transient faults and its implementation, IEEE Trans. Softw. Eng. 29 (8) (2003) 752–767.
[21] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman & Co Ltd, 1979.
[22] I. Broster, A. Burns, Timely use of the CAN protocol in critical hard real-time systems with faults, in: Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS), Delft, The Netherlands, 2001.
[23] Z. Hughes, M.J. Pont, Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed, Trans. Inst. Meas. Control 30 (5) (2008) 427–450.
[24] M. Short, I. Sheikh, Timely recovery from task failures in non-preemptive, deadline–driven schedulers, in: Proceedings of the

7th IEEE International Conference on Embedded Software and Systems (ICESS 2010), Bradford, UK, pp. 1856–1863.

[25] M.K. Gardner, J.W.S. Liu, Performance of algorithms for scheduling real-time systems with overrun and overload, in: Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS), York, UK, 1999.

[26] M. Caccamo, G.C. Buttazzo, L. Sha, Handling execution overruns in hard real-time control systems, IEEE Trans. Comput. 51 (7) (2002) 835–849.

[27] E. Normand, Single event effects in avionics, IEEE Trans. Nucl. Sci. 43 (2) (1996) 461–474.

[28] E. Bini, G.C. Buttazzo, Measuring the performance of schedulability tests, Real-Time Syst. 30 (2005) 127–152.

[29] S. Andrei, A. Cheng, V. Radulescu, An improved upper-bound algorithm for non-preemptive task scheduling, in: Proceedings of 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, September 21–24, IEEE Computer Society, Timisoara, Romania, 2015.