Real-Time Syst (2017) 53:45–81
DOI 10.1007/s11241-016-9258-z

Real-time scheduling algorithm for safety-critical systems on faulty multicore environments

Risat Mahmud Pathan[1]

Published online: 20 September 2016

Abstract An algorithm (called FTM) for scheduling of real-time sporadic tasks on a multicore platform is proposed. Each task has a deadline by which it must complete its non-erroneous execution. The FTM algorithm executes backups in order to recover from errors caused by non-permanent and permanent hardware faults. The worst-case schedulability analysis of FTM algorithm is presented considering an application-level error model, which is *independent* of the stochastic behavior of the underlying hardware-level fault model. Then, the stochastic behavior of hardware-level fault model is plugged in to the analysis to derive the probability of meeting all the deadlines. Such probabilistic guarantee is the level of assurance (i.e., reliability) regarding the correct functional and timing behaviors of the system. One of the salient features of FTM algorithm is that it executes some backups in active redundancy to exploit the parallel multicore architecture while other backups passively to avoid unnecessary execution of too many active backups. This paper also proposes a scheme to determine for each task the number of backups that should run in active redundancy in order to increase the probability of meeting all the deadlines. The effectiveness of the proposed approach is demonstrated using an example application.

Keywords Real-time systems · Fault-tolerant systems · Global multiprocessor scheduling · Schedulability tests · Probabilistic analysis

✉ Risat Mahmud Pathan
risat@chalmers.se

[1] Department of Computer Science and Engineering, Chalmers University of Technology, 412-96 Göteborg, Sweden

1 Introduction

The demand for more functions and comfort features in today's prevailing computerized systems is increasing. The types and varieties of different functions or services determine the competitiveness of computerized systems—e.g., portable devices, cars, aircrafts—in the market. A modern passenger car, now-a-days equipped with dozens of processors, does not only provide functions related to vehicle control but also supports services related to comfort and safety. Moreover, such safety-critical systems also have certain important design constraints, for example, real-time and fault-tolerant constraints. Integrating more functions while satisfying such design constraints requires more computation power. Luckily, contemporary multicore processor provides such computation power.

Multicore processors are and will be the main enabler to meet the growing demand of computing power for many embedded and safety-critical systems, e.g., in the automotive and aerospace domains. Two of the main advantages of multicore processor are higher computation power and the ability to execute programs in parallel. These benefits of multicore (unlike uniprocessor) result in shorter response time for time-critical application by allowing independent parts of the application to execute in parallel.

This paper considers safety-critical applications (e.g., control and monitoring) that are modeled as a collection of sporadic tasks with *hard* real-time constraints, i.e., computation of each task must be completed by some pre-specified deadline. In addition, the correct output of such tasks has to be guaranteed. In order to avoid catastrophic consequences (e.g., injury or death), one of the major challenges during the design of safety-critical real-time systems is to guarantee that correct output is generated before deadline even in the presence of faults. The design and analysis of a multicore scheduling algorithm to meet the hard deadlines of an application while making the algorithm robust with fault-tolerance capability is the focus of the research presented in this paper.

Each sporadic task potentially releases infinite number of instances, called *jobs*, such that consecutive jobs are separated by a minimum inter-arrival time (often called, the *period* of the task). Each task τ_i has a relative deadline D_i which is less than or equal to its period (i.e., *constrained-deadline* sporadic task model is considered). Tasks are assumed to be independent, i.e., the only resource they share is the CPU time. However, the proposed approach of this paper can be extended for other shared resources, for example, based on abort-and-restart model of computation (Wong and Burns 2014; Ras and Cheng 2010). The *real-time constraint* is that if a job of task τ_i released at time r, then the job must generate its output before its deadline which is at time $(r + D_i)$. The *fault-tolerant constraint* is that the correct output of each job has to be generated by recovering from any possible occurrence of error during execution.

According to Avižienis et al. (2004), a *fault* is a source of an *error* which is an incorrect state in the system that may cause deviation from correct service, called *failure*. When some core of a multicore platform ceases functioning (i.e., fails), such faulty behavior is characterized as a *core failure*. By a core failure, we mean that one core of the multicore chip is faulty; not that the entire chip is damaged. When all the cores fail, then the entire chip is considered to be permanently faulty. On the other hand, when the behavior of a job of a task is incorrect (e.g., wrong output or wrong

path) while the core executing that job is non-faulty, such behavior is characterized as a *job error*, not as a job failure, because the aim of this paper is to mask job errors to avoid job failures.

This paper proposes the design and the analysis of a Fault-Tolerant Multicore (FTM) scheduling algorithm based on time-redundant execution of backups to mask job errors and tolerate core failures. The job errors and core failures are assumed to occur due to *non-permanent* and *permanent* hardware faults, respectively. By non-permanent hardware faults we mean temporary malfunctioning of computing unit, for example, due to hardware transient faults. By *tolerating* core failures, we do not mean that a faulty core becomes functional again or repaired; rather we mean that the job of the task that was executing on the faulty core still meets its deadline by completing execution on some other (non-faulty) core.

The main contribution of this work is the derivation of closed-form expression that can be applied offline (i.e., before the system is put in mission) to compute *probabilistic schedulability guarantee* of an application hosted on a multicore platform. Such probabilistic guarantee is the level of assurance regarding the correct functional and timing behaviors of the system. Safety-critical (e.g., automotive) systems require such offline guarantee. If the level of assurance (computed in terms of probabilistic schedulability guarantee in this paper) is low with respect to some standard, then the design of the system is considered to be unsafe and the system may need to be redesigned. This paper presents a complete methodology to evaluate such level of assurance of safety-critical system regarding its ability to generate timely and correct output.

Fault-Tolerant Real-Time Scheduling There are two main approaches to real-time scheduling on multicores: *partitioned* and *global* approach (Carpenter et al. 2004). In partitioned scheduling, each task of an application is allocated to some core during design time and its jobs are allowed to execute *only* on that core at run-time (i.e., a job of a task cannot migrate to another core). In global scheduling, a job of a task is allowed to execute on *any* core even when it is resumed after preemption (i.e., a task can migrate to another core). The strict non-migratory characteristic of partitioned multiprocessor scheduling is relaxed in so called *semi-partitioned* scheduling in which some tasks are allowed to migrate to a different processor (Andersson et al. 2008; Kato and Yamasaki 2009; Lakshmanan et al. 2009; Pathan and Jonsson 2010a). A recent survey on (non-fault-tolerant) multiprocessors real-time scheduling can be found in Davis and Burns (2011).

In fault-tolerant real-time scheduling, each task is considered to have one primary and one or more backups that apply to its jobs. There are several works on fault-tolerant scheduling based on partitioned and global approaches for multiprocessors (Oh and Son 1994; Bertossi et al. 1999; Hashimoto et al. 2000; Chen et al. 2007; Kim et al. 2010; Berten et al. 2006; Girault et al. 2003; Liberato et al. 1999; Pathan and Jonsson 2011a; Huang et al. 2011). In partitioned fault-tolerant scheduling, a task-allocation algorithm assigns the primary and backups of each task to *distinct* processors[1] at design time. In case of a job error or a processor failure is detected, the output of a

[1] The terms "core" and "processor" are used synonymously in this paper.

backup that is non-erroneous and assigned to some other processor is chosen as the output.

One of the limitations of the previously proposed task-allocation algorithms (Oh and Son 1994; Bertossi et al. 1999; Chen et al. 2007; Kim et al. 2010;Girault et al. 2003) is that these algorithms do not make any distinction between job errors and processor failures. Most of the earlier works considered mainly partitioned scheduling to tolerate particularly permanent faults, which (implicitly) can also tolerate transient faults; but at the expense of space redundancy since all backups of a task are allocated to distinct cores. In other words, job errors are masked by pessimistically assuming that the processor on which the corresponding task is assigned has failed. This pessimism requires a relatively higher number of processors to successfully assign all the primary and backups even when *only* job errors are to be masked. Such over provisioning of the computing resources in the partitioned approach is costly for resource- and cost-constraint embedded systems. Moreover, finding an optimal assignment of tasks to processors is NP-hard (Garey and Johnson 1979).

The main motivation for this work is to design a resource-efficient scheduling algorithm to tolerate core failures and propose particular mechanisms to mask job errors. Coming up with a resource-efficient scheduling algorithm to mask job errors is important since job errors are more frequent due to the rising trend of non-permanent hardware faults in computer electronics, as it was pointed out by Baumann (2005), Srinivasan et al. (2004), and Borkar (2005). This paper considers tolerating both non-permanent and permanent faults using global scheduling that does not need any (offline) allocation of backups, which can reduce the number of required cores. In global scheduling, the primary and backups are not statically allocated to any core; rather the primary and the backups are dispatched for execution on any free core even after preemptions. Unlike the abort-and-restart model of computation (Wong and Burns 2014; Ras and Cheng 2010), the preempted tasks continue their execution once higher priority tasks complete execution[2]. The proposed `FTM` algorithm in this paper is designed based on the following two simple strategies applied to global multicore scheduling:

- To mask job errors caused by non-permanent faults, the global scheduling approach during run-time can simply schedule a backup of the faulty job to any core *even* to the core on which the job error was detected. This is because, due to the transient nature of non-permanent faults, tolerating a job error does not restrict a backup to be executed on a fixed (pre-allocated) core.
- To tolerate core failures caused by permanent hardware faults, the global scheduling approach during run-time simply considers that the job which was executing on the faulty core has encountered a job error. In other words, a core failure can be viewed from the migratory scheduler's point of view as a job error. Then, tolerating a core failure is same as tolerating a job error, i.e., the scheduler simply dispatches the backup of the affected job to any *non-faulty* core.

[2] In abort-and-restart model (Wong and Burns 2014; Ras and Cheng 2010), a preempted (lower priority) task is aborted and starts as a new task when higher priority tasks complete execution.

Our proposed algorithm, called FTM, is designed based on global preemptive fixed-priority (FP) scheduling approach to tolerate both job errors and core failures. The detail dispatching policy of FTM algorithm is presented in Sect. 3.

Time Redundancy Achieving fault-tolerance in computer systems requires employing redundancy either in space or time (Koren and Krishna 2007). The use of *time redundancy* to tolerate both job errors and core failures is advocated in this paper considering the size, weight and power constraints in many embedded systems. We do not address the problem of tolerating the failure of an entire multicore chip or system software (e.g., RTOS), which can be addressed using space redundancy, for example, by replicating all the channels using triple modular redundancy (TMR). In this paper, errors are assumed to be detected using some existing hardware- or software-based error detection mechanisms that are already available on the target platform (Meixner et al. 2008; Al-Asaad et al. 1998; Jhumka et al. 2002; Hiller 2000).

The FTM algorithm employs time-redundant execution of multiple backups to tolerate multiple errors. A backup can be the *re-execution* or a *different implementation* of the primary. The term "error" in general in this paper is used to specify both job errors and core failures. Multiple errors, for example, due to burst of non-permanent hardware faults, may affect the same job in such a way that the primary and multiple backups of that job may become erroneous. To deal with such scenarios, FTM considers the use of multiple backups for each task (in particular, for each of its jobs). No single primary/backup can execute on more than one core at any given time instant in FTM scheduling. However, different primaries and backups may execute on different cores in parallel, as is explained in next paragraph.

Each backup of a job is categorized either as an *active* or *passive*. In FTM scheduling, when a job of a task is released, all of its active backups *always* become ready for execution along with its primary. Therefore, the primary and the active backups have the potential to execute in parallel on different cores. In contrast, the passive backups of a job become ready for execution one-by-one if the primary and all the previously-dispatched backups (active or passive) of that job have been detected to be erroneous. If the primary or any of the active backups completes execution without an error being detected, no passive backup is executed. Note that we may abort the execution of active backups as soon as the primary or any of the active backups completes successfully. Such abortion of active backups will definitely avoid unnecessary execution. In many works on mixed-criticality scheduling, low-critical tasks are aborted when some high-critical task risk missing its deadline (Baruah and Fohler 2011; Baruah et al. 2011; Pathan 2014). An interesting future work is to extend this work by applying analysis similar to that of mixed-critical systems to abort active backups. In this paper, we assume that all the active backups are executed till completion (i.e., no active backup is aborted).

An active backup can execute in advance (i.e., pessimistically assuming that errors will be detected later on) even though no error is ultimately detected. Active redundancy may consume more processing resource (hence, energy) but provides better fault-tolerance for low-laxity (shorter deadline) tasks. In contrast, passive redundancy may consume less processing resource but may not provide enough fault-tolerance for the low-laxity tasks. One of the major challenges addressed in this paper is determining how many backups of a task should be active whenever a job of that task is released.

In particular, a heuristic is proposed to determine the number of active backups for each task so that the system has higher likelihood of meeting all the deadlines even in the presence of faults.

Contributions To take advantages of both active and passive redundancy, FTM executes a *fixed* number of backups for any job of a particular task in active redundancy and other backups in passive redundancy. Combining active and passive redundancy in FTM scheduling has the following benefit: multiple active backups and the primary have the potential to execute in parallel on the multicore architecture and executing other backups passively avoids unnecessary execution of too many active backups if too many errors are unlikely.

The exploitation of time to achieve fault tolerance in hard real-time systems must not lead jobs to miss their deadlines. In order to ensure that the deadlines of the jobs of each task are met even in the presence of faults, the worst-case schedulability analysis of FTM algorithm is presented. This analysis is based on a simple but powerful *application-level error model* but independent of the stochastic behavior of any particular *hardware-level fault model*. The outcome of the analysis is that, for each task, the maximum number of errors that each of its job can tolerate is derived. Note that this maximum is same for the all jobs of the same task and could be different for jobs of different tasks since the maximum duration for which jobs of different tasks execute (i.e, remain active) are different.

The "separation" of the stochastic behavior of a particular fault model from the application-level error model has the following advantage. The stochastic behavior of *any* fault model can be plugged in to the analysis later in order to determine the probability of "actual" number of errors during run-time not being larger than the maximum number of tolerable errors for each job. To this end, a closed-form expression to compute the *probabilistic schedulability guarantee*, i.e., all deadlines are met with certain probability, considering the stochastic behavior of *two* particular fault models is derived. Coming up with such probabilistic guarantee during the design and the analysis is important to demonstrate adequate confidence in safety-critical system's correct timing and functional behavior, for example, for system certification.

The two hardware-level fault models considered in his paper are called (i) the $\mathcal{B}$ fault model, and (ii) the $\boldsymbol{\mathcal{R}}$ fault model. The $\mathcal{B}$ model considers that non-permanent hardware faults occur randomly as well as in $\underline{\mathcal{B}}$ursts whereas the $\mathcal{R}$ model considers that such faults occur only $\underline{\mathcal{R}}$andomly. The failure rate of non-permanent faults during a bursty period is much higher than that of during non-bursty period. However, both $\mathcal{B}$ and $\mathcal{R}$ fault models consider that permanent hardware faults occur only randomly. Note that the $\mathcal{R}$ model is a specialization of the $\mathcal{B}$ model with no bursty period. The $\mathcal{B}$ model is same as the fault-burst model recently used by Short and Proenza (2013) for uniprocessor fault-tolerant scheduling. We extend the analysis of this model to apply in the multicore context.

Finally, a heuristic that can be used by the system designer to determine the number of active backups for each task is proposed. Rather than arbitrarily configuring some backups of each task as active and others as passive, this heuristic presents a systematic approach to determine the number of active backups for each task so that the probability of meeting all the deadlines for a given fault model is increased. The effectiveness of the proposed approach in this paper is demonstrated using an example application.

Organization This paper is organized as follows. Section 2 presents the error, fault and application models. Section 3 presents the FTM algorithm. The worst-case schedulability analysis of FTM scheduling algorithm to determine the maximum number of tolerable errors for each task is performed in Sect. 4. Then, the probabilistic analysis of FTM algorithm is presented in Sect. 5 considering the stochastic behavior of fault models. A heuristic to determine the number of active backups for each task is presented in Sect. 6. The results of sensitivity analysis using an example application is proposed in Sect. 7. Related works are presented in Sect. 8 before we conclude in Sect. 9.

2 Error, fault and application model

This paper considers the scheduling of a set of sporadic tasks on a multicore platform consisting of M homogeneous cores. The normalized speed of each core is 1. The error, fault and application model are presented in this section.

Error Model Algorithm FTM considers tolerating both job errors and on-chip core failures. The application-level error model is very general in the sense that errors can be detected in the primary and even in the backups of any job of any task, at any time and in any core. The error model does not put any separation restriction between occurrences of consecutive errors or does not assume any probability distribution of the errors. Considering such a general application-level error model provides us the "power" to keep the stochastic behavior of hardware-level fault model separate during the worst-case schedulability analysis of FTM algorithm. Considering the separation between the error model and fault model is reasonable because the way hardware faults are manifested as application-level errors depends on many factors, like error-detection mechanisms, the schedule of the tasks, the type and frequency of faults, etc.

Errors are assumed to be detected using some existing hardware/software based error-detection mechanisms, for example, built-in error detection capabilities in modern processors (Meixner et al. 2008; Al-Asaad et al. 1998) or executable assertions (Jhumka et al. 2002; Hiller Hiller:2000), etc. The analysis presented in this paper is not changed for undetected errors. Errors that bypass the detection at the multicore chip level need to be tolerated using space redundancy at the system level and is not addressed in this paper.

Fault Model The job errors and core failures are assumed to occur due to hardware faults that are non-permanent and permanent, respectively. The type of non-permanent faults that may cause task not to generate correct output (called, *value failures*) are considered in our fault model. Value failures can be arbitrary, i.e., produces different results at different run. Since we rely on effective error-detection mechanisms, such arbitrary faults can also be tolerated using our proposed scheme whenever the corresponding errors are detected. To tolerate f arbitrary faults, we do not need $3f + 1$ backups as is required in distributed consensus with malicious nodes [known as Byzantine agreement problem Jalote (1994)]. This is because malicious nodes in Byzantine problem are not identifiable by the non-malicious nodes. In contrast, this paper considers that value failures are detected (i.e., identifiable) using some error-detection

mechanism. The type of non-permanent faults that may cause task not to generate output by its deadline [characterized as *timing faults* or *omission faults* Jalote (1994)] are considered in our fault model. Such faults may be detected using watchdog timer and can be tolerated using our proposed fault-tolerant scheduling policy. Since we rely on effective error-detection mechanisms, we can tolerate any type of faults as long as the corresponding error is detected and a non-erroneous execution meets task's deadline.

The errors/failures may be detected during the execution of any task at any time, even during the execution of backups. By a core failure, we mean that one core of the multicore chip is faulty; not that the entire chip is damaged [known as *crash failure* Jalote (1994)]. When all the cores are permanently faulty, then the entire chip is considered to be permanently faulty. A permanent core failure is assumed to be caused by some permanent fault in the hardware, for example, failure of an ALU in some core. We consider *fail-stop* cores: each core is either working correctly or ceases functioning. Many interesting architectural techniques, for example, configurable isolation (Aggarwal et al. 2007), are proposed for fault containment at core level (i.e, one core failure does not cause other cores on the same chip to fail).

A core failure is tolerated in `FTM` algorithm by executing the backup of the affected job on other (non-faulty) core. This paper considers the scenario in which the multicore exhausts the cores over time. Such core-exhaustion will cause the schedulability of the jobs to become worse over time since a relatively less number of non-faulty cores will be available due to such architecture deterioration. To account for such scenario, as will be evident later, the schedulability analysis of the jobs of a task is performed by varying the number of non-faulty cores ranging from `M` to 0.

A job error can be caused by *random* non-permanent faults that are temporary malfunctioning of the computing unit that happens for a short time and then disappear without causing a permanent damage. Hardware transient fault is a well-known example of random non-permanent faults. The main sources of transient faults in hardware are environmental disturbances like power fluctuations, electromagnetic interference and ionization particles. Transient faults are the most common, and their number is continuously increasing due to high complexity, smaller transistor sizes and low operating voltage for computer electronics (Shivakumar et al. 2002; Baumann 2005; Borkar 2005). Job error due to such fault can be masked by re-executing the job because it is expected that the same fault would not reappear. Multiple random non-permanent faults may cause multiple job errors.

Multiple job errors can also be caused by *bursts* of non-permanent faults. Fault burst may occur when the system is exposed to very harsh environment and the rate of faults during a burst period is much higher in comparison to that of rate of faults during non-bursty period. For example, electromagnetic interference (EMI) caused by lighting strikes may cause incorrect computation and bit flips in architectural registers. Other examples are automotive and avionics systems exposed to higher EMI levels when passing airport radar or communication facilities (Many and Doose 2011).

In a study by Ferreira et al. (2004), it is found that 90 % of the errors in the CAN network are due to burst of faults with an average burst length of 5 μs. The length of fault burst is also found to be bounded due to voltage (Joseph et al. 2003) and temperature fluctuations (Sherwood et al. 2003). To that end, fault burst is assumed to be non-permanent since the length of a bursty period is generally bounded but fault

burst may reappear, for example, when the system enters again in the high EMI field. Although burst of faults can be avoided using shielding (e.g., in space shuttles), the cost of shielding each critical hardware could be quite high for relatively low-cost safety-critical systems, for example, automotive. Therefore, cost-constrained safety-critical systems need alternative mechanism to mitigate the affect of burst of faults. This paper exploits the computation power of multicore processor to tolerate bursts of faults using time redundancy. Job errors due to random/burst of non-permanent faults can be masked by executing backups of the corresponding task.

This paper considers stochastic behavior of two different faults models: (i) $\mathcal{B}$ fault model, and (ii) $\mathcal{R}$ fault model.

(i) $\mathcal{B}$ *Model* This model is a derivative of the Gilbert–Elliott's fault-burst model (Elliott 1963) as used by Short and Proenza (2013) in the context of fault-tolerant scheduling. The $\mathcal{B}$ model is represented using five parameters: λ_c, λ_b, λ_r, L_G and L_B having the following interpretations.

The $\mathcal{B}$ model considers that non-permanent faults occur in *bursts* as well as *randomly* during so called bursty and non-bursty states, respectively. The expected (mean) gap between two consecutive fault bursts is L_G and the expected (mean) duration of one fault burst is L_B where both L_G and L_B has geometric distribution. In other words, the expected inter-arrival time of fault bursts is $(L_G + L_B)$.

Permanent hardware faults are assumed to occur only *randomly*. Permanent hardware faults in the multicore occur at constant failure rate λ_c. The non-permanent hardware faults in *each* core occur with constant failure rates λ_b and λ_r during bursty and non-bursty states, respectively. Note that the failure rates of non-permanent faults apply to each core. This is because we consider the situation where some external condition that causes non-permanent faults to occur at a rate λ_b or λ_r may affect all the cores. In other words, fault propagation is considered as follows: if a multicore chip suffers non-permanent faults at a rate λ_b or λ_r, then each core is assumed to suffer non-permanent faults at a rate λ_b or λ_r. This is depicted in Fig. 1.

Note that the stochastic behavior of this fault model does not force any temporal distribution regarding the *actual* occurrences of the faults. The actual faults may occur at any time and there is no assumption regarding the inter-arrival time of consecutive faults. The stochastic behavior of a fault model is used to reason about the actual faults that occur at run-time.

(ii) $\mathcal{R}$ *model* This model is a specialization of the $\mathcal{B}$ model with no fault burst. The $\mathcal{R}$ model considers that both permanent and non-permanent hardware faults occur only *randomly* with constant failure rate λ_c and λ_r, respectively. The detail stochastic behavior of these two fault models will be presented when the probabilistic schedulability analysis of `FTM` algorithm is presented in Sect. 5.

Relationship Between Error and Fault Model The fault model represents the frequency, type and nature of actual faults that affect the hardware, for example, transient faults arriving at a constant rate. The manifestation of faults as job errors or core failures at the application level is captured in the error model. The error model is the application-level view of the underlying hardware-level fault model.

In this paper, the worst-case schedulability analysis of `FTM` algorithm is performed considering the error model and without any concern regarding the stochastic behavior of any particular fault model. For example, if the occurrences of two hardware faults

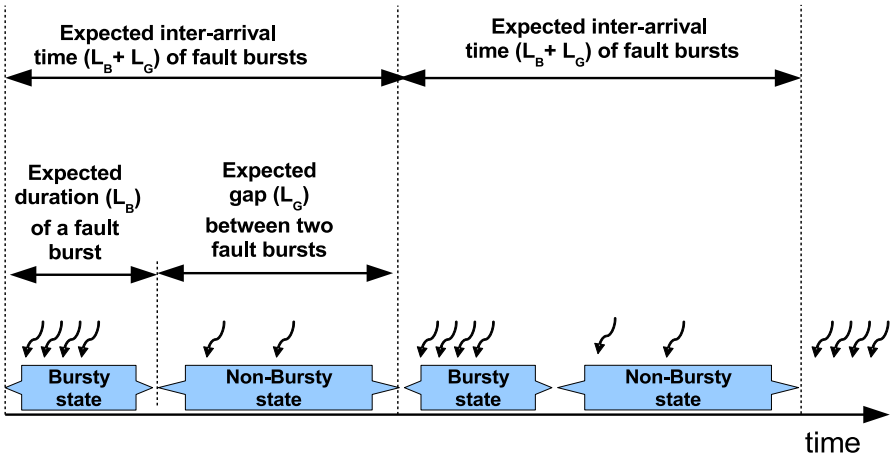

Fig. 1 Stochastic behavior of non-permanent faults where L_B and L_G are respectively the expected (mean) length of bursty and non-bursty states where both L_G and L_B have geometric distribution. The expected (mean) gap between two consecutive bursts is L_G and the expected (mean) duration of a fault burst is L_B. The expected inter-arrival time of fault bursts is $(L_G + L_B)$. Non-permanent faults occur at rate λ_b and λ_r during bursty and non-bursty periods, respectively

are separated by a certain distance, then the detection of the corresponding errors (given that these faults lead to errors) may not be separated by the same distance at the application level. This is because the way hardware faults are manifested as application-level errors depends on many factors, like error-detection mechanisms, error-detection latency, the schedule of the tasks, and so on.

The independence of the error model from the underlying fault model enables us to perform the probabilistic schedulability analysis by plugging in the stochastic behavior of *any* particular fault model to the analysis of the FTM algorithm. As will be evident later that due to such "separation of concern" between the fault and error models, different probabilistic schedulability guarantee for the same application is derived for different fault models.

Application Model This paper considers real-time application modeled as a collection of sporadic tasks. The tasks of the application are scheduled on a multicore platform having M cores. An application is a collection of *n sporadic* tasks in set $\Gamma = \{\tau_1, \ldots \tau_n\}$. Each sporadic task τ_i is characterized by a 4-tuple $< T_i, D_i, \overrightarrow{B_i}, h_i >$ where

- T_i is the *minimum* inter-arrival time of the jobs of task τ_i. The parameter T_i is often called the period of the task.
- D_i is the *relative deadline* of each job of task τ_i such that $D_i \leq T_i$.
- $\overrightarrow{B_i}$ is a vector $< E_i^0, E_i^1, E_i^2 \ldots >$ where E_i^0 is the (WCET) of the primary and E_i^b, for $b \geq 1$, is the WCET of b^{th} backup of τ_i. The size of this vector is finite since each task can execute only a finite number of backups before its deadline.
- h_i is the number of active backups that each job of task τ_i execute in active redundancy. In other words, total h_i backups for each job of task τ_i become ready along with the primary regardless whether an error is detected or not.

The task-level parameter h_i, which represents the number of active backups of task τ_i, is set by the system designer to take advantages of both active and passive backups. Section 6 presents a heuristic to determine the value of h_i for each task τ_i such that the probability of missing any deadline is reduced in comparison to the case where all the backups of each task are passive. Note that different tasks may have different number of active backups. This is because some low-laxity task may not complete the execution of its backups if such backups are configured as passive. Therefore, the number of active backups for such low-laxity tasks may be higher in comparison to the tasks that have relatively higher laxity. For example, some high-laxity task may have all its backups configured as passive backups whereas some other low-laxity task may have all its backups as active backups.

A sporadic task τ_i generates an infinite sequence of jobs with consecutive arrivals separated by at least T_i time units. If a job of task τ_i is released at time r, then this job must complete the execution of its primary and backups before deadline $d = (r + D_i)$. And, the next job of task τ_i is released no earlier than $(r + T_i)$. In this paper, all time values (e.g., WCET, relative deadline, time length) are assumed to be non-negative integer. This is a reasonable assumption since all the events in the system happen only at clock ticks.

The multiple backups of task τ_i are ordered in vector $\overrightarrow{B_i}$ based on system designer's preference, i.e., the output of the primary is preferred over the first backup which in turn is preferred over the second backup, and so on. A backup can be the re-execution or a different implementation of the primary. Therefore, the WCET of a backup may be smaller, greater or equal to the WCET of the corresponding primary.

For each job of task τ_i, the number of backups that execute in active redundancy (configured by the system designer) is h_i. If f errors affect the same job of task τ_i during *run-time*, then at most $max\{h_i, f\}$ backups of that job need to complete execution to mask f errors. We denote C_i^f the sum of WCET of the primary and backups of a job of task τ_i that needs to mask f errors at run-time, and is calculated as follows:

$$C_i^f = E_i^0 + E_i^1 + \ldots + E_i^{max\{h_i, f\}} \tag{1}$$

Equation (1) computes the total workload of the primary and backups of task τ_i when a job of τ_i suffers f errors. Although Eq. (1) is the sum of the primary and $max\{h_i, f\}$ backups of task τ_i, it does not imply that the primary and backups are executed sequentially; the primary and active backups can execute in parallel on different cores. We denote $\widehat{C}_i^f$ the sum of WCET of the *passive* backups of a job of task τ_i that needs to mask f errors at run-time. Since $C_i^{h_i}$ is the total WCET of the primary and h_i active backups according to Eq. (1), $\widehat{C}_i^f$ is calculated as follows:

$$\widehat{C}_i^f = C_i^f - C_i^{h_i} \tag{2}$$

Note that if f is less than h_i, then no passive backup needs to be executed to mask f errors, and therefore, $\widehat{C}_i^f = 0$. Since passive backups are executed one at a time, the execution of $\widehat{C}_i^f$ time units is completed sequentially. Next we present an example application which will be used as the running example throughout this paper.

Table 1 Five tasks of instrument control application

Task's name (τ_i)	E_i^0	E_i^1	E_i^b for $b \geq 2$	D_i	T_i	h_i
Mode management (τ_1)	25	18	25	70	100	1
Mission data management (τ_2)	10	12	10	80	200	0
Instrument monitoring (τ_3)	5	10	5	100	250	1
Instrument configuration (τ_4)	40	42	40	120	200	0
Instrument processing (τ_5)	25	15	25	150	300	1

All values are in milliseconds (ms)

Example 1 Consider a real-time application, called "Instrument Control (IC)", that is responsible for managing the onboard instruments in some safety-critical systems used for some specific mission (for example, the instruments used for collecting and analyzing diagnostic information of the electrical and electronic components in a modern truck or car). Also consider that this application is modeled as a collection of five real-time tasks given in Table 1. The first backup of each task has different WCET than that of the corresponding primary. Also assume that the WCET of bth backup for $b \geq 2$ is equal to that of the primary, i.e., $E_i^b = E_i^0$ for $b \geq 2$.

The number of backups that are executed in active redundancy for each job of task τ_i is h_i (given in the last column of Table 1). For example, each job of *Instrument monitoring* task (τ_3) executes $h_3 = 1$ backup in active redundancy. The WCET of the primary, first and second backups of a job of τ_3 are $E_3^1 = 5$, $E_3^1 = 10$ and $E_3^2 = 5$, respectively.

If a job of task τ_3 is affected during run-time by $f = 2$ errors, then at most $max\{f, h_3\} = 2$ backups of task τ_3 need to execute to mask $f = 2$ errors. The sum of WCET for the primary and two backups of this job to mask 2 errors is $C_i^f = C_3^2 = \sum_{z=0}^{2} E_3^z = 5 + 10 + 5 = 20$ according to Eq. (1). And, the total WCET due to the passive backups (there is only one passive backup) for this job is $\widehat{C}_i^f = \widehat{C}_3^2 = C_3^2 - C_3^1 = 20 - 15 = 5$ using Eq. (2).

The tasks are assumed to be independent in the sense that the only resource they share is the processor platform. The cost of different kinds of overhead, for example, context switch, preemption and migration, error detection are assumed to be included in the WCET of each task. This is because, to the best of our knowledge, at least for now, there is no analytical method available to calculate the cost of such overheads for sporadic task systems considering different processors architecture and operating systems. Although we do not address such issues in this paper, one can rely on experimental studies (similar to Brandenburg et al. 2008) to measure these overhead costs considering the application, operating system and the target hardware platform.

Priority Without loss of generality, a task with lower index is assumed to have higher fixed priority (i.e., τ_1 is the highest and τ_n is the lowest priority task). The set of higher priority tasks of τ_k is denoted as $hp(k)$. The primary and all backups of task $\tau_i \in hp(k)$ have higher priorities than the priority of the primary and all backups of τ_k. And, the priority ordering among the primary and backups of the *same* job of each

task τ_i is based on the ordering of backups in $\overrightarrow{B_i}$, i.e., primary has higher priority over the first backup which in turn has higher priority over the second backup, and so on. Such fixed-priority ordering respects the designer's preference of ordering the primary and backups for each task.

3 Algorithm FTM: tasks dispatching policy

In this section, the tasks dispatching policy for FTM algorithm is presented. We consider that there are total M cores available only for executing the tasks. All jobs that are released but have not yet completed their execution are stored in a common *ready queue* and dispatched by FTM as follows:
① The primary and h_i (active) backups of a job of τ_i are stored in the ready queue whenever a new job of τ_i is released.
② If a core fails, then no job is dispatched on that core.
③ The primaries/backups from the ready queue are dispatched based on preemptive global FP scheduling approach. If some (non-faulty) core is idle, then the highest-priority primary/backup from the ready queue is dispatched for execution on that core. If all the cores are busy executing jobs, then a higher-priority primary/backup in the ready queue is allowed to preempt a currently-executing relatively lower-priority primary/backup. The preempted primary/backup may later resume its execution on any core.
④ If the primary or any backup of a job of task τ_i completes execution without signaling an error, then the corresponding output is selected as the job's output. If the primary and *all* the active backups of a job of task τ_i are erroneous, then one-by-one additional backup of task τ_i (which was passive so far) becomes active/ready.

In summary, the FTM scheduler executes the primary and all the h_i (active) backups of each job of task τ_i using preemptive global FP scheduling policy. If the primary or a backup of task τ_i generates non-erroneous output, then this output is selected as τ_i's output. If the primary and all the backups of τ_i are erroneous, then one-by-one passive backup becomes active until the error is masked.

It is considered that an error is detected at the end of execution of the primary or backup. This is the worst-case scenario for error detection because it corresponds to having a relatively closer job's deadline than that of the case when the error is detected earlier. The worst-case schedulability analysis of FTM algorithm is presented in Sect. 4 to determine for each task the maximum number errors that each of its jobs can tolerate. In Sect. 5, the probabilistic schedulability analysis is presented considering the stochastic behavior of fault models.

4 Worst-case schedulability analysis of FTM algorithm

This section presents the schedulability analysis of FTM algorithm to derive, for each task $\tau_k \in \Gamma$, the maximum number of task errors that any of its job can tolerate between its release time and deadline by assuming that exactly ρ cores fail during run-time. The analysis of this section [particularly Eq. (12) of Theorem 2] is applied to each

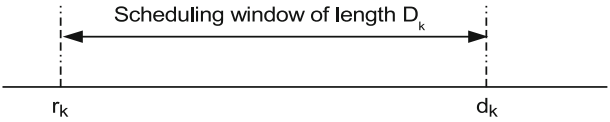

Fig. 2 Scheduling window of an arbitrary job J_k of task τ_k

task to find the maximum number of task errors that any job of the task can tolerate. The parameter ρ captures the effect of core-exhaustion over time due to permanent faults. The value of ρ can range from 0 to $\mathtt{M}$ (where $\rho = 0$ specifies that all the cores are non-faulty and $\rho = \mathtt{M}$ specify that all cores are faulty).

The schedulability condition (i.e., whether all jobs' deadlines are met) for each task τ_k is derived based on the worst-case analysis of an arbitrary job of this task. Consider an arbitrary job, denoted by J_k, of task τ_k where r_k and d_k are the release time and deadline of job J_k such that $d_k = (r_k + D_k)$. The interval $[r_k, d_k)$ is called the *scheduling window* of J_k (as is depicted in Fig. 2). The worst-case schedulability analysis of FTM algorithm in this scheduling window is performed to determine whether job J_k meets its deadline. Without loss of generality, job J_k is considered as a *generic job* of task τ_k in the sense that if J_k meets its deadline, then all other jobs of τ_k also meet their deadlines. It will be evident later from the schedulability analysis that we do not need to know where in the schedule this generic job J_k is released. Since J_k is an arbitrary job, all other jobs of τ_k are guaranteed to meet their deadlines if J_k meets its deadline.

Since FTM algorithm is based on fixed-priority dispatching policy, whether a job J_k meets its deadline or not depends on the FTM schedule of the tasks only in set $hp(k) \cup \{\tau_k\}$ and the number of errors that affect the these tasks in $[r_k, d_k)$. We denote $\mathtt{HJ}_k$ the set of jobs of the higher priority tasks in set $hp(k)$ that are eligible to execute in $[r_k, d_k)$.

We denote je_k the *maximum* number of job errors in $[r_k, d_k)$. We also assume that *exactly* ρ cores fail in $[r_k, d_k)$ where $0 \leq \rho \leq \mathtt{M}$. While ρ specifies the number of faulty cores, the value of je_k specifies the number of maximum job errors in the scheduling window of J_k. We denote $\widehat{\mathtt{M}}$ as the number of non-faulty cores and $\ominus_k$ as the number of errors (task errors and core failures) in $[r_k, d_k)$ where

$$\widehat{\mathtt{M}} = (\mathtt{M} - \rho) \tag{3}$$

$$\ominus_k = (je_k + \rho) \quad \text{if } je_k \geq 0 \quad \text{and} \quad \rho \geq 0 \tag{4}$$

Our aim is to derive the *largest* value of je_k considering ρ number of faulty cores in the scheduling window of J_k. For some given ρ, if J_k cannot be guaranteed to meet its deadline even when there is *no* task error, then we set $je_k = -\infty$ to specify that J_k cannot tolerate ρ number of core failures even when there is no job error[3].

To find the largest value of je_k considering ρ core failures, we have to consider the worst-case that all $(je_k + \rho)$ errors affect the jobs in set $\mathtt{HJ}_k \cup \{J_k\}$ because

[3] The value of je_k is set to $-\infty$ under the assumption that exactly ρ cores are faulty. Although $\ominus_k = -\infty$ when $je_k = -\infty$ according to Eq. (4), it does not mean that J_k cannot tolerate a number of core failures smaller than ρ. Also notice that if $je_k = -\infty$ for a given ρ, then je_k is $-\infty$ for a number of core failures larger than ρ. The schedulability analysis (presented later) never computes $\ominus_k$ by adding je_k and ρ.

algorithm FTM masks both task errors and core failures using time-redundant execution of backups. Whether J_k can meet its deadline depends on the workload of the jobs in $\mathtt{HJ}_k$. The *workload* of the higher priority jobs is the cumulative amount of time during which the jobs of the tasks in set $\mathtt{HJ}_k$ execute in $[r_k, d_k)$.

Finding the exact amount of workload of jobs of sporadic tasks in an interval requires to consider all possible release times of all the tasks, which is computationally infeasible. Instead, an upper bound on the workload of the jobs in set $\mathtt{HJ}_k$ is computed. We denote $\mathtt{W}^c(\mathtt{HJ}_k)$ the upper bound on the actual workload of the jobs in set $\mathtt{HJ}_k$ such that these jobs in set $\mathtt{HJ}_k$ suffer at most c errors in $[r_k, d_k)$. Note that $0 \leq c \leq \in_k$ since the c errors are part of the $\in_k$ errors that can occur in $[r_k, d_k)$.

Organization of This Section The sketch of the worst-case schedulability analysis of FTM algorithm to find the largest je_k for some given ρ, where $0 \leq \rho \leq \mathtt{M}$, is as follows. First, we determine the set $\mathtt{HJ}_k$ in Sect. 4.1. Second, the workload of the jobs in set $\mathtt{HJ}_k$ that suffer at most c errors in $[r_k, d_k)$, i.e., value of $\mathtt{W}^c(\mathtt{HJ}_k)$, is computed in Sect. 4.2. Since there are at most $\in_k$ errors in $[r_k, d_k)$ where c of these errors affect the higher priority jobs, there are at most $(\in_k - c)$ errors that can exclusively affect the primary and backups of job J_k. Based on this observation, a schedulability condition to determine whether job J_k can meet its deadline is derived in Sect. 4.3 based on (i) the value of $\mathtt{W}^c(\mathtt{HJ}_k)$, (ii) the possible parallel execution of the primary and h_k active backups of job J_k on the non-faulty cores, and (iii) the non-parallel execution of the passive backups of J_k. And, the maximum value of je_k for some given ρ is determined based on this schedulability condition (i.e., Eq. 12).

4.1 Finding set $\mathtt{HJ}_k$

Set $\mathtt{HJ}_k$ is computed by finding the maximum number of jobs of each higher priority task $\tau_i \in hp(k)$ that are eligible for execution in $[r_k, d_k)$. The idea to compute $\mathtt{HJ}_k$ is inspired from the schedulability analysis of (non-fault-tolerant) global fixed-priority scheduling for which the critical instant is not known (Pathan and Jonsson 2011b). Not knowing the critical instant is the main difficulty in determining the exact number of high-priority jobs in $[r_k, d_k)$. This problem is avoided by finding an upper bound on the number of higher-priority jobs that are eligible to execute in $[r_k, d_k)$. The maximum number of jobs of task τ_i that are eligible to execute in $[r_k, d_k)$ is determined by considering two cases: Case (i) $D_k < (T_i - D_i)$, and Case (ii) $D_k \geq (T_i - D_i)$.

Case (i) $D_k < (T_i - D_i)$: After the deadline of each higher priority job of τ_i, there is forced to be an interval of length $(T_i - D_i)$ during which no new job of τ_i is released since the minimum inter-arrival time of the jobs of τ_i is T_i. Therefore, there is at most *one* job of τ_i that can execute in $[r_k, d_k)$ for this case because $(d - r) = D_k < (T_i - D_i)$, which implies the length of the scheduling window is smaller than $(T_i - D_i)$. Without loss of generality, assuming that job J_i^1 is the first job of τ_i that is eligible to execute in $[r_k, d_k)$, i.e., $J_i^1 \in \mathtt{HJ}_k$.

Case (ii) $D_k \geq (T_i - D_i)$: For this case, the number of jobs of task τ_i that are eligible to execute in $[r_k, d_k)$ is at most $\lceil \frac{D_k - (T_i - D_i)}{T_i} \rceil + 1$. This is because, after the deadline of the first job J_i^1 that is eligible to execute in $[r_k, d_k)$, there are at most $\lceil \frac{D_k - (T_i - D_i)}{T_i} \rceil$ jobs of task τ_i that can be released (as compactly/early as possible) in

the interval $[r_k, d_k)$. Therefore, there are at most $\lceil \frac{D_k-(T_i-D_i)}{T_i} \rceil + 1$ jobs of task τ_i that can be executed in the interval $[r_k, d_k)$.

From these two cases, the maximum number of jobs of task τ_i, denoted by $N(i)$, that may execute in the interval $[r_k, d_k)$ of length D_k is given as follows:

$$N(i) = \left\lceil \frac{max\{0,\, D_k - (T_i - D_i)\}}{T_i} \right\rceil + 1 \tag{5}$$

Equation (5) is combines both cases as follows. From case (i), when $D_k < (T_i - D_i)$, we have $D_k - (T_i - D_i) < 0$. Therefore, $max\{0,\, D_k - (T_i - D_i)\} = 0$ and $N(i) = \lceil \frac{max\{0, D_k-(T_i-D_i)\}}{T_i} \rceil + 1 = 1$. Remember from case (i) that there is at most one job of task τ_i that can be executed in interval $[r_k, d_k)$. From case (ii), when $D_k \geq (T_i - D_i)$, we have $D_k - (T_i - D_i) \geq 0$. Therefore, $max\{0,\, D_k - (T_i - D_i)\} = D_k - (T_i - D_i)$ and $N(i) = \lceil \frac{max\{0, D_k-(T_i-D_i)\}}{T_i} \rceil + 1 = \lceil \frac{D_k-(T_i-D_i)}{T_i} \rceil + 1$. Remember from case (ii) that there are at most $\lceil \frac{D_k-(T_i-D_i)}{T_i} \rceil + 1$ jobs of task τ_i that can be executed in interval $[r_k, d_k)$.

Set $\mathtt{HJ}_k$ is the collection of $N(i)$ jobs of each task $\tau_i \in hp(k)$. Therefore, the set $\mathtt{HJ}_k$ is given as follows:

$$\mathtt{HJ}_k = \bigcup_{\tau_i \in hp(k)} \{J_i^1, J_i^2, \ldots J_i^{N(i)}\} \tag{6}$$

4.2 Computing workload $\mathtt{W}^c(\mathtt{HJ}_k)$

To compute $\mathtt{W}^c(\mathtt{HJ}_k)$ we have to consider the worst-case occurrences of c errors that affect the jobs of set $\mathtt{HJ}_k$ so that the total CPU time requirement due to these jobs is maximized in $[r_k, d_k)$. Computing the value of $\mathtt{W}^c(\mathtt{HJ}_k)$ exactly requires to consider all possible releases of the jobs of set $\mathtt{HJ}_k$ within the scheduling window of J_k and there are potentially exponential number of possible releases of these jobs in the scheduling window. To overcome this problem, we rely on sufficient analysis as described as follows. The value of $\mathtt{W}^c(\mathtt{HJ}_k)$ is recursively computed using Eqs. (7) and (8). The basis of the recursion is given in Eq. (7), for each $J_i^p \in \mathtt{HJ}_k$, as follows:

$$\mathtt{W}^f(\{J_i^p\}) = C_i^f \tag{7}$$

for $f = 0, 1, \ldots c$ where C_i^f is computed using Eq. (1). The value of $\mathtt{W}^f(\{J_i^p\})$ in Eq. (7) is the workload of each job $J_i^p \in \mathtt{HJ}_k$ such that f errors exclusively affect the job, $0 \leq f \leq c$. By assuming the values of $\mathtt{W}^{(c-f)}(\mathcal{A})$ are known for $(c - f) = 0, 1, \ldots c$, where $\mathcal{A} = (\mathtt{HJ}_k - \{J_i^p\})$ for some $J_i^p \in \mathtt{HJ}_k$, we recursively compute $\mathtt{W}^c(\mathtt{HJ}_k)$ as follows:

$$\mathtt{W}^c(\mathtt{HP}_k) = \max_{f=0}^{c} \left\{ \mathtt{W}^f(\{J_i^p\}) + \mathtt{W}^{(c-f)}(\mathcal{A}) \right\} \tag{8}$$

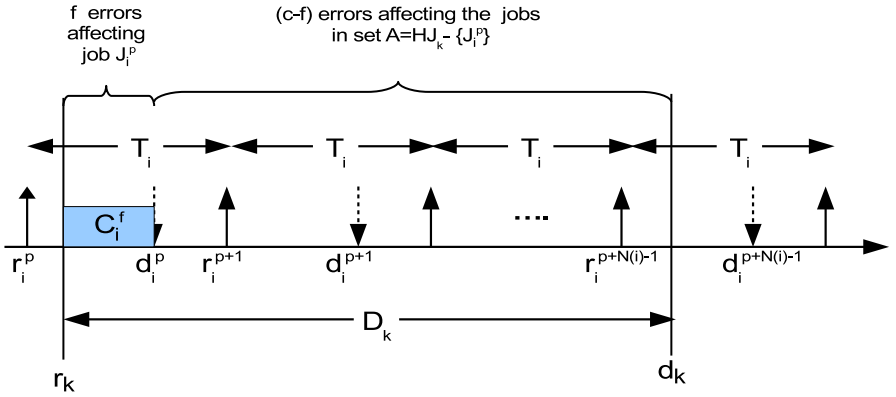

Fig. 3 The worst-case occurrence of c errors affecting the jobs in set $\mathtt{HJ}_k = \mathcal{A} \cup \{J_i^p\}$. The release time and deadline of job J_i^x are denoted respectively by r_i^x and d_i^x for $x = 1, 2 \ldots N(i)$

The value on the right-hand side in Eq. (8) is maximum for one of the $(c+1)$ possible values of f where $0 \leq f \leq c$. The value of f is selected such that if f errors exclusively affect job $J_i^p \in \mathtt{HJ}_k$ and $(c-f)$ errors affect the other jobs in set $\mathcal{A} = (\mathtt{HJ}_k - \{J_i^p\})$, then $\mathtt{W}^c(\mathtt{HJ}_k)$ is at its maximum for some f, $0 \leq f \leq c$. This is depicted in Fig. 3. By considering one-by-one higher priority job in set $\mathtt{HJ}_k$, the value of $\mathtt{W}^c(\mathtt{HJ}_k)$ can be computed using $O(c^2 \cdot |\mathtt{HJ}_k|)$ comparison and addition operations (i.e., workload computation in Eq. (8) has pseudo-polynomial time complexity in the representation of the system).

Example 2 Consider the taskset in Table 1. Also, consider the scheduling window of length $D_3 = 100$ for an arbitrary job J_3 of task τ_3. We have $hp(3) = \{\tau_1, \tau_2\}$ since task with lower index is assumed to have higher fixed priority.

Based on Eq. (5), at most $N(1) = \lceil \frac{max\{0,100-30\}}{100} \rceil + 1 = 2$ jobs of τ_1 and $N(2) = \lceil \frac{max\{0,100-120\}}{200} \rceil + 1 = 1$ job of τ_2 can execute in the scheduling window of J_3. Based on Eq. (6), $\mathtt{HJ}_3 = \{J_1^1, J_1^2, J_2^1\}$. Assume that jobs in $\mathtt{HJ}_3$ suffer $c = 2$ errors. We now show how to find $\mathtt{W}^2(\mathtt{HJ}_3)$.

First, we compute $\mathtt{W}^f(\{J_i^p\})$ for each job $J_i^p \in \mathtt{HJ}_3$ using Eq. (7) for $f = 0, 1, 2$. The results are shown in Table 2.

Now we will compute $\mathtt{W}^f(\{J_1^1, J_1^2\})$ for $f = 0, 1, 2$. Let $\mathcal{A} = \{J_1^1, J_1^2\} - \{J_1^1\} = \{J_1^2\}$. The values of $\mathtt{W}^f(\{J_1^1\})$ and $\mathtt{W}^f(\mathcal{A}) = \mathtt{W}^f(\{J_1^2\})$ are known for $f = 0, 1, 2$ from Table 2. Value of $\mathtt{W}^2(\{J_1^1, J_1^2\})$ is computed using Eq. (8) as follows:

Table 2 Value of $\mathtt{W}^f(\{J_i^p\})$ for each job $J_i^p \in \mathtt{HJ}_3$ for $f = 0, 1, 2$

	$\mathtt{W}^f(\{J_1^1\})$	$\mathtt{W}^f(\{J_1^2\})$	$\mathtt{W}^f(\{J_2^1\})$
$f = 0$	43	43	10
$f = 1$	43	43	22
$f = 2$	68	68	32

$$\begin{aligned}\mathtt{W}^2(\{J_1^1, J_1^2\}) &= \max_{f=0}^{2}\{\mathtt{W}^f(\{J_1^1\}) + \mathtt{W}^{(2-f)}(\mathcal{A})\}\\ &= \max_{f=0}^{2}\{\mathtt{W}^f(\{J_1^1\}) + \mathtt{W}^{(2-f)}(\{J_1^2\})\}\\ &= max\{43+68, 43+43, 68+43\} = 111\end{aligned}$$

Similarly, $\mathtt{W}^1(\{J_1^1, J_1^2\}) = 86$ and $\mathtt{W}^0(\{J_1^1, J_1^2\}) = 86$. And, based on the values of $\mathtt{W}^f(\{J_1^1, J_1^2\})$ and $\mathtt{W}^f(\{J_2^1\})$ for $f = 0, 1, 2$, the value of $\mathtt{W}^2(\{J_1^1, J_1^2, J_2^1\})$ can be computed.

4.3 Determining the schedulability of job J_k

In this subsection, we determine whether job J_k can meet its deadline on $\widehat{\mathtt{M}}$ non-faulty cores where the higher-priority jobs in set $\mathtt{HJ}_k$ suffer c errors and job J_k suffers $(\epsilon_k - c)$ errors in $[r_k, d_k)$ for any $c, 0 \leq c \leq \epsilon_k$. Remember that the ϵ_k errors in $[r_k, d_k)$ include je_k task errors and ρ core failures.

The primary and h_k active backups of job J_k can execute in parallel on the multicore processor because they become ready at the same time in FTM scheduling. And, each of the passive backups of J_k executes one-by-one. The total WCET of the passive backups of J_k is $\widehat{C}_k^{(\epsilon_k - c)}$ according to Eq. (2).

Theorem 1 (proof is given in the appendix) determines the total CPU time that needs to be allocated for completing the execution of the primary and all h_k active backups of J_k. This total CPU time is computed based on the observation that the primary and the h_k active backups can execute in parallel by exploiting the parallel multicore architecture. Based on this total CPU time required by the primary and h_k active backups of J_k (given in Theorem 1) and the workload of the higher priority jobs (derived in Sect. 4.2), we determine the minimum amount of sequential CPU time available for the passive backups of job J_k in $[r_k, d_k)$. And, a schedulability condition for J_k is derived.

Theorem 1 *The total CPU time that need to be allocated for the primary and all h_k active backups of job J_k in FTM scheduling on $\widehat{M}$ non-faulty cores is $(\widehat{M} \cdot s_{k,\widehat{M}})$ such that*

$$s_{k,\widehat{M}} = \max_{z=0}^{h_k}\left\{E_k^z + \frac{\sum_{b=0}^{z-1} E_k^b}{\widehat{M}}\right\} \tag{9}$$

where E_k^z is the WCET of the z^{th} backup[4] of J_k for $z = 0, 1, \ldots h_k$.

In Theorem 1, the value of $(\widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})$ is an upper bound on the sufficient amount of CPU time required to complete the execution of the primary and all h_k active backups of J_k. In Eq. (9), the first term on the right-hand side of the *max* function ensures that the z^{th} active backup with WCET E_k^z executes on at most one core at any time instant,

[4] For ease of discussion, the primary is called the "0^{th} backup" in Eq. (9). When $z = 0$, we consider $\sum_{b=0}^{z-1} E_k^b = 0$ in Eq. (9).

where $z = 0, 1 \ldots h_k$. And, the second term considers that the primary and different active backups of J_k are allowed to execute in parallel on $\widehat{\mathtt{M}}$ cores.

If the active backups are executed without any parallel execution (i.e., configured as passive backups), then the primary and all the h_k active backups of job J_k may need at most $\widehat{\mathtt{M}} \cdot \sum_{z=0}^{h_k} E_k^z$ execution time across all the cores. This is because when E_k^z executes in one core, then we have to make the worst-case assumption that all the other $(\widehat{\mathtt{M}} - 1)$ cores are idle. Since $\sum_{z=0}^{h_k} E_k^z \geq s_{k,\widehat{\mathtt{M}}}$, Theorem 1 implies that the total CPU time required for executing h_k number of backups actively is lower than that of the case when these h_k backups are executed passively. The following example demonstrates why executing certain backups in active redundancy is beneficial in reducing the worst-case workload that needs to be considered for the analysis of FTM algorithm.

Example 3 Consider that a job J of task τ_3 (given in Table 1) is affected by 2 errors at run-time. Assume that there are $\widehat{\mathtt{M}} = 2$ non-faulty cores. Since $h_3 = 1$, the primary and the active backup can run in parallel. Based on Eq. (9), the total execution time required for the primary and the active backup is $\widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}} = 2 \cdot s_{3,2} = 2 \cdot max\{E_3^0, E_3^1 + \frac{E_3^0}{2}\} = 2 \cdot max\{5, 10 + 5/2\} = 25$.

The total execution time requirement due to passive backup of J is $\widehat{C}_3^2 = 5$ time units according to Eq. (2). However, when a passive backup executes in one core, the other core may be idle in FTM scheduling. Consequently, the worst-case total CPU time need to be allocated during the execution of the passive backup is $2 \cdot \widehat{C}_3^2 = 2 \cdot 5 = 10$ time units. Therefore, total $(25 + 10) = 35$ CPU time units are required in the worst-case for the primary and all (active and passive) backups of job J_3.

In contrast, if all the backups of J_3 are passive (i.e., $h_3 = 0$), then one core may be idle in the worst-case when the primary and the passive backups are executed on the other core. Therefore, the total CPU time that may need to be allocated for the primary and passive backups to mask 2 errors is $\widehat{\mathtt{M}} \cdot (E_3^0 + E_3^1 + E_3^2) = 2 \cdot (5 + 10 + 5) = 40$ CPU time units. This shows that active redundancy requires less CPU time (i.e., 35 time units) than that of in pure passive redundancy (i.e., 40 time units) in FTM scheduling. On the other hand, if there is at no error during run-time and $h_3 = 1$, then only the primary and one active backup execute, and the total CPU time required is $\widehat{\mathtt{M}} \cdot (E_3^0 + E_3^1) = 2 \cdot (5 + 10) = 30$. This shows the power of passive redundancy that FTM exploits.

Based on the analysis in Sect. 4.2 and Theorem 1, if there are c errors that affect the jobs of set $\mathtt{HJ}_k$, then at most $(\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})$ units of CPU time are allocated for the higher priority jobs, the primary, and h_k backups of J_k. Since each core has speed 1, total $(\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})$ units of execution can be completed on $\widehat{\mathtt{M}}$ cores in at most $\lceil (\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})/\widehat{\mathtt{M}} \rceil$ time units. This implies the two following crucial observations:

Fact 1: The primary and all the active backups of J_k complete no later than time instant $(r_k + \lceil (\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})/\widehat{\mathtt{M}} \rceil)$ in FTM scheduling, where r_k is the release time of job J_k.

Fact 2: All the $\widehat{\mathtt{M}}$ cores are *simultaneously* busy executing jobs in set $\mathtt{HJ}_k \cup \{J_k\}$ except the passive backups of J_k for at most $\lceil (\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})/\widehat{\mathtt{M}} \rceil$ time units in $[r_k, d_k)$.

The passive backups of J_k can start execution in FTM scheduling only after the primary and all active backups of J_k are detected to be erroneous. From Fact 1, the maximum delay in detecting that the primary and all active backups of J_k are erroneous is at time $(r_k + \lceil (\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})/\widehat{\mathtt{M}} \rceil)$. And, from Fact 2, there is at least one core available for executing the passive backups of J_k in $[r_k, d_k)$ for at least $(D_k - \lceil (\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})/\widehat{\mathtt{M}} \rceil)$ time units after the primary and all active backups of J_k are detected to be erroneous. The passive backups of J_k needs $\widehat{C}_k^{(\mathtt{e}_k - c)}$ units of sequential execution according to Eq. (2). Given that jobs in $\mathtt{HJ}_k$ are affected by c errors and the job J_k is affected by $(\mathtt{e}_k - c)$ errors in $[r_k, d_k)$ for some given c, job J_k meets its deadline if

$$\lceil (\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})/\widehat{\mathtt{M}} \rceil + \widehat{C}_k^{(\mathtt{e}_k - c)} \leq D_k \tag{10}$$

Since c can vary between 0 and $\mathtt{e}_k$, by considering the worst-case occurrence of $\mathtt{e}_k$ errors in $[r_k, d_k)$ so that the value on the left-hand side of Eq. (10) is maximized, job J_k meets its deadline if

$$\max_{c=0}^{\mathtt{e}_k} \left\{ \lceil (\mathtt{W}^c(\mathtt{HJ}_k) + \widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})/\widehat{\mathtt{M}} \rceil + \widehat{C}_k^{(\mathtt{e}_k - c)} \right\} \leq D_k \tag{11}$$

From Eqs. (3)–(4), we have $\widehat{\mathtt{M}} = (\mathtt{M} - \rho)$ and $\mathtt{e}_k = (je_k + \rho)$. Therefore, Equation (11) can be re-written as:

$$\max_{c=0}^{(je_k+\rho)} \left\{ \left\lceil \frac{\mathtt{W}^c(\mathtt{HJ}_k)}{(\mathtt{M} - \rho)} + s_{k,(\mathtt{M}-\rho)} \right\rceil + \widehat{C}_k^{(je_k+\rho-c)} \right\} \leq D_k \tag{12}$$

Equation (12) essentially considers the worst-case occurrences of $(je_k + \rho)$ errors where the higher priority jobs suffer c errors in $[r_k, d_k)$ and job J_k suffers $(je_k + \rho - c)$ errors for any c where $0 \leq c \leq (je_k + \rho)$. Eq. (12) is a *sufficient schedulability condition* to check whether an arbitrary job of τ_k meets its deadline where there are at most je_k job errors and exactly ρ core failures in $[r_k, d - K)$. In the left-hand side of Eq. (12) and inside the *max* function, the term inside the ceiling function $\frac{\mathtt{W}^c(\mathtt{HJ}_k)}{(\mathtt{M}-\rho)} + s_{k,(\mathtt{M}-\rho)}$ is related to possible parallel execution of the primary and the active backups of job J_k and its higher priority jobs, and the second term $\widehat{C}_k^{(je_k+\rho-c)}$ is related to sequential execution of the passive backups of J_k. In order determine the probability of meeting all the deadlines of all the tasks, Eq. (12) has to be evaluated for each task $\tau_k \in \Gamma$. We have the following theorem (proof follows directly from the analysis presented in this subsection):

Theorem 2 *Consider a system of n sporadic tasks $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$ according to the task model defined in Sect. 2. Each job of task $\tau_k \in \Gamma$ can tolerate je_k task errors for a given ρ number of core failures if Eq. (12) is satisfied.*

Remember that our aim is to find the largest je_k that satisfies Eq. (12) for each task $\tau_k \in \Gamma$. No job of $\tau_k \in \Gamma$ can tolerate more that $D_k \cdot (\mathtt{M} - \rho)$ task errors. The largest

je_k that satisfies Eq. (12) can be searched linearly (or using bisection search) in the range $[0, D_k \cdot (\mathtt{M} - \rho)]$. Given some ρ, if Eq. (12) is not satisfied even for $je_k = 0$, then τ_k cannot be guaranteed to meet all the deadlines on $(\mathtt{M} - \rho)$ cores even when there is no job error. In such case, we set $je_k = -\infty$. Since workload computation has pseudo-polynomial time complexity, Eq. (12) can also be computed to find the largest je_k in pseudo-polynomial time.

Note that Theorem 2 for the assumed fault model of this paper is not a system level schedulability condition that can be used to determine whether all the tasks meet their deadlines or not[5]. Theorem 2 is used to determine the maximum number of task errors that each job of a particular task can tolerate for a given number of core failures. Based on this maximum number, we determine in next section the probability of meeting deadline of all jobs of each task. Finally, the probability of meeting deadline of all the tasks is computed, which is the system-level schedulability guarantee [please see Eq. (15) of Theorem 3].

Bookkeeping for Probabilistic Analysis For each task, the number of job errors that each of its jobs can tolerate in its scheduling window depends on how many cores are non-faulty. The maximum number of errors that can be tolerated by any job of a task decreases as the number of faulty cores increases. The schedulability condition in Eq. (12) captures this fact by considering the parameter ρ that can take values ranging from 0 to M, where M is the number of maximum cores of the multicore processor. For each value of ρ, the maximum number of job errors that each job of a task can tolerate is determined using Eq. (12).

We find the largest je_k for each (k, ρ) pair where $k \in \{1, 2, \ldots n\}$ and $\rho \in \{0, 1 \ldots \mathtt{M}\}$ using Eq. (12). And, these $n \times (\mathtt{M} + 1)$ values are stored (for probabilistic schedulability analysis) in a $n \times (\mathtt{M} + 1)$ matrix, called the $\mathcal{S}$ *chedulability matrix*, denoted by $\mathcal{S}$. The schedulability matrix is used to store the information regarding the maximum number of job error that any job of a task can tolerate for different number of faulty cores. The entry $\mathcal{S}_{k,\rho}$ in matrix $\mathcal{S}$ is interpreted during the probabilistic schedulability analysis as follows:

- If $\mathcal{S}_{k,\rho} = -\infty$, then *some* job of τ_k cannot guaranteed to be schedulable when ρ cores fails in $[r_k, d_k)$ even if there is no job error.
- If $\mathcal{S}_{k,\rho} \geq 0$, then *some* job of τ_k cannot guaranteed to be schedulable on $(\mathtt{M} - \rho)$ cores if the number of job errors between the release and deadline of the job (i.e., in an interval of length D_k) exceeds $\mathcal{S}_{k,\rho}$ given that exactly ρ cores are faulty.

Example 4 Consider the FTM scheduling of $n = 5$ tasks in Table 1 on $\mathtt{M} = 4$ cores. Task with lower index has higher priority. We computed the schedulability matrix $\mathcal{S}$ of size (5×5) based on Eq. (12). The result is shown in Table 3.

[5] If a fault model considers that *any* job can suffer at most a fixed number of task errors and a fixed number of core failures, then Eq. (12) is applicable to determine whether all the tasks meet their deadlines or not. The fault model that we consider in this paper allows that jobs of different tasks may suffer different number of maximum errors.

Table 3 Schedulability matrix $\mathcal{S}$ for the tasks of the Instrument Control application in Table 1 for $\mathtt{M} = 4$

Task's name	Number of core failures				
	$\rho = 0$	$\rho = 1$	$\rho = 2$	$\rho = 3$	$\rho = 4$
Mode management (τ_1)	2	1	0	$-\infty$	$-\infty$
Mission data management (τ_2)	4	2	0	$-\infty$	$-\infty$
Instrument monitoring (τ_3)	11	6	2	$-\infty$	$-\infty$
Instrument configuration (τ_4)	1	0	$-\infty$	$-\infty$	$-\infty$
Instrument processing (τ_5)	3	1	$-\infty$	$-\infty$	$-\infty$

Number of faulty cores, i.e, value of ρ ranges from 0 to $\mathtt{M} = 4$ to capture the effect increasing number of permanent core failure. Each cell in the matrix specifies the maximum number of job errors that any job of each task τ_k (given in each row) can mask when there are exactly ρ core failures (given in each column)

5 Probabilistic schedulability analysis

The schedulability analysis in the previous section is based on application-level error model (i.e., considering job errors and core failures) and is independent of the underlying hardware-level fault model. Based on this schedulability analysis, the maximum number of job errors that any job of a task can tolerate (i.e., deadline of the job is met) for a given number of core failures is derived and captured in the schedulability matrix $\mathcal{S}$. However, some faulty environment may cause too many job errors and/or core failures that jobs may miss their deadlines. In other words, the "actual" number of faults at run-time may cause a number of errors that is larger than the maximum number of errors that a job of some task can tolerate. In such case, the schedulability is not guaranteed.

The question is what is the probability that the number of actual errors during run-time is larger than the maximum number of errors that a job can tolerate. The answer to this question depends on the hardware-level fault model. By considering the stochastic behavior of specific fault model, the schedulability of an application can be derived with a probabilistic guarantee, i.e., the likelihood of the number of actual errors being not larger than the maximum number of job errors and core failures that any job of any task can withstand.

In this section, we first derive a general closed-form expression considering an arbitrary fault model to compute the probability of meeting all the deadlines (i.e., probabilistic schedulability guarantee). Then, the stochastic behavior of $\mathcal{B}$ and $\mathcal{R}$ fault models are used to compute the probability of meeting all the deadlines based on this general expression.

In order to find the probability of meeting all the deadlines of all the tasks, we first determine using Theorem 2 the maximum number of task errors a job of a task can tolerate for a given number of core failures. Second, the probability that *one* job of a task meets its deadline is computed by considering the probability that the number of task errors for a given number of core failures does not exceed the maximum tolerable number of task error for a job of the task [please see Eq. (13)]. Then, the probability that *all* the jobs of a task meet their deadlines is computed by considering the prob-

ability that one job of the task meets its deadline [please see Eq. (14)]. Finally, the probability that all jobs of *all* the tasks meet their deadlines is computed by considering the probability that all jobs of each individual task meet their deadlines [please see Eq. (15)]. The following notations are used in this section:

- CF_k: a random variable that counts the number of *permanent hardware faults* in an interval of length D_k. This variable captures the number of cores failures in an interval of length D_k.
- JE_k: a random variable that counts the number of *non-permanent hardware faults* in an interval of length D_k. This variable captures the number of job errors in an interval of length D_k.
- $Pr(\mathcal{X})$: probability of occurring an event $\mathcal{X}$. For example, $Pr(CF_k = \rho)$ is the probability that exactly ρ core fail in an interval of length D_k.
- $PrF_{k,\rho}$: probability that some job of task τ_k cannot guaranteed to be schedulable. The value of $PrF_{k,\rho}$ will be computed based on two scenarios under which some job of τ_k may miss the deadline: (i) regardless of how many job errors are there in an interval of length D_k, the job misses its deadline if exactly ρ cores are faulty, and (ii) when exactly ρ cores are faulty, the job misses its deadline if the number of actual job errors is larger than the maximum number of errors that can be tolerated by the job (available from the schedulability matrix) in its scheduling window.
- PrS: the probability that all the jobs of all the tasks meet their deadlines, i.e., PrS is the *probabilistic schedulability guarantee* of the system of all tasks.
- LT: the lifetime of the mission of the system.

It is assumed that each permanent hardware fault leads to exactly one core failure. However, there might be source of permanent (fault-triggering) event such that one such event causes permanent fault in all the cores. We do not address fault tolerance for the entire multicore chip, which needs to be tackled using hardware redundancy (e.g., TMR multicore chips).

Similarly, there might be source of non-permanent (fault-triggering) event such that one such event may cause non-permanent faults in all the cores (known as *fault propagation*). We address such scenarios by mapping the failure rate of non-permanent (fault-triggering) faults to each core of the multicore chip. In other words, if the rate of occurring non-permanent fault-triggering event at the multicore chip level is λ, then we consider that non-permanent faults in *each core* occur at rate λ.

Finally, we assume that each non-permanent hardware fault in one core causes exactly one job error executing on that core. This is reasonable under the assumption that there is no *error propagation* across jobs. However, if error propagation across jobs is likely, then a job error in turn may cause additional (propagated) job errors. The number of such propagated job errors due to one non-permanent fault depends on the number of jobs that are currently in the preempted state. This is because jobs that are not released or has not yet started their execution are not affected due to fault propagation and thus cannot cause the workload due to be increased. Therefore, the workload due to fault propagation can be increased by the amount equal to the sum of execution time of all lower priority (preempted) jobs of task τ_k. Extending the probabilistic analysis assuming error propagation therefore does not pose any fundamental challenge and its complete analysis is left as a future work. In summary,

each permanent and each non-permanent fault causes exactly one core failure and exactly one job error, respectively.

Now consider the FTM schedule of an application. All the different cases for which some job of task τ_k of the application may miss its deadline are available from the $(\mathtt{M}+1)$ columns of the kth row in the schedulability matrix $\mathcal{S}$, i.e., in entries $\mathcal{S}_{k,0}, \mathcal{S}_{k,1} \ldots \mathcal{S}_{k,\mathtt{M}}$. Given that one fault causes one error, the probability $PrF_{k,\rho}$ that some job of task τ_k misses its deadline when exactly ρ cores fail is determined based on $\mathcal{S}_{k,\rho}$ as follows:

- If $\mathcal{S}_{k,\rho} = -\infty$, then some job of task τ_k cannot tolerate ρ core failures in its scheduling window. Consequently, the probability that some job of task τ_k misses its deadline is equal to the probability that there are exactly ρ permanent hardware faults in an interval of length D_k regardless of the number of job errors in that interval. And, this probability is $Pr(CF_k = \rho)$.
- If $\mathcal{S}_{k,\rho} \geq 0$, then some job of task τ_k cannot tolerate more than $je_k = \mathcal{S}_{k,\rho}$ job errors in an interval of length D_k given that there are exactly ρ core failures. Consequently, the probability that some job of task τ_k misses its deadline is equal to the probability that there are exactly ρ permanent hardware faults and there are more than $\mathcal{S}_{k,\rho}$ non-permanent hardware faults in an interval of length D_k. And, this probability is equal to the product of $Pr(JE_k > \mathcal{S}_{k,\rho})$ and $Pr(CF_k = \rho)$. This is because the sources of faults that cause core failures and job errors are assumed to be independent: the former is caused by permanent hardware faults while the latter is caused by non-permanent faults.

Therefore, the probability $PrF_{k,\rho}$ that some job of task τ_k misses its deadline when exactly ρ cores fail in FTM scheduling is computed based on $\mathcal{S}_{k,\rho}$ as follows:

$$PrF_{k,\rho} = \begin{cases} Pr(CF_k = \rho) & \text{if } \mathcal{S}_{k,\rho} = -\infty \\ Pr(JE_k > \mathcal{S}_{k,\rho}) \cdot Pr(CF_k = \rho) & \text{if } \mathcal{S}_{k,\rho} \geq 0 \end{cases} \tag{13}$$

where the value of $\mathcal{S}_{k,\rho}$ is obtained from the schedulability matrix that is derived using the schedulability condition in Eq. (12). The probability that some job of task τ_k misses its deadline for *any* number of core failures is $\sum_{\rho=0}^{\mathtt{M}} PrF_{k,\rho}$. Therefore, the probability that a job of task τ_k is guaranteed to meet its deadline is $(1 - \sum_{\rho=0}^{\mathtt{M}} PrF_{k,\rho})$.

There are at most $\lceil \frac{LT}{T_k} \rceil$ jobs of task τ_k that may be released during the mission's lifetime LT. The task τ_k is guaranteed to be schedulable in FTM scheduling if all $\lceil \frac{LT}{T_k} \rceil$ jobs are schedulable. Since the scheduling windows of different jobs of task τ_k do not overlap in time (these scheduling windows are separated by at least $(T_k - D_k)$ time units), the event that a job of task τ_k is erroneous is independent of the event that another job of the same task is erroneous. So, the probability that all the jobs of task τ_k meet their deadlines is given as follows:

$$\left(1 - \sum_{\rho=0}^{\mathtt{M}} PrF_{k,\rho}\right)^{\lceil \frac{LT}{T_k} \rceil} \tag{14}$$

Finally, the probability PrS that all the jobs of all the n tasks of an application meet their deadlines in FTM scheduling is given (proof follows directly from the discussion presented above) in the following Theorem 3:

Theorem 3 *Consider a system of n sporadic tasks $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$ according to the task model presented in Sect. 2. The probability of meeting deadlines of all the jobs of all the tasks in set Γ is:*

$$\begin{aligned} PrS &= \prod_{k=1}^{n} \left(1 - \sum_{\rho=0}^{M} PrF_{k,\rho}\right)^{\lceil \frac{LT}{T_k} \rceil} \\ &= \left(1 - \sum_{\rho=0}^{M} PrF_{1,\rho}\right)^{\lceil \frac{LT}{T_1} \rceil} \times \left(1 - \sum_{\rho=0}^{M} PrF_{2,\rho}\right)^{\lceil \frac{LT}{T_2} \rceil} \\ &\quad \times \cdots \times \left(1 - \sum_{\rho=0}^{M} PrF_{n,\rho}\right)^{\lceil \frac{LT}{T_n} \rceil} \end{aligned} \tag{15}$$

The value of PrS in Eq. (15) is the probabilistic schedulability guarantee of the application. In order to find PrS using Eq. (15), we have to find $PrF_{k,\rho}$ for each possible (k, ρ) pair where $k \in \{1, 2, \ldots n\}$ and $\rho \in \{0, 1, \ldots M\}$. According to Eq. (13), the value of $PrF_{k,\rho}$ can be computed based on $Pr(CF_k = \rho)$ and $Pr(JE_k > \mathcal{S}_{k,\rho})$. Eq. (15) is the general closed-from expression that can be used to find the probabilistic schedulability guarantee by computing $Pr(CF_k = \rho)$ and $Pr(JE_k > \mathcal{S}_{k,\rho})$ considering the stochastic behavior of *any* fault model. We conclude this section by showing how to compute the values of $Pr(CF_k = \rho)$ and $Pr(JE_k > \mathcal{S}_{k,\rho})$ considering the stochastic behavior of $\mathcal{B}$ and $\mathcal{R}$ fault models. Finally in Sect. 7, the value of PrS is computed for the Instrument Control application given in Table 1 to show the applicability of the research presented in this paper.

Computing $Pr(CF_k = \rho)$ for $\mathcal{B}$ and $\mathcal{R}$ Model Permanent hardware faults in both $\mathcal{B}$ and $\mathcal{R}$ fault models are assumed to occur randomly at constant failure rate λ_c. The inter-arrival time of permanent hardware faults has exponential distribution. The probability of occurring exactly ρ permanent hardware faults in an interval of length D_k can be determined using Poisson process with rate λ_c as follows:

$$Pr(CF_k = \rho) = \frac{e^{-(\lambda_c \cdot D_k)} \cdot (\lambda_c \cdot D_k)^{\rho}}{\rho!} \tag{16}$$

Since Poisson process has *stationary* and *independent* increments, the probability of occurring exactly ρ permanent faults in the scheduling window of each job of τ_k is equal because all jobs of task τ_k have the same length of scheduling window. Therefore, $Pr(CF_k = \rho)$ is equal for all the $\lceil \frac{LT}{T_k} \rceil$ jobs of τ_k that may be released during the lifetime.

Our assumption that the probability of permanent hardware faults follows a Poisson process is fairly a standard in many hardware literature (Koren and Krishna 2007).

This assumption is reasonable for many practical multicore systems that are subjected to a particular fault model, i.e., where one core failure does not lead other cores to fail. For example, if the ALU of a core becomes faulty, then such a failure may not lead other cores of a multicore processor to fail. In such case, our assumption that consecutive failures are independent is valid since other cores could operate normally.
Computing $Pr(JE_k > \mathcal{S}_{k,\rho})$ for $\mathcal{B}$ model The $\mathcal{B}$ model considers that non-permanent faults occur in bursts as well as randomly during bursty and non-bursty periods, respectively. Fault-burst model is well-studied for error structure in data transmission, for example, using Gilbert–Elliott model (Elliott 1963). The $\mathcal{B}$ model for non-permanent faults is essentially the Gilbert–Elliott model applied to fault-tolerant uniprocessor scheduling by Short and Proenza (2013). We extend this model for multicores in this paper. Short and Proenza computed an upper (hence, pessimistic) bound on the number of errors in an interval for a *given* probability. In contrast, we want to compute the probability of exceeding a certain number of faults, i.e., value of $Pr(JE_k > \mathcal{S}_{k,\rho})$, in an interval of length D_k.

To better understand the $\mathcal{B}$ model for multicores, we first present how the Gilbert–Elliott model is used by Short and Proenza for uniprocessor and then show how we extend its analysis for multicores. It will be evident shortly that the number of non-permanent faults in an interval of particular length under the $\mathcal{B}$ fault model is a random variable, denoted by Υ, having Poisson Binomial distribution. The cumulative distribution function (CDF) of Υ can be computed based on the technique proposed by Barlow and Heidtmann (1984) to find the probability that there are at most $\mathcal{S}_{k,\rho}$ non-permanent faults in an interval of length D_k. And, 1 minus this probability is $Pr(JE_k > \mathcal{S}_{k,\rho})$, i.e., the probability that number of non-permanent faults in an interval of length D_k is larger than $\mathcal{S}_{k,\rho}$.
Gilbert–Elliott Model (Elliott 1963) This model is represented by a Markov model with two states **G** and **B** that represent 'non-bursty/good' and 'bursty/bad' states, respectively. Short and Proenza (2013) used this model for fault-tolerant uniprocessor scheduling using four parameters: λ_b, λ_r, L_G, and L_B. We use these parameters for *multicores* as follows.

The non-permanent faults in *each* core occur at constant failure rates λ_r and λ_b when the system is in state G and B, respectively. The expected (mean) inter-arrival time of non-permanent faults in both states have geometric (discrete equivalent of exponential) distribution since faults are assumed to affect tasks at integer time instants. The expected inter-arrival time of fault bursts is $(L_B + L_G)$ where L_B and L_G (see Fig. 1) are respectively the expected length of bursty and non-bursty periods, both having geometric distributions.

The probability that the system transits from state G to B (denoted by P_{GB}) is equal to $1/L_G$. The larger is the expected length of non-bursty period L_G, the smaller is the probability P_{GB} that the system transits to bursty(B) state. The probability that the system transits from state B to G (denoted by P_{BG}) is $1/L_B$. The two-state Gilbert–Elliott Markov model for each core is shown in Fig. 4.

We need to analyze the Markov model in Fig. 4 to compute the probability $Pr(JE_k > \mathcal{S}_{k,\rho})$. Short and Proenza (2013) modeled the probability of a fault occurrence at each time instant t using an independent and non-identically distributed Bernoulli variable I_t. We now extend this concept for multicore. For simplicity

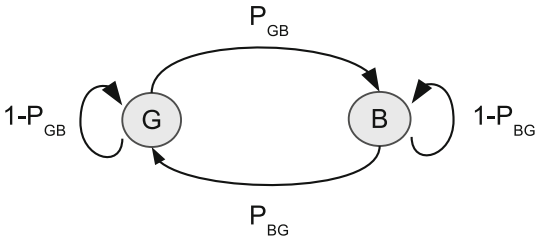

Fig. 4 Gilbert–Elliott Markov model for fault bursts in a core. When the system is in state G and B, the non-permanent faults in each core occurs at rate λ_r and λ_b, respectively

of presentation, we denote r_k and d_k simply as r and d. We consider the scheduling window $[r, d)$ of an arbitrary job J_k of task τ_k in FTM scheduling where $(d - r) = D_k$. Note that there are $\widehat{\mathtt{M}} = (\mathtt{M} - \rho)$ non-faulty cores when computing $Pr(TE_k > \mathcal{S}_{k,\rho})$.

Without loss of generality, we order the $\widehat{\mathtt{M}}$ non-faulty cores as $1^{st}, 2^{nd}, \ldots, \widehat{\mathtt{M}}^{th}$ core. Unlike uniprocessor, multiple non-permanent faults can affect different cores at the same time instant. Since the primary and active backups of a job can execute in parallel, multiple job errors may be detected to be erroneous on different cores at the same time instant. Our consideration of the failure rate of non-permanent faults at the individual core level rather than at the multicore chip level takes this fact into account.

We consider total $(\widehat{\mathtt{M}} \cdot D_k)$ independent and non-identically distributed Bernoulli variables I_t^{ν} to model the probability of a non-permanent fault occurrence at time instant t in the ν^{th} core, where $t = r, r+1, \ldots (d-1)$ and $\nu = 1, 2, \ldots \widehat{\mathtt{M}}$. In other words, $I_t^{\nu} \sim Bernoulli(p_t^{\nu})$, where $p_t^{\nu} = Pr(I_t^{\nu} = 1)$ is the probability that there is a non-permanent fault at time t on the ν^{th} core. The value of p_t^{ν} for each time instant can be computed as follows (Short and Proenza 2013):

$$p_t^{\nu} = \lambda_b \cdot m_t + \lambda_r \cdot (1 - m_t) \tag{17}$$

where m_t and $(1 - m_t)$ are the probabilities that the system is in state B and state G at time t, respectively. And, λ_b and λ_r are the failure rates of non-permanent faults in each core in state B and G, respectively. The probability m_t (i.e., the system is in state B) can be computed by solving the Markov model in Fig. 4 for some given initial state at time $t = r$. The worst-case initial state [as explained in Short and Proenza (2013)] is $m_r = 1$, i.e., the system starts in a bursty state at time r with probability 1. This worst-case ensures that the number of non-permanent faults in *any* interval of length D_k is maximized. Assuming $m_r = 1$, the value of m_{t+1} for $t = (r+1), \ldots (d-1)$ is computed recursively as follows:

$$m_{t+1} = (1 - P_{BG}) \cdot m_t + P_{GB} \cdot (1 - m_t) \tag{18}$$

where $P_{BG} = 1/L_B$ and $P_{GB} = 1/L_G$. The probability p_t^{ν} for each $t = r, (r+1), \ldots (d-1)$ and each $\nu = 1, 2 \ldots \widehat{\mathtt{M}}$ can be computed using Eqs. (17) and (18) based on the four parameters of the $\mathcal{B}$ model: λ_b, λ_r, L_G and L_B.

Given the probabilities p_t^{ν} for $t = r, (r+1), \ldots (d-1)$ and $\nu = 1, 2, \ldots \widehat{\mathtt{M}}$, our aim is to compute the probability of more than $\mathcal{S}_{k,\rho}$ faults in $[r, d)$. We denote Υ the random variable that counts the number of non-permanent faults in $[r, d)$. The variable Υ is sum

of $(\widehat{\mathtt{M}} \cdot D_k)$ non-identical Bernoulli variables $I_r^1, \ldots I_{d-1}^1, I_r^2, \ldots I_{d-1}^2, \ldots I_r^{\widehat{\mathtt{M}}}, \ldots I_{d-1}^{\widehat{\mathtt{M}}}$ and has Poisson Binomial distribution (Hong 2013). Given the probabilities p_t^v for $t = r, (r+1), \ldots (d-1)$ and $v = 1, 2, \ldots \widehat{\mathtt{M}}$, our aim is to compute the probability of more than $\mathcal{S}_{k,\rho}$ faults in $[r, d)$. We denote Υ the random variable that counts the number of non-permanent faults in $[r, d)$. The variable Υ is sum of $(\widehat{\mathtt{M}} \cdot D_k)$ non-identical Bernoulli variables $I_r^1, \ldots I_{d-1}^1, I_r^2, \ldots I_{d-1}^2, \ldots I_r^{\widehat{\mathtt{M}}}, \ldots I_{d-1}^{\widehat{\mathtt{M}}}$ and has Poisson Binomial distribution (Hong 2013).

The CDF of Υ, denoted by $F_\Upsilon(x)$, is the probability of having at most x faults in the interval $[r_k, d_k)$ where $0 \leq x \leq (\widehat{\mathtt{M}} \cdot D_k)$. Hong (2013) proposed technique to compute $F_\Upsilon(x)$ based on the probabilities $p_r^1, \ldots p_{d-1}^1, p_r^2, \ldots p_{d-1}^2, \ldots, p_r^{\widehat{\mathtt{M}}}, \ldots, p_{d-1}^{\widehat{\mathtt{M}}}$ (Hong 2013) has an easy-to-understand description of $F_\Upsilon(x)$). After computing $F_\Upsilon(\mathcal{S}_{k,\rho})$ based on the work by Hong (2013), the probability of more than $\mathcal{S}_{k,\rho}$ faults in an interval of length D_k for the $\mathcal{B}$ model is

$$Pr(TE_k > \mathcal{S}_{k,\rho}) = 1 - F_\Upsilon(\mathcal{S}_{k,\rho}) \tag{19}$$

Computing $Pr(JE_k > \mathcal{S}_{k,\rho})$ for $\mathcal{R}$ model The $\mathcal{R}$ model is a specialization of the $\mathcal{B}$ model with no bursty period. We set $m_r = 0$, $(1 - m_r) = 1$ and $P_{GB} = 0$ in Eq. (18) so that the system starts in non-bursty state and never transits to bursty state. Consequently, the $\mathcal{B}$ model is now equivalent to $\mathcal{R}$ model where λ_r is the constant failure rate of non-permanent faults in each core in state G. The value of $Pr(TE_k > \mathcal{S}_{k,\rho})$ is computed using Eq. (19) based only on parameter λ_r.

By computing the values of $Pr(CF_k = \rho)$ and $Pr(TE_k > \mathcal{S}_{k,\rho})$ respectively using Eqs. (16) and (19) for each entry of the matrix $\mathcal{S}$, the value of PrS, i.e., probability of meeting all deadlines of an application can be computed according to Eq. (15). Since the considered window has length D_k and because the schedulability matrix can be generated in pseudo-polynomial time, the time complexity to find the value of PrS for a given application is pseudo-polynomial, which is acceptable for (offline) schedulability analysis of real-time systems (Baruah et al. 1990).

6 A heuristic for configuring the number of active backups

Algorithm `FTM` considers h_i number of active backups (configured by the designer) for each task $\tau_i \in \Gamma$. Configuring the number of active backup appropriately is a major challenge due to three reasons:

- the different possible of ways the number active backups for all the tasks can be configured is *exponential* since each h_i can be within the range between $0, \ldots, \infty$;
- setting h_i too high is a wastage of processing resource if not too many errors are likely to occur; and
- setting h_i too low may not fully exploit the parallel multicore architecture. A relatively higher number of backups may be considered as passive backups to mask the errors. Since passive backups are executed one-by-one (i.e., sequentially) in `FTM` algorithm, some low laxity (shorter deadline) task may miss its deadline.

To address these challenges and to help the system designer in deciding the number of active backups for each task, a heuristic is proposed. The basic principle of the

heuristic is that: *assuming no core failure, increment the number of active backup for those tasks that can tolerate a relatively smaller number of job errors*. The step-by-step process to implement this heuristic is as follows:

①Initially, no backup of any task is considered as active, i.e., $\forall i$, $h_i = 0$. Generate the schedulability matrix $\mathcal{S}$ based on the schedulability condition in Eq. (12). And, compute the PrS using Eq. (15) considering the fault model of the target system.

② Based on the schedulability matrix $\mathcal{S}$ computed in last step, determine the task (called *target* task) that tolerates the minimum number of job errors when no core is faulty, i.e., when $\rho = 0$. Obviously, the task τ_i with minimum $\mathcal{S}_{i,\rho=0}$ for $i = 1, 2, \ldots n$ is the target task. Executing active backups for this target task may increase the maximum number of tolerable job errors for each job of τ_i, which may increase PrS. This is because the active backup can execute in parallel and thus can finish earlier than the case if the backup is executed passively.

③ Increment h_i of the target task τ_i by 1, i.e., set $h_i = h_i + 1$. Generate the schedulability matrix $\mathcal{S}$ based on the schedulability condition in Eq. (12) considering the incremented value of h_i. And, compute the PrS using Eq. (15).

④ If PrS is increased, then go to Step 2 and repeat the process.

⑤ If PrS is not increased, then the last increment in Step 3 is *undone*, i.e., set $h_i = h_i - 1$. Task τ_i will not be considered as a target task any further since increasing the number of active backups for τ_i does not increase PrS.

⑥ If not all the n tasks are considered as target tasks, then generate the schedulability matrix $\mathcal{S}$ based on the schedulability condition in Eq. (12) considering the decremented value of h_i in last step. Go to Step 2 to select the next target task.

⑦If all the n tasks are considered as target tasks, then stop.

In summary, we start with a system with all backups configured as passive backups. We consider a system where there is no core failures. We increase the number of active backups for that particular task that can tolerate the minimum number of job errors if doing so results in the value of PrS to increase. We stop when PrS does not increase by incrementing number of active backups for any task. Under this heuristic, the PrS is increased because more active backups that can execute in parallel can effectively tolerate more job errors before deadline.

7 Case study: instrument control application

In this section, PrS is computed for the Instrument Control application in Table 1 hosted on a multicore platform having $\mathtt{M} = 4$ cores. Note that $h_1 = 1$, $h_2 = 0$, $h_3 = 1$, $h_4 = 0$, and $h_5 = 1$. The Schedulability Matrix $\mathcal{S}$ of the application for $\mathtt{M} = 4$ is already given in Table 3. Before we present the results, the different values of fault-model parameters used for computing PrS are presented.

Parameters of Fault Model In general, failure rate is not very easy to derive since it depends on many factors, for example, the underlying hardware, the software implementation, and operating environment. Remember that the $\mathcal{B}$ fault model is characterized using five parameters: λ_c, λ_b, L_B, λ_r, and L_G whereas the $\mathcal{R}$ fault model is characterized using two parameters: λ_c and λ_r. The value of PrS for the Instrument

Control application is computed based on the following values for the $\mathcal{B}$ and $\mathcal{R}$ fault models.

The failure rate of permanent hardware faults is $\lambda_c = 10^{-5}$/h. Similar failure rate for permanent faults is also considered in Claesson (2002) where the fault rate of a computer node for heavy duty trucks is derived based on the MIL-HDBK-217 standard. Experiments by Campbell et al. (1992) using an orbiting satellite containing a microelectronics test system found that, within a small time interval (∼15 min), the number of errors due to transient faults could be quite high. Since non-permanent faults are more frequent, the failure rate of random non-permanent hardware faults in each core during *non-bursty* period is $\lambda_r = 10^{-4}$/h. In other words, non-permanent faults during non-bursty period occurs at a rate 10 times higher than that of permanent faults. Finally, we consider $\lambda_b = 10^{-2}$/s since non-permanent hardware faults during a *burst* is much higher. The failure rate λ_c applies to the entire multicore chip while the failure rates λ_b and λ_r apply to each individual core of the multicore chip.

The value of L_G (i.e., expected duration of one non-bursty period) is $L_G = 10^6$ ms. Since the duration of bursty period is relatively smaller than non-bursty period, the value of L_B (i.e., expected duration of one bursty period) is $L_B = 10^2$ ms. In summary, the values of the parameters for the $\mathcal{B}$ and $\mathcal{R}$ fault models are given in Table 4.

Experiments Typical mission lifetime of safety-critical system varies, for example, from several hours for aircraft to several years for satellites. We considered four different mission's lifetimes $LT \in \{10\text{ h}, 1\text{ day}, 1\text{ month}, 1\text{ year}\}$.

An experiment is characterized by a fault model and the lifetime of the system. We conducted 8 experiments considering each of the four mission's lifetimes, and each of the $\mathcal{B}$ and $\mathcal{R}$ fault models using parameters from Table 4. For each experiment, we computed PrS using Eq. (15) for the Instrument Control application in Table 1. We wrote a C program to implement Eq. (15) and ran our experiments on a Dell Optilex 990 Intel Dual Core i7-2600 processor, each with speed 3.40 GHz. The total run-time to compute the 8 values of PrS was 63 ms. The results of the 8 experiments are shown in Table 5. Each row in Table 5 represents the values of PrS for four different lifetimes for a given fault model.

Table 4 Values of different parameters of $\mathcal{B}$ and $\mathcal{R}$ fault models

λ_c	λ_r	λ_b	L_G	L_B
10^{-5}/h	10^{-4}/h	10^{-2}/s	10^6 ms	10^2 ms

Table 5 Value of PrS for Instrument Control application considering the $\mathcal{B}$ and $\mathcal{R}$ fault models using model parameters in Table 4

Fault model	10 h	1 day	1 month	1 year
$\mathcal{R}$	0.99999999	0.99999999	0.99999997	0.99999973
$\mathcal{B}$	0.99999667	0.99999336	0.99986397	0.99838547

As expected the PrS of the Instrument Control application decreases with larger lifetime. Moreover, the $\mathcal{B}$ fault model results in a relatively lower value of PrS in comparison to that of the $\mathcal{R}$ fault model. The explanation is given below:
Impact of Lifetime The PrS decreases with longer lifetime for each particular fault model. For example, PrS for the $\mathcal{R}$ model *decreases* from 0.99999999 to 0.99999973 when the lifetime is increased from 10 hours to 1 year. A similar trend can be observed for the $\mathcal{B}$ model in Table 5. This is because, PrS in Eq. (15) is the product of the probabilities of meeting deadlines of *all* jobs that may be released during the lifetime, and larger number of jobs are allowed to be released with longer lifetime.
Impact of Bursts Burst of non-permanent faults have severe impact on PrS. We observe that the PrS in $\mathcal{B}$ fault model for each lifetime is noticeably smaller than that of in $\mathcal{R}$ fault model. For example, the value of PrS in $\mathcal{R}$ model for a lifetime of 1 month *decreases* from 0.99999997 to 0.99986397 in $\mathcal{B}$ fault model. This is because the failure rate during a burst is much higher, and therefore, a relatively higher number of errors may need to be masked for each job in $\mathcal{B}$ model than that of in $\mathcal{R}$ model. This shows that a given application may have different PrS for different fault models.

The result of this section is based on the parameters of the tasks and the parameters of the fault model. If the parameters of the tasks (e.g., period, deadline or WCET) are changed or a different failure rate for the fault models is considered, then the value of PrS will be different. This paper proposes a methodology to determine the reliability of a system over some lifetime in terms of real-time schedulability and considering faulty multicore environment. Such methodology can be used by the system designer to determine the confidence regarding the correctness of the system with respect to generating non-erroneous output by some deadline.

Impact of the Heuristic to Determine Number of Active Backups The results in Table 5 are for Instrument Control application in Table 1 where the number of active backups for the five tasks are $h_1 = 1$, $h_2 = 0$, $h_3 = 1$, $h_4 = 0$, and $h_5 = 1$. The PrS for lifetime of 1 year for the $\mathcal{B}$ fault model in Table 5 is **0.99838547**. We also computed the PrS where all backups are passive ($\forall i, h_i = 0$) as is assumed in many previous works (Chen et al. 2007; Hashimoto et al. 2000; Kim et al. 2010; Bertossi et al. 1999; Oh and Son 1994; Berten et al. 2006; Liberato et al. 1999; Pathan and Jonsson 2011a; Huang et al. 2011). This value of PrS is also **0.99838547** for lifetime of 1 year for the $\mathcal{B}$ fault model.

The proposed heuristic in Sect. 6 is then applied to the tasks of the Instrument Control application considering lifetime of 1 year and using parameters of the $\mathcal{B}$ fault model in Table 4. The resultant configuration for the active backups is $h_1 = 0$, $h_2 = 0$, $h_3 = 0$, $h_4 = 1$, and $h_5 = 0$. The PrS for this configuration is increased to **0.99999501**. This shows the effectiveness of the proposed heuristic in determining the number of active backups for each task of the Instrument Control application and its impact in improving PrS compared to earlier works.

8 Related works

There are numerous works in fault-tolerant real-time scheduling. Being fully aware of the impossibility to discuss all earlier works, this section presents the works closely

related to ours. Fault-tolerant partitioned scheduling traditionally aims to tolerate only permanent processor/core failures. Many of the earlier approaches to partitioned fault-tolerant scheduling proposed novel task allocation algorithms but assume a relatively restricted task or fault model in comparison to the models we consider in this paper. For example, the task allocation algorithms in Oh and Son (1994), Bertossi et al. (1999), Chen et al. (2007), Kim et al. (2010) assumed strictly periodic or aperiodic real-time tasks, only active backups and considered tolerating only processor failures.

Chen et al. (2007), Hashimoto et al. (2000) and Kim et al. (2010) considered a backup as the execution of the primary. In contrast, Oh and Son (1994) considered multiple backups for each task while Bertossi et al. (1999) considered exactly one backup that may be different from the primary. None of these works considered sporadic task model, do not explicitly address mechanisms to mask task errors, do not consider combining passive and active redundancy, and neither address probabilistic schedulability analysis.

Fault-tolerant scheduling of aperiodic tasks based on primary-backup approach is proposed in Ghosh et al. (1994), Tsuchiya et al. (1995). Instead of considering active backup, passive backup (Ghosh et al. 1994) or partially active backup (Tsuchiya et al. 1995) are also found to be effective for fault tolerance. Moreover, in order to efficiently utilize the processors, the scheduling algorithms in Ghosh et al. (1994) and Tsuchiya et al. (1995) consider backup-backup *overloading* and backup *deallocation*. The term "overloading" means that different backups of different tasks are overlapped in the same time interval on a processor. The term "deallocation" refers to reclaiming the processing resource given to a backup if the primary completes successfully.

Many interesting table-driven fault-tolerant scheduling are addressed in Girault et al. (2003), Isovic and Fohler (2000), Pop et al. (2009), Kandasamy et al. (2003). The scheduling table for each processor is generated offline, which sometimes could be challenging if task parameters are 'badly' related. However, table-driven scheduling provides high predictability but they are inflexible to changes since the schedule in each processor needs to regenerated if any task parameter is changed, for example, due to upgrade. Priority-driven (e.g., `FTM`) scheduling does not have this disadvantage.

The work by Girault et al. (2003) considers allocation of only active backups to tolerate processor failures and is not applicable to sporadic tasks. Isovic and Fohler (2000) consider time-triggered scheduling of sporadic tasks. In their approach, if the generated schedule is not fault-tolerant, then the schedule is discarded and new schedule is re-generated. However, Kandasamy et al. (2003) avoid this by directly generating the time-triggered fault-tolerant schedule. Pop et al. (2009) considered overhead-aware check-pointing for fault-tolerance in time-triggered scheduling. Huang et al. (2011) considered fault-tolerant schedule of jobs (not tasks) based on simple re-execution and rely on approximation (hence pessimistic) of a schedulability test that has exponential time complexity.

There are few works that addressed fault-tolerant global scheduling (Berten et al. 2006; Liberato et al. 1999; Pathan and Jonsson 2011a; Huang et al. 2011). Berten et al. (2006) proposed fault-tolerant EDF scheduling based on probabilistic analysis assuming strictly periodic tasks, deadline equals period, re-execution of the primary as only means of backup and does not consider processor failures. Each task τ_i is assumed to have t_i replicas and a probability p_i which is the probability that a replica fails. The

work in Berten et al. (2006) has several interesting results. First, sufficient number of processors that are required to meet all the deadlines of all the replicas of all tasks is computed based on a sufficient schedulability condition for global EDF scheduling. Second, the probability of meeting deadlines of at least one replica of each task over a given time frame is computed, which is the reliability of the system for that given time frame. Third, the minimum number of processors required to achieve a given target reliability is computed. Finally, the maximum reliability that can be achieved for a given number of processors is computed by considering different heuristics to determine the number of replicas for different tasks. Huang et al. (2011) considered migratory table-driven scheduling of only jobs (thus, not applicable to sporadic tasks) but considers tolerating both task error and core failures based on re-execution of the primary as the only means of backup.

Liberato et al. (1999) considers Earliest-Deadline-First (EDF) scheduling for tolerating a single fault on uniprocessor considering periodic tasks and derive an exact test that can be used to determine whether all the deadlines of all the periodic tasks are met or not. Liberato et al. (1999) extended the dynamic-priority-based P-fair (Baruah et al. 1996) scheduling for fault tolerance considering periodic task model, exactly one backup for each task (thus cannot tolerate multiple task errors affecting the same job) and assuming a minimum inter-arrival time between occurrences of consecutive task errors. The main idea is to execute tasks by artificially increasing their utilization to ensure that each task completes early enough to complete recovery if an error is detected in that task. A extra processor is deployed to execute the recovery so that schedulability guarantee to other tasks are not compromised.

Pathan and Jonsson (2011a) considered global fixed-priority scheduling where only passive backups are considered and do not address probabilistic analysis. A sufficient schedulability condition is presented that can be used to determine if all the tasks meet their deadlines. The use of only passive backups imply that at most one backup executes at a time and does not exploit the benefit of parallel architecture to recover quickly from errors using active backups. The works in Huang et al. (2011) and Pathan and Jonsson (2011a) are based on a fault model that assumes that each job can suffer the *same* number of faults in its scheduling window. This is unreasonable since the scheduling windows of different jobs may have different length. And, job with relatively longer scheduling window and longer WCET may suffer relatively larger number of faults. To the best of our knowledge, no partitioned or global scheduling considered fault burst model as we have considered in this paper.

A similar fault model is considered for uniprocessor platform in Pathan and Jonsson (2010b) where each job can suffer the same number of faults in its scheduling window and an exact schedulability condition for sporadic tasks is proposed. Similar to the work in Pathan and Jonsson (2010b), this paper presents a deadline-based analysis in which the load in the scheduling window of each task is computed and compared against the deadline to determine if the task meets its deadline or not. One interesting future work is to extend the analysis of this work to derive a response-time based schedulability analysis.

In summary, many of the previous works on fault-tolerant scheduling on multiprocessors consider a relatively restricted error/fault model by assuming, for example, that (i) the inter-arrival time of two errors/faults must be separated by a minimum

distance (Ghosh et al. 1994; Tsuchiya et al. 1995; Liberato et al. 1999), (ii) at most one error affect each task (Liberato et al. 1999; Ghosh et al. 1994; Bertossi et al. 1999), (iii) tolerating task error is same as tolerating processor failures (Oh and Son 1994; Bertossi et al. 1999; Chen et al. 2007; Kim et al. 2010; Girault et al. 2003), and (iv) consider only active backups (Girault et al. 2003) or only passive backups (Pathan and Jonsson 2011a).

To the best of our knowledge, there is no work that addresses fault-tolerant global scheduling that considers sporadic task model, deadline of the tasks being less than or equal to the periods, considers both core failures and task errors using multiple and possibly different backups, provides probabilistic guarantee for different fault models, and combines advantages of both active and passive redundancy. Our proposed FTM scheduling algorithm presented in this paper possesses all these characteristics.

9 Conclusion

This paper presents the design and analysis of a scheduling algorithm FTM that considers application-level error level which is independent of the stochastic behavior of the hardware-level fault model. The schedulability analysis is performed to determine the application's ability to tolerate the number of core failures and job errors for each task. And, the actual fault model's stochastic behavior is plugged in to the analysis later to perform a probabilistic analysis to determine the application's ability to withstand actual faults during run-time. Such separation of concern between the "error model" and "fault model" enables to derive the probabilistic schedulability guarantee of the same application under different faulty environments.

The other major strengths of FTM algorithm are that it considers (i) combination of both active and passive redundancy, (ii) both job errors and core failures, (iii) use backups that may be different from the primary, (iv) probabilistic schedulability guarantee for different fault models, and (v) proposing an effective heuristic to determine the number of active backups for each task. The methodology presented in this paper can be applied to determine the probability of meeting all real-time constraints of safety-critical application operated in faulty multicore environment.

Although hardware faults are considered, the FTM scheduling is also applicable for tolerating software faults relying on the execution of diverse backups. Based on the exponential failure law for software faults, as is proposed by Sha (2001), we can extend our fault models by including the stochastic behavior of software faults and can apply the same probabilistic analysis presented in this paper.

We assume that when a core fails, no task can execute on that core. However, a core may be faulty only for certain tasks and may be non-faulty for other tasks. For example, if the floating-point unit (FPU) of a core is not working, then all tasks that do not use FPU can be executed on that core. Extending the analysis of this paper for task-specific permanent hardware faults is left as a future work.

Acknowledgements This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA.

Appendix: Proof of Theorem 1

Proof We will show that if total $(\widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}})$ units of CPU time is available, then the z^{th} backup has enough CPU time to complete its E_k^z units of execution in FTM scheduling for $z = 0, 1, \ldots h_k$. From Eq. (9), we have that

$$\widehat{\mathtt{M}} \cdot s_{k,\widehat{\mathtt{M}}} \geq \widehat{\mathtt{M}} \cdot E_k^z + \sum_{b=0}^{z-1} E_k^b \tag{20}$$

for $z = 0, 1, \ldots h_k$. It is evident from Eq. (20) that the total CPU time available on $\widehat{\mathtt{M}}$ cores is at least $(\widehat{\mathtt{M}} \cdot E_k^z + \sum_{b=0}^{z-1} E_k^z)$ for $z = 0, 1, \ldots h_k$.

In FTM scheduling, the 0^{th}, 1^{st}, $\ldots$ $(z-1)^{th}$ backups of job J_k have higher priorities over the z^{th} backup for $z = 0, 1, \ldots h_k$. Therefore, when considering whether the z^{th} backup has enough CPU time or not, we have to consider the amount of CPU time required for the higher priority 0^{th}, 1^{st}, $\ldots$ $(z-1)^{th}$ backups and the z^{th} backup. The total CPU time required for the $(z-1)$ higher priority backups is $\sum_{b=0}^{z-1} E_k^b$. Consequently, the minimum CPU time available for the z^{th} backup is $(\widehat{\mathtt{M}} \cdot E_k^z)$ according to Eq. (20).

The amount of sequential execution time available for the z^{th} backup is minimized when this available CPU time of $(\widehat{\mathtt{M}} \cdot E_k^z)$ units are evenly divided across all the $\widehat{\mathtt{M}}$ cores. In other words, at least $\frac{\widehat{\mathtt{M}} \cdot E_k^z}{\widehat{\mathtt{M}}} = E_k^z$ units of sequential execution is possible for the z^{th} backup, which is enough for z^{th} backup to complete its execution for $z = 0, 1, \ldots h_k$.

References

Aggarwal N, Ranganathan P, Jouppi NP, Smith JE (2007) Configurable isolation: building high availability systems with commodity multi-core processors. In: Proceedings of the ISCA

Al-Asaad H, Murray BT, Hayes JP (1998) Online BIST for embedded systems. IEEE Des Test 15(4):17–24. doi:10.1109/54.735923

Andersson B, Bletsas K, Baruah S (2008) Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In: Proceedings of the IEEE real-time systems symposium, pp 385–394

Avižienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Depend Secur Comput 1(1):11–33. doi:10.1109/TDSC.2004.2

Barlow RE, Heidtmann KD (1984) Computing k-out-of-n system reliability. IEEE Trans Reliab 33(4):322–323. doi:10.1109/TR.1984.5221843

Baruah S, Rosier LE, Howell RR (1990) Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Syst 2(4):301–324

Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1996) Proportionate progress: a notion of fairness in resource allocation. Algorithmica 15(6):600–625

Baruah S, Burns A, Davis R (2011) Response-time analysis for mixed criticality systems. In: Proceedings of RTSS

Baruah S, Fohler G (2011) Certification-cognizant time-triggered scheduling of mixed-criticality systems. In: Proc RTSS, pp 3–12

Baumann R (2005) Soft errors in advanced computer systems. IEEE Des Test Comput 22(3):258–266
Berten V, Goossens J, Jeannot E (2006) A probabilistic approach for fault tolerant multiprocessor real-time scheduling. In: Proceedings of IPDPS, p 8
Bertossi AA, Mancini LV, Rossini F (1999) Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. IEEE Trans Parallel Distrib Syst 10(9):934–945. doi:10.1109/71.798317
Borkar S (2005) Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro 25(6):10–16. doi:10.1109/MM.2005.110
Brandenburg BB, Calandrino JM, Anderson JH (2008) On the scalability of real-time scheduling algorithms on multicore platforms: a case study. In: Proceedings of RTSS, pp 157–169
Campbell A, McDonald P, Ray K (1992) Single event upset rates in space. IEEE Trans Nucl Sci 39(6):1828–1835. doi:10.1109/23.211373
Carpenter J, Funk S, Holman P, Anderson JH, Baruah S (2004) A categorization of real-time multiprocessor scheduling problems and algorithms. Methods, and Models, Handbook on Scheduling Algorithms
Chen J-J, Yang C-Y, Kuo T-W, Tseng S-Y (2007) Real-Time Task Replication for Fault Tolerance in Identical Multiprocessor Systems. In: Proceedings of RTAS, pp 249–258
Claesson V (2002) Efficient and reliable communication in distributed embedded systems. Ph.D. Thesis, Chalmers University of Technology, Gothenburg
Davis R, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv 43(4):35:1–35:44
Elliott EO (1963) Estimates of error rates for codes on burst-noise channels. Bell Syst Tech J 42(5):1977–1997
Ferreira J, Oliveira A, Fonseca P, Fonseca J (2004) An experiment to assess bit error rate in CAN. In: Proceedings of 3rd international workshop of real-time networks
Garey MR, Johnson DS (1979) Computers and Intractability: a guide to the theory of NP-completeness. W. H. Freeman & Co., New York
Ghosh S, Melhem R, Mosse D (1994) Fault-tolerant scheduling on a hard real-time multiprocessor system. In: Proceedings of the Parallel Processing Symposium, pp 775–782
Girault A, Kalla H, Sorel Y, Sighireanu M (2003) An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In: Proceedings of the DSN
Hashimoto Koji, Tsuchiya Tatsuhiro, Kikuno Tohru (2000) A new approach to fault-tolerant scheduling using task duplication in multiprocessor systems. J Syst Softw 53(2):159–171
Hiller M (2000) Executable assertions for detecting data errors in embedded control systems. In: Proceedings of the DSN
Hong Y (2013) On computing the distribution function for the poisson binomial distribution. Comput Stat Data Anal 59(C):41–51. http://ideas.repec.org/a/eee/csdana/v59y2013icp41-51.html
Huang J, Blech JO, Raabe A, Buckl C, Knoll A (2011) Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In: Proceedings of the CODES+ISSS, pp 247–256
Isovic D, Fohler G (2000) Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In: Proceedings of the RTSS, pp 207–216
Jalote P (1994) Fault tolerance in distributed systems. Prentice Hall, Upper Saddle River
Jhumka A, Hiller M, Claesson V, Suri N (2002) On systematic design of globally consistent executable assertions in embedded software. In: Proceedings of the LCTES
Joseph R, Brooks D, Martonosi M (2003) Control techniques to eliminate voltage emergencies in high performance processors. In: Proceedings of the 9th international symposium on high-performance computer architecture, HPCA '03
Kandasamy N, Hayes JP, Murray BT (2003) Transparent recovery from intermittent faults in time-triggered distributed systems. IEEE Trans Comput 52(2):113–125. doi:10.1109/TC.2003.1176980
Kato S, Yamasaki N (2009) Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: Proceedings of the EuroMicro conference on real-time systems, pp 249–258
Kim J, Lakshmanan K, Rajkumar R (2010) R-BATCH: task partitioning for fault-tolerant multiprocessor real-time systems. In: Proceedings of ICESS, pp 1872–1879
Koren I, Krishna CM (2007) Fault-tolerant systems. Morgan Kaufmann, ISBN 0-12-088525-5
Lakshmanan K, Rajkumar R, Lehoczky J (2009) Partitioned fixed-priority preemptive scheduling for multi-core processors. In: Proceedings of the EuroMicro Conference on Real-Time Systems, pp 239–248
Liberato F, Lauzac S, Melhem R, Mossé D (1999) Fault tolerant real-time global scheduling on multiprocessors. Proceedings of ECRTS, p 252

Many F, Doose D (2011) Scheduling analysis under fault bursts. In: 17th IEEE Real-time and embedded technology and applications symposium (RTAS), pp 113–122
Meixner A, Bauer ME, Sorin DJ (2008) Argus: low-cost, comprehensive error detection in simple cores. IEEE Micro 28(1):52–59. doi:10.1109/MM.2008.3
Oh Y, Son SH (1994) Enhancing fault-tolerance in rate-monotonic scheduling. Real-Time Syst 7(3):315–329. doi:10.1007/BF01088524
Pathan RM, Jonsson J (2010a) Load regulating algorithm for static-priority task scheduling on multiprocessors. In: Proceedings of the IPDPS, pp 1 –12
Pathan RM, Jonsson J (2010b) Exact fault-tolerant feasibility analysis of fixed-priority real-time tasks. In: International workshop on real-time computing systems and applications, pp 265–274
Pathan RM, Jonsson J (2011a) FTGS: Fault-tolerant fixed-priority scheduling on multiprocessors. In: Proceedings of ICESS
Pathan RM, Jonsson J (2011b) Improved schedulability tests for global fixed-priority scheduling. In: Proceedings of ECRTS
Pathan RM (2014) Fault-tolerant and real-time scheduling for mixed-criticality systems. Real-Time Syst 50(4):509–547
Pop P, Izosimov V, Eles P, Peng Z (2009) Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. IEEE Trans VLSI Syst 17(3):389–402. doi:10.1109/TVLSI.2008.2003166
Ras J, Cheng AMK (2010) Response time analysis of the abort-and-restart model under symmetric multiprocessing. In: Proceedings of CIT, pp 1954–1961
Sha L (2001) Using simplicity to control complexity. IEEE Softw 18(4):20–28. doi:10.1109/MS.2001.936213
Sherwood T, Sair S, Calder B (2003) Phase tracking and prediction. SIGARCH Comput Archit News 31(2):336–349
Shivakumar P, Kistler M, Keckler SW, Burger D, Alvisi L (2002) Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings of the DSN, pp 389–398
Short M, Proenza J (2013) Towards efficient probabilistic scheduling guarantees for real-time systems subject to random errors and random bursts of errors. In: Proceedings of ECRTS, pp 259–268. doi:10.1109/ECRTS.2013.35
Srinivasan J, Adve SV, Bose P, Rivers JA (2004) The impact of technology scaling on lifetime reliability. In: Proceedings of the DSN
Tsuchiya T, Kakuda Y, Kikuno T (1995) Fault-tolerant scheduling algorithm for distributed real-time systems. In: Proceedings of the workshop on parallel and distributed real-time systems, p 99
Wong HC, Burns A (2014) Schedulability analysis for the abort-and-restart (ar) model. In: Proceeding of RTNS, 2014