eng

MDPI

Article

FTI-TMR: A Fault Tolerance and Isolation Algorithm for Interconnected Multicore Systems

Yiming Hu and Chao Wang *

School of Computer Science, Beijing Information and Technology University, No. 12, Qinghe Xiaoying East Road, Haidian District, Beijing 100192, China; hym13526@163.com

* Correspondence: wangchao@bistu.edu.cn

Abstract

Two-Phase TMR conserves energy by partitioning redundancy operations into two stages and making the execution of the third task copy optional, yet it remains susceptible to permanent faults. Reactive TMR (R-TMR) counters this by isolating faulty cores, handling both transient and permanent faults. However, the lightweight hardware required by R-TMR not only increases complexity but also becomes a single point of failure itself. To bypass isolated node constraints, this paper proposes a Fault Tolerance and Isolation TMR (FTI-TMR) algorithm for interconnected multicore systems. We construct a stability metric characterized by its prior and posterior estimates to identify the most reliable nodes in the system. These nodes then perform periodic diagnostics to isolate permanent faults. The experimental results show that FTI-TMR reduces task workload by approximately 30% compared with baseline TMR, while achieving higher permanent-fault coverage.

Keywords: fault tolerance and isolation; interconnected systems; diagnostics; permanent fault; multicore systems

1. Introduction

With continuous growth in both the number and complexity of processor cores, ensuring computational reliability in the presence of transient and permanent faults has become a fundamental requirement for hard real-time and safety-critical applications [1,2]. Triple Modular Redundancy (TMR) addresses this need by executing three identical copies of a task and using majority voting on their outputs to determine the system's final result.

However, TMR inherently incurs roughly three times the energy consumption of a non-redundant system. Recent research has increasingly focused on improving fault tolerance while minimizing energy overhead. It has also explored the trade-offs between these two objectives [3]. Various TMR variants have been proposed to achieve this balance [4,5]. For instance, ref. [4] introduces a Two-Phase TMR scheme in which the third copy can be skipped if the outputs of the first two copies match. Building on this concept, ref. [5] presents an energy-efficient Reactive TMR (R-TMR) approach that detects and isolates cores with permanent faults using lightweight hardware mechanisms and optimized scheduling policies.

Despite these advancements, TMR and its variants are largely restricted to the "isolated node" paradigm, relying on redundant resources within a single computing device. This paradigm presents inherent limitations: the added hardware (e.g., detection logic in R-TMR) increases design complexity and introduces new single points of failure. Moreover, if a node suffers multiple permanent core failures or significant resource degradation, its fault

Academic Editors: Maki Habib and Fusaomi Nagata

Received: 15 May 2026
Revised: 28 July 2026
Accepted: 31 July 2026
Published: 6 August 2026

tolerance capability deteriorates rapidly. In essence, the reliability ceiling of these methods is bound by the hardware resources of individual nodes.

The thriving interconnected systems paradigm provides a new opportunity to overcome these limitations. According to the report in [6], the Machine-to-Machine (M2M) Connections Market is valued at USD 40.7 billion in 2026. A significant portion of these interconnected systems relies on multicore processors and graphics processing units (GPUs) to meet the ever-growing demands for performance and energy efficiency [7]. However, as transistor dimensions continue to shrink and device densities increase, these components have become increasingly susceptible to aging-related degradation mechanisms, including bias temperature instability (BTI), hot-carrier injection (HCI), and electromigration [8]. The cumulative impact of these phenomena can result in permanent hardware faults.

Motivated by this, this paper proposes a dynamic, adaptive, fault-tolerant algorithm for interconnected systems: the Fault Tolerance and Isolation TMR (FTI-TMR) algorithm. The key idea is to shift fault detection from reliance on dedicated hardware to a dynamically allocatable "service" distributed across nodes. Based on a stability metric with prior and posterior estimates, the two most stable nodes are selected to periodically perform fault detection. This metric-driven, lightweight mechanism eliminates the need for additional hardware, avoids single points of failure, and can adaptively manage scenarios involving multiple node failures, thereby significantly enhancing overall system robustness.

The experimental results show that, compared with Conventional TMR techniques, FTI-TMR reduces task workload by approximately 30%. Compared to the state-of-the-art R-TMR [5], FTI-TMR achieves higher permanent-fault coverage. The main contributions of this work are as follows:

- We developed and provided open-source access to a TMR fault tolerance simulation tool based on the Standard Task Graph (STG) dataset, simplifying the data collection and comparative experimentation process.
- We extended the experiments on Reactive TMR and demonstrated its fault tolerance limit.
- We proposed a prior–posterior stability metric suitable for multicore devices under TMR.
- We designed a dynamic, fault-tolerant algorithm for interconnected systems capable of tolerating transient faults and isolating permanent faults.

2. Background and Related Work

In real-time systems, fault tolerance mechanisms are among the key technologies for ensuring system reliability. Redundant execution is a widely adopted fault tolerance strategy, the core idea of which is to introduce additional computational resources to improve a system's error tolerance. N-Modular Redundancy (NMR) achieves fault tolerance by executing multiple copies of a task in parallel and using a majority vote to produce the final output. By masking errors in individual copies, this approach helps maintain correct system operation. NMR can be implemented via hardware redundancy (e.g., replicating multiple circuit modules) or software means (e.g., running multiple program instances).

Triple Modular Redundancy (TMR), as the most representative form of the NMR scheme, can effectively tolerate single-point failures through the execution and voting mechanism of three redundant copies. TMR is widely used in high-reliability scenarios such as spacecraft and nuclear power plant control systems. Under the majority voting mechanism, a TMR system requires at least two correct copies to maintain system functionality. Numerous studies [4,5,9] have highlighted TMR's effectiveness in improving the reliability of hard real-time systems. The TMR and its variants have always focused on

an isolated node paradigm, where all redundant resources are contained within a single computing device.

Recently, the thriving of interconnected systems has become a defining characteristic of modern computing infrastructures; from large-scale data centers and cloud platforms to embedded and edge devices, interconnected computing nodes now support unprecedented parallelism and resource sharing.

Despite the existence of several studies proposing fault-detection algorithms for interconnected systems, most of them remain theoretical. Their primary objective is to demonstrate, under specific system models and detection schemes, how many faulty nodes can be detected given a certain number of fault-free nodes [10–13].

This section provides a short overview of TMR-related algorithms in multicore environments, covering Conventional TMR (Section 2.1), Two-Phase TMR (Section 2.2), Enhanced Two-Phase TMR (Section 2.3), and Reactive TMR (Section 2.4). The first three approaches are implemented entirely in software, whereas Reactive TMR requires extra hardware to detect and isolate permanent faults. Section 2.5 introduces the Raft consensus algorithm from distributed systems, which serves as a key source of inspiration for the core concepts developed in this paper. Section 2.6 presents several works regarding fault detection in interconnected systems. In addition, Section 2.7 presents a classical reliability estimate.

2.1. Conventional TMR

Conventional TMR is the most fundamental implementation. Its execution process is illustrated in Figure 1a. The system executes three copies of a task and determines the final output by comparing their results. If one copy's result diverges from the other two, that copy is identified as faulty, and its output is discarded. System failure occurs only if all three copies produce mutually inconsistent results. However, in practice, the probability of concurrent failures in two or more copies (in the absence of permanent faults) is exceedingly low [14]. Thus, Conventional TMR effectively maintains system functionality in most scenarios.

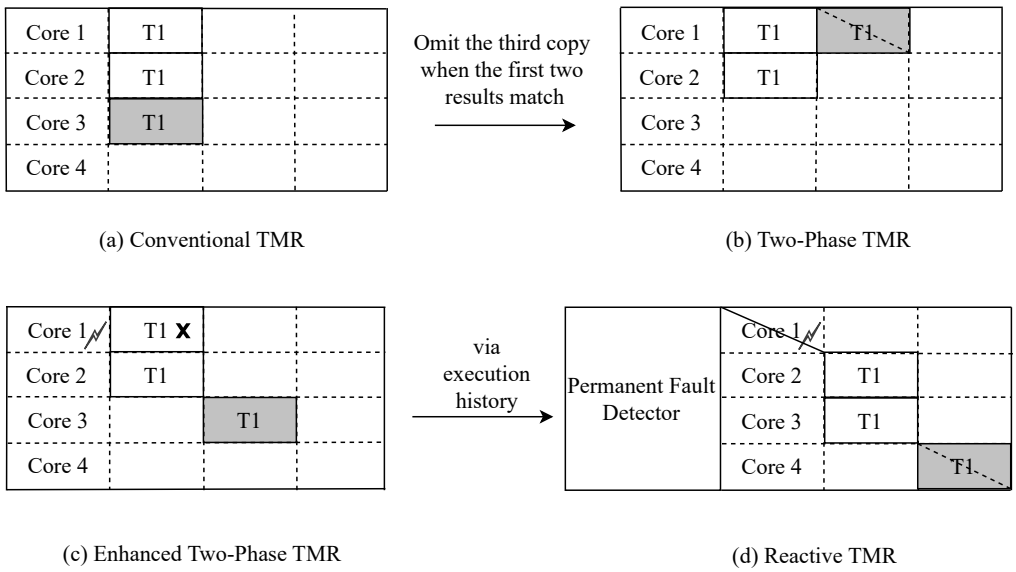

Figure 1. TMR schemes: (**a**) Conventional TMR: three task copies must be executed. (**b**) Two-Phase TMR: the third copy can be omitted under normal conditions. (**c**) Enhanced Two-Phase TMR: task copies are distributed across different cores, allowing tolerance to permanent core faults. (**d**) Reactive TMR: based on Enhanced Two-Phase TMR, a permanent-fault detector is introduced to disable the use of faulty cores according to execution history.

2.2. Two-Phase TMR

Conventional TMR execution triples the workload of a non-redundant baseline system, introducing significant energy overheads. Because transient faults are relatively rare in practice, Two-Phase TMR divides the traditional TMR execution workflow into two phases: a mandatory phase and an on-demand phase [4]. Two copies of each task are scheduled for execution in the mandatory phase, while the third copy, reserved for the on-demand phase, is executed only if needed.

During the mandatory phase, the system executes the two task copies and compares their results. If the outputs agree, the third task copy is skipped, thereby conserving energy. Since transient faults are uncommon [15], the results of the two mandatory copies typically match. This allows Two-Phase TMR to omit the on-demand phase in most execution cycles, leading to substantial overall energy savings. Figure 1b illustrates an example of task execution under Two-Phase TMR.

2.3. Enhanced Two-Phase TMR

Two-Phase TMR does not specify how task copies are allocated across processor cores. Two task copies could be executed serially on a single core. If executed serially on one core, the system's fault tolerance is compromised if that specific core experiences a permanent fault [5]. As shown in Figure 1c, Enhanced Two-Phase TMR addresses this by distributing the three task copies across three distinct cores for execution, thereby tolerating permanent faults affecting any single core.

2.4. Reactive TMR

Although Enhanced Two-Phase TMR tolerates single-core permanent faults, it increases the likelihood that the third task copy cannot be omitted. Building upon Enhanced Two-Phase TMR, Reactive TMR (R-TMR), illustrated in Figure 1d, introduces additional lightweight hardware for permanent-fault detection. When it detects a core consistently producing erroneous outputs over multiple voting rounds, R-TMR classifies that core as a permanently faulty core. Subsequently, a task migration mechanism reassigns tasks from the failed core to healthy ones, maintaining system correctness and availability.

Although prior work primarily demonstrated R-TMR's capability for detecting and isolating a single faulty core [5], our experiments in Section 4.3 show that, provided the permanent-fault detection hardware remains functional, R-TMR can detect and isolate any number of permanent faulty cores within the system as long as at least two fault-free cores are available.

Despite its advantages, R-TMR has two limitations. First, the algorithm fails rapidly when fewer than two fault-free cores are available. Second, its reliance on additional hardware fault-detection units (including fault history arrays, flag bits, backup scheduling policy storage, etc.) introduces a potential single point of failure; malfunction of this unit could potentially induce more severe system failures. Section 3.1 discusses the limitations of R-TMR in more detail.

2.5. Raft

Raft is a consensus algorithm designed for managing replicated logs in distributed systems, with the primary design goal of being easy to understand [16]. Unlike conventional algorithms, like Paxos [17], known for their complexity and difficulty in practical implementation, Raft significantly reduces engineering complexity and cognitive load by decomposing the consensus problem into three relatively independent subproblems: leader election, log replication, and safety.

In the Raft architecture, any server node is in one of three states at any given time: leader, follower, or candidate. Its operation centers around a strong leader model: for a given term, one and only one leader is elected. This leader is responsible for receiving all client requests and managing the log replication process as shown in Figure 2. This centralized management simplifies the algorithm's logic, ensuring log entries are appended unidirectionally from the leader to followers, avoiding complex conflict resolution mechanisms.

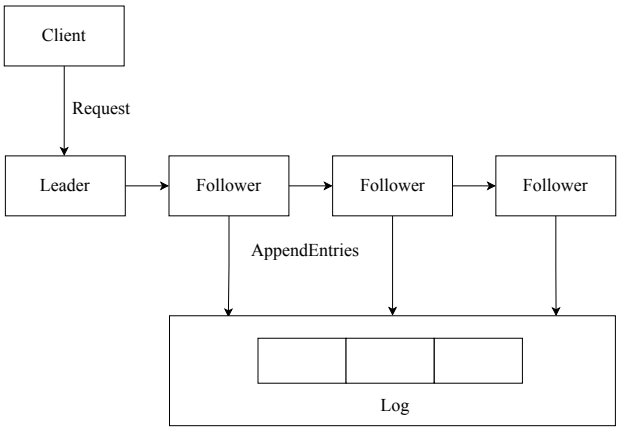

Figure 2. Raft consensus algorithm.

Raft not only provides an efficient and reliable consensus solution but has also become the de facto standard in industry for building distributed systems [18,19], due to its clear modular design and strong leader model. The success of Raft and TCP/IP, compared to more complex alternatives such as Paxos or the ISO protocol stack, suggests that simplicity and understandability can sometimes outweigh theoretical sophistication, making protocols more practical for real-world deployment. The fault-tolerance mechanism proposed in this paper is deeply inspired by Raft's approaches to strong leader modeling and state consistency.

2.6. Fault Detection in Interconnected Systems

One work proposed a distributed fault-detection method tailored for large-scale, nonlinear uncertain systems [20]. By decomposing a large system into several subsystems that allow overlapping states, this approach utilizes local fault diagnosers (LFDs) combined with adaptive approximation techniques to dynamically estimate the physical interconnections and uncertainties between adjacent subsystems. This "divide-and-conquer" paradigm permits localized information exchange. Recently, two notable works have advanced the field of fault diagnosis in different directions. The first addresses nonlinear discrete-time stochastic systems by combining particle filtering with a hidden Markov model, where fault modes are represented as binary states embedded in the system dynamics, allowing fault-indicative particles to proliferate when faults occur [21]. The second focuses on islanded microgrids comprising multiple interconnected distributed generation units, proposing a Plug-and-Play fault-detection approach [22].

However, when deploying such mechanisms in concrete engineering scenarios and system-level applications, resource constraints and dynamic workloads often degrade the performance of traditional detection mechanisms. Another work systematically evaluated timeout-based fault-detection mechanisms under resource stress (e.g., CPU overload, memory contention) across distributed edge nodes [23]. Their findings demonstrated that conventional static timeout thresholds are extremely fragile, leading to a false-positive rate of up to 30% during fault-free operations while simultaneously masking performance degradation when real faults occur due to overly conservative tuning. This observation fur-

ther highlights the urgency of our research: relying solely on time-triggered or hard-coded thresholds is inadequate for handling complex, interconnected multicore environments.

Furthermore, the GridAI system was proposed to maintain high fault coverage while reducing the computational and communication overheads associated with diagnosis in scenarios where sensing devices report to monitoring servers [24]. It deploys an intelligent fault-detection algorithm based on XGBoost at the edge and innovatively incorporates a goal-oriented semantic communication strategy. This approach reduces communication overheads by three orders of magnitude while preserving sub-second inference and high accuracy. Unlike GridAI, which uses continuous edge-based detection to monitor transformers, FTI-TMR nodes leverage the CPU's multiple cores for redundancy and periodically execute autonomous fault detection within the system.

In summary, existing distributed fault detection studies generally emphasize either sophisticated nonlinear continuous/discrete theoretical modeling or data-driven empirical approaches, including threshold tuning and intelligent classification models at the edge. Distinct from these methodologies, the FTI-TMR algorithm proposed in this paper specifically targets the discrete task execution characteristics of interconnected multicore systems. It provides tolerance against transient faults and isolation of permanent faults, which together constitute the fault tolerance and isolation (FTI) capability. FTI-TMR is not a replacement for existing paradigms, but might be a complement to them: it could work with existing Edge AI methodologies and hierarchical subsystem decomposition paradigms, enabling a more practical and scalable fault-management framework for large-scale interconnected systems.

2.7. Reliability

Device reliability is conventionally defined as a function of time, such as the exponential distribution function or the Bathtub curve function. Figure 3 shows the Bathtub curve.

$$f(t) = (\frac{k}{\lambda})(\frac{t}{\lambda})^{k-1} e^{-(\frac{t}{\lambda})^k}. \quad (1)$$

There are many ways to fit this curve [25–28]. Equation (1) shows the Weibull probability density function. λ is the time scale parameter and k is the shape parameter. It can model three different phases depending on the value of k ($k < 1$ is an infant mortality phase, $k = 1$ is useful life, and $k > 1$ is in the wear-out phase). A Weibull plot can be used to estimate changes in the failure rate [28]. For a new device, this serves primarily as a prior estimate.

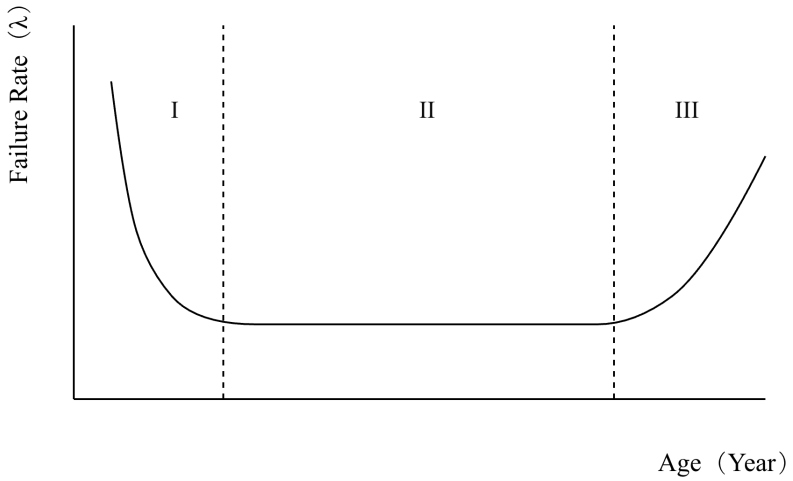

Figure 3. Bathtub curve: *I*, infant mortality phase; *II*, useful life; *III*, wear-out phase.

3. The Proposal

This section presents a fault-tolerant architecture for interconnected multicore systems, capable of addressing both transient and permanent faults. The core idea of our approach is to assess the relative reliability of processing units based on their historical and runtime behavior, to select two highly stable computation nodes to perform fault detection on the remaining nodes, and then to notify them to isolate the permanent fault cores. We begin this section with an example: Section 3.1 highlights a specific scenario that the recently proposed Reactive TMR (R-TMR) fails to handle. We then describe our method in detail, covering the following topics: the system model (Section 3.2), the handling of transient faults (Section3.3), the stability metrics (Section 3.4), periodic permanent fault detection and isolation (Section 3.5), and the formal correctness of FTI-TMR (Section 3.6).

3.1. Motivation

Figure 4 illustrates the execution flow of a hypothetical task, T1, under R-TMR. When nodes Core 1, Core 2, and Core 3 experience permanent faults, as shown in Figure 4a, the R-TMR system fails if fewer than two fault-free cores remain. Because Core 1 and Core 2 are faulty, their results after T1's execution are inconsistent with those of Core 3, prompting the system to switch to on-demand execution of the third copy. However, since Core 3 is also faulty, executing T1 on it still produces inconsistent results. The system completely fails and cannot identify the faulty cores.

Figure 4b shows an even more severe situation: if the permanent-fault-detection unit is broken, the healthy cores Core 1 and Core 2 may be unexpectedly shut down, while the faulty Core 3 remains active. Then, the system cannot assign the third task copy to a different core. If two copies of T1 are executed on the faulty cores, correct voting cannot be achieved. In such a scenario, the entire algorithm inevitably fails.

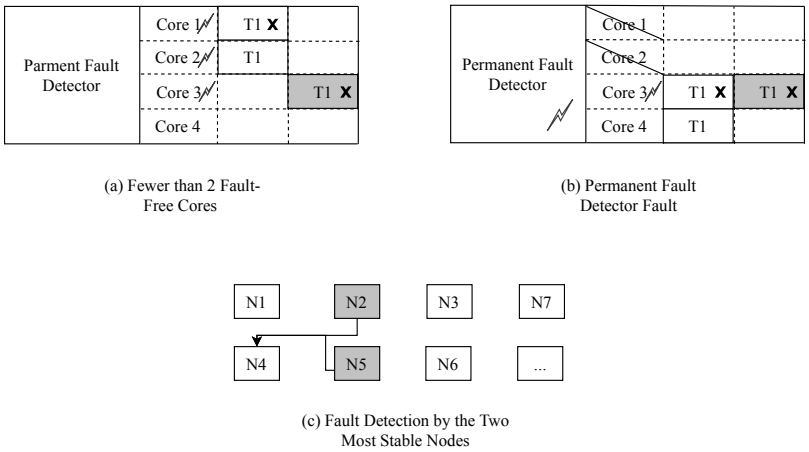

Figure 4. Limitations of R-TMR and improvement concept: (**a**) the system fails when there are fewer than 2 fault-free cores or (**b**) when the additional fault-detection hardware becomes defective. Improvement concept: (**c**) the two most stable nodes are selected to perform permanent-fault detection on the other nodes.

In light of this, we propose an improved approach that does not rely on additional hardware units for fault detection and isolation. As illustrated in Figure 4c, within an interconnected system, the system periodically votes to select two more stable primary and secondary leader nodes, which perform fault detection on the other nodes and instruct the healthy cores to isolate the cores with permanent faults.

3.2. The System Model

We consider an interconnected system comprising multicore nodes, N_1, N_2, and $N_3 \ldots N_n$. The system can be either homogeneous or heterogeneous. Each node is assumed to possess at least one CPU, and each CPU contains a minimum of three cores. During normal operation, each node can independently run different applications.

A process is bound to each core on every node. These processes enable inner-core communication within a node and inter-node communication across the system. $Core_{ij}$ denotes the j-th core of node N_i. For instance, communication between $Core_{12}$ and $Core_{23}$ signifies communication between the process on the second core of node N_1 and the process on the third core of node N_2. We assume that inter-node communication is based on the TCP protocol, employing encrypted channels that are secure, fault-free, and reliable. The disabling of cores within a node is formally defined under Item 5 in Section 3.6.

Both transient faults (e.g., induced by radiation or noise) and permanent faults (e.g., caused by aging or physical damage) may occur unpredictably on any core. The FTI-TMR algorithm proposed in this paper primarily targets both transient and permanent faults within interconnected multicore systems. We assume that cores with faults in the system are honest but faulty. This means a faulty core might exhibit non-malicious failure behaviors such as crashing, timing out, sending messages with outdated states, or producing incorrect computational results. It will not actively collude with other cores to send malicious or contradictory messages intended to deceive the system (which is known as the Byzantine generals problem [29]). We define this as an Accidental-Behavior Fault Tolerance (ABFT) model. The reliable operation of the system relies on the fundamental premise that, for n nodes in the system, $\lfloor \frac{n}{2} \rfloor + 1$ nodes remain fault-free.

3.3. Handling Transient Faults

To handle transient faults during the operation of a node, N_i, we employ Enhanced Two-Phase TMR (TP-TMR+, see Section 2.3). This approach distributes task copies across distinct cores for execution and voting. For directed acyclic graph (DAG)-based tasks, a list scheduling approach [30] with the Longest Task First (LTF) [31] scheduling strategy is adopted, while for independent discrete tasks, the First-Come-First-Served (FCFS) policy is applied.

A disputed task is defined as one that triggers the on-demand phase due to inconsistent outputs during majority voting. When such a task is identified, the core, $Core_{ij}$, increments its dispute counter, D_i, and stores the task along with its input data in a local disputed task list, denoted as $DTList_i$. It should be noted that, in a node with four cores, for instance, each core maintains its own copy of the disputed task data—resulting in four separate yet identical records across the node. The structure of $DTList_i$ is detailed in Table 1, where $DTList_i[0]$ corresponds to the disputed tasks recorded on the first core.

Table 1. The data structure of disputed task list, $DTList$.

```
DTList = [
  [{
    task, // Disputed task
    input, // task input data on Ni
    passed, // passed test or not
    failedCount // used to exponential backoff
  }, ...], // Core 1
  [...], // Core 2
  [...], // Core 3
  ...
]
```

Both the dispute counter, D_i, and the dispute list, $DList_i$, are utilized in the computation of the node stability metric and support permanent-fault isolation.

In scenarios where a node has degraded to fewer than three active cores—making it impossible to allocate each task copy to a distinct core—the system continues to distribute task copies across the remaining cores as evenly as possible. For example, with three tasks and only two functioning cores, two tasks will inevitably be executed on the same core.

3.4. Stability Metrics

Equation (2) quantifies the general stability of any device. *Prior_SS* provides a prior estimate. *Post_SS*, which can be estimated from historical data, represents a posterior estimate. The stability score, $SS \in [0, 1]$, is defined such that a higher value indicates greater stability. Given that $\alpha + \beta = 1$, in the absence of historical failure data, one can increase α while decreasing β, or vice versa, to reflect different prior beliefs.

$$\begin{gathered} SS = \alpha \cdot Post\_SS + \beta \cdot Prior\_SS, \\ (\alpha + \beta = 1, Post\_SS \leq 1, Prior\_SS \leq 1) \end{gathered} \tag{2}$$

To identify nodes with long-term operational stability under TMR scenarios, for each node, N_i, the total number of tasks executed is denoted as S_i, and the number of disputed tasks generated is denoted as D_i. Let f_i and F_i represent the actual value and the maximum value, respectively, of other prior factors influencing stability. This mechanism is based on the stability score (SS) computed for each node:

$$SS_i = \alpha \frac{S_i - D_i}{S_i} + \beta \frac{f_i}{F_i}, (\alpha + \beta = 1) \tag{3}$$

Equation (3) quantifies the operational stability of device N_i. Each core records the relevant parameters (D_i, f_i, etc.) for this computation. Let θ_i denote the true stability of node N_i. We have a belief that more stable nodes should generate fewer disputed tasks. This belief is reflected in our choice of $\frac{S_i - D_i}{S_i}$ to serve as a posterior correction term, while $\frac{f_i}{F_i}$ serves as a prior proxy for θ_i. For example, $f_i = e^{-(\frac{t}{\lambda})^k}$ and $F_i = 1$, as introduced in Section 2.7.

3.5. Periodic Detection and Isolation of Permanent Faults

The periodic detection and isolation of permanent faults is initiated by each node with period t, where t may be a fixed or variable duration. This process comprises two main components: primary and secondary leader election and permanent-fault detection and isolation.

3.5.1. Primary and Secondary Leader Election

For a node, N_i, within the system, each local core records a round number, *term*, during the permanent-fault isolation phase. The *term* starts from 0 and increments by 1 each time a voting process begins. All cores on all nodes attach the current *term* to their requests, and the receiving nodes verify whether the received *term* matches their own. This *term* mechanism is a key factor in ensuring the reliability of the algorithm under the Accidental-Behavior Fault Tolerance (ABFT) model: it guarantees that all correct cores operate within the same logical time cycle during leader election and state synchronization, effectively rejecting interference from cores whose states may be inconsistent due to crashes or restarts, thereby preserving the temporal consistency of leader election. The handling of *term* inconsistencies is summarized in Table 2.

Table 2. Strategies for addressing term inconsistency.

Situation	Strategy
Received SS_i value	Insert $SS_i = -1$ into max-heap
N_i under fault isolation	Perform fault isolation on the N_i if data is intact, else reject request
Heartbeat of N_v and N_p	See Section 3.6, Item 5

Figures 5 and 6 illustrate the leader-election process. Each node introduces a random delay of 150–300 ms, as in previous work [16], before a core is selected to respond and communicate with other nodes. This core is referred to as the *Contact Core*. Upon computing SS_i, the *Contact Core* broadcasts SS_i along with the dependent parameters (e.g., S_i and D_i) to the *Contact Core* of other nodes. When node N_j receives SS_i and the associated parameters from node N_i, it executes the following logic: if the current *term* matches, and verification confirms that $SS_i \in [0, 1]$ strictly corresponds to the parameters, then the node is inserted into a max-heap. The heap is ordered primarily by SS_i; if two nodes share identical SS_i values, their unique identifier i is used as a tie-breaker. If the *term* is inconsistent, SS_i does not match the computed result, or both conditions occur, then the node is inserted into the max-heap with a value of -1, and subsequently N_j will reject votes from nodes with $SS_i = -1$. All subsequent primary and secondary leader elections rely on the *Contact Core* to perform inter-node communication and coordination.

After node N_i in the system receives the SS values from all nodes, it initiates the primary leader election. Node N_i extracts the top element from the max-heap as the intended candidate (Primary OP) and votes for the node N_p with the highest SS value to be elected as the primary leader. Since the system operates correctly only when more than half of the nodes function properly, the majority of nodes will vote for N_p, electing it as the primary leader, as shown in Figure 5.

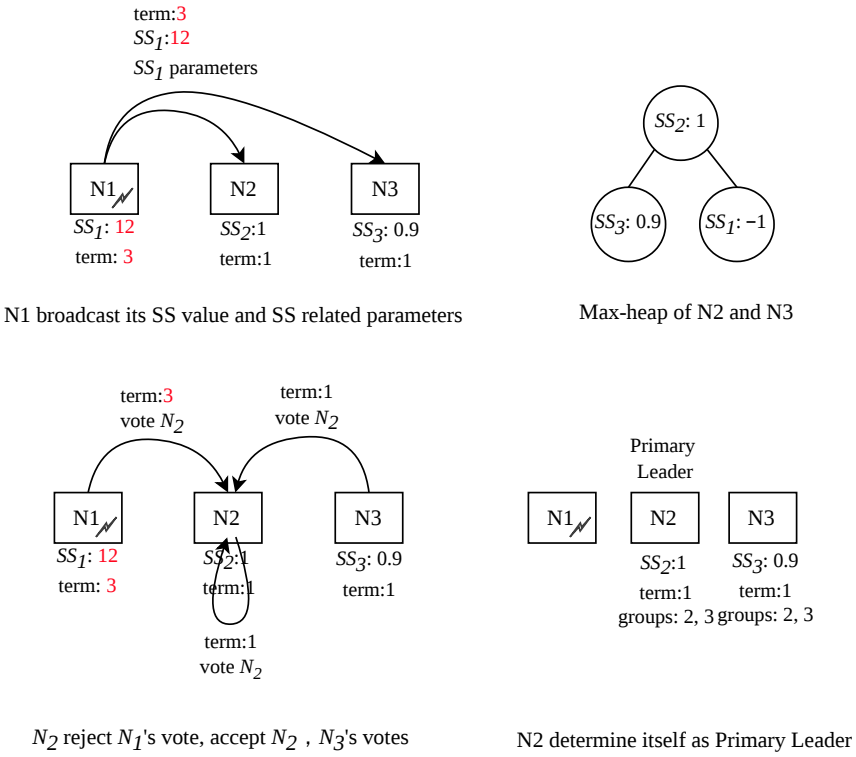

Figure 5. Primary leader election.

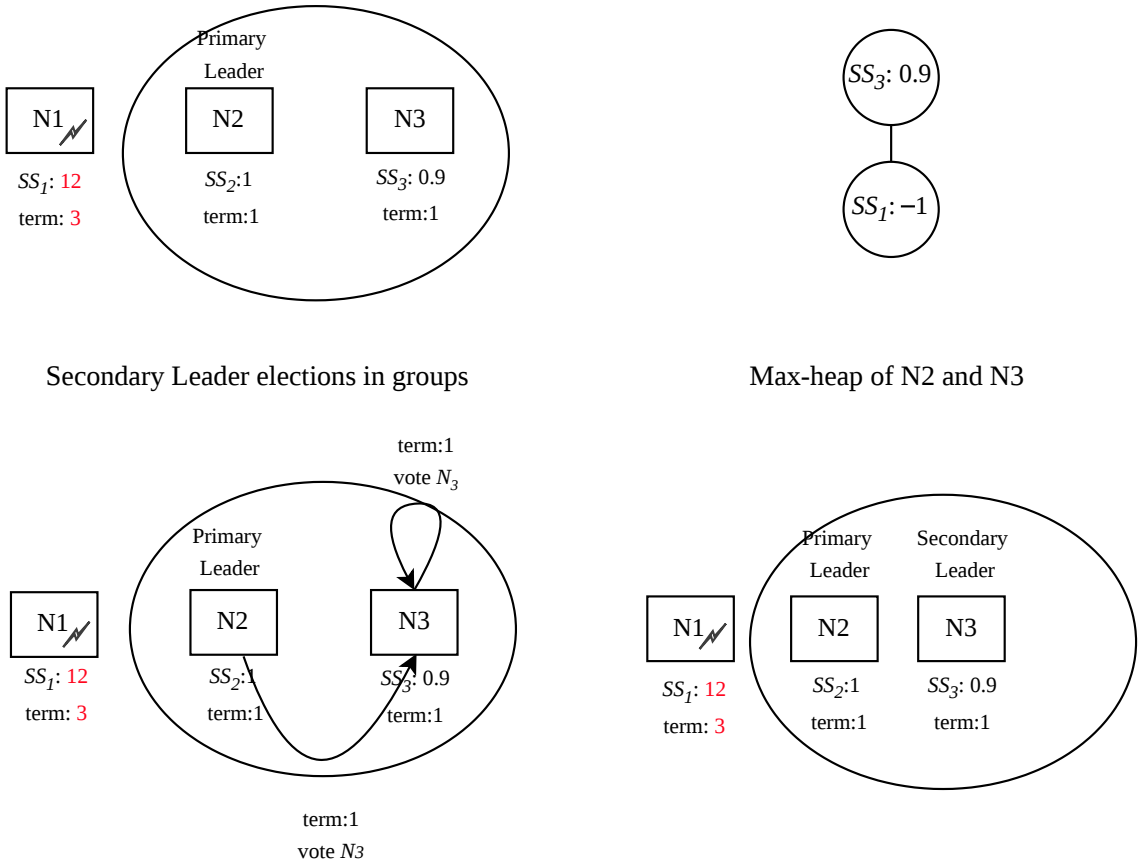

Figure 6. Secondary leader election.

Once N_p receives votes from more than half of the nodes, it broadcasts a notification to all nodes in the system announcing itself as the primary leader. Upon receiving acknowledgment (Ack) messages from other nodes, N_p adds them to its group list. After receiving Acks from the majority of nodes, N_p confirms itself as the primary leader. When the primary leader election phase ends, the primary leader instructs all nodes in its group to perform a secondary leader election.

The primary leader and the nodes in the group vote for a secondary leader, N_v, in the same manner as in the primary leader election. N_p is responsible for making judgments and submitting orders, while N_v supervises and assists N_p's actions, Nodes that fail to establish a primary leader before the timeout will reinitiate the primary leader election, while nodes that have already confirmed a primary leader will reject such election requests. The process of secondary leader election is illustrated in Figure 6.

The random selection of a *Contact Core* is designed to prevent a node from inadvertently choosing a faulty core to broadcast the *SS* value. If a node selects a faulty core as its *Contact Core*, the system must handle the following three cases:

1. The faulty *Contact Core* broadcasts an incorrect *SS* value. According to Equation (3), the *SS* value should fall within the range $[0, 1]$. If the received *SS* value is outside this range, or verification based on Equation (3) fails, the receiver will assign the *SS* value of that node to -1 and insert it into the max-heap. The same handling applies when the term values are inconsistent.

2. The faulty *Contact Core* fails to broadcast the *SS* value. If the *SS* value is never broadcast, the system will immediately initiate the primary leader election once the *SS* broadcasting phase times out. Since more than half of the nodes are functioning correctly, the election can still proceed normally.
3. The faulty *Contact Core* falsely claims leadership. Each node maintains its intended candidate; if a node receives a leadership claim from a node different from its own intended candidate, it will reject the claim.

As shown in Figure 6, the two most reliable nodes, N_2 and N_3, are designated as the primary and secondary nodes to perform fault detection for other nodes within the system: N_2 is making judgments and submitting orders, while N_3 supervises and assists N_2's actions. After the timeout of the primary–secondary leader election phase, nodes that have successfully established the primary and secondary leaders subsequently enter the fault detection and isolation phase. The fault tolerance guarantees of the FTI-TMR algorithm will be discussed in detail in Section 3.6.

3.5.2. Permanent-Fault Detection and Isolation

The primary and secondary leaders perform fault isolation sequentially on each core of all nodes in the system. We begin with a brief description based on the finite state machine of the fault-isolation process shown in Figure 7. Firstly, fault isolation for node N_i is initialized. The primary and secondary leaders, N_p and N_v, request the disputed task list, $DTList_i$, from core $Core_{ik}$ (where i denotes node i and k denotes its k-th core). After receiving $DTList_i$ and verifying the data integrity, N_p and N_v store a copy of $DTList_i$ and execute the tasks in it, comparing the results with the execution of N_i.

At this point, the system enters the fault isolation phase. Once fault isolation is completed, N_p and N_v determine the status of each core of N_i. The primary leader, N_p, then instructs the fault-free cores to disable the faulty ones (Logical Isolation is defined in Section 3.6, Item 5), re-enable any mistakenly disabled cores, and replace $DTList_i$ with the updated version processed by N_p. If all cores of N_i are faulty, an alert is raised indicating that N_i is completely abnormal. Once the fault isolation for N_i is completed, the process proceeds to node N_{i+1}, and this continues until all nodes in the system have been examined. Subsequently, we describe in detail the operations performed at each stage in Figure 7.

The initialization of fault isolation for node N_i is illustrated in top left area of Figure 7. If N_p and N_v request $DTList_i$ from $Core_{ik}$ but neither receive a response or the returned data is corrupted, N_p records the core as faulty and requests $DTList_i$ from $Core_{i,k+1}$. If all cores are queried sequentially and either no response is received or the data is corrupted, N_p raises an alert indicating that the entire node has failed. If $DTList_i$ is received and the data is intact, N_p and N_v store a copy of $DTList_i$ and execute $DTList_i[j][m].task$, which represents the m-th disputed task generated on the j-th core. The system then starts the fault-isolation process for N_i.

The fault isolation phase is illustrated in top right area of Figure 5. If the results of three consecutive executions on $Core_{ij}$ are consistent with N_p, N_p and N_v execute $DTList_i[m+1]$ and remove the entry, $DTList_i[m]$. If all tasks in $DTList_i$ passed the test, Core_{ij} is recorded as fault-free.

If the results are inconsistent, N_p and N_v mark $Core_{ij}$ as faulty and increment $DTList_i[m].failedCount$, which is used for exponential backoff to prevent frequent rechecking of the same core. N_p and N_v then execute $DTList_i[j+1][m]$.task, i.e., they check $\text{Core}_{i,j+1}$, and repeat this process until all the cores of N_i have been examined.

If there are available cores in N_i, let E_i denote the set of cores with permanent faults and A_i denote the set of fault-free cores. N_p replaces $DTList_i$ on the cores in A_i and instructs them to disable the cores in E_i.

If all cores of N_i are faulty, an alert is raised indicating that N_i is completely abnormal. After the fault isolation for N_i is completed, the system proceeds to node N_{i+1}. This process continues until all nodes in the system are inspected.

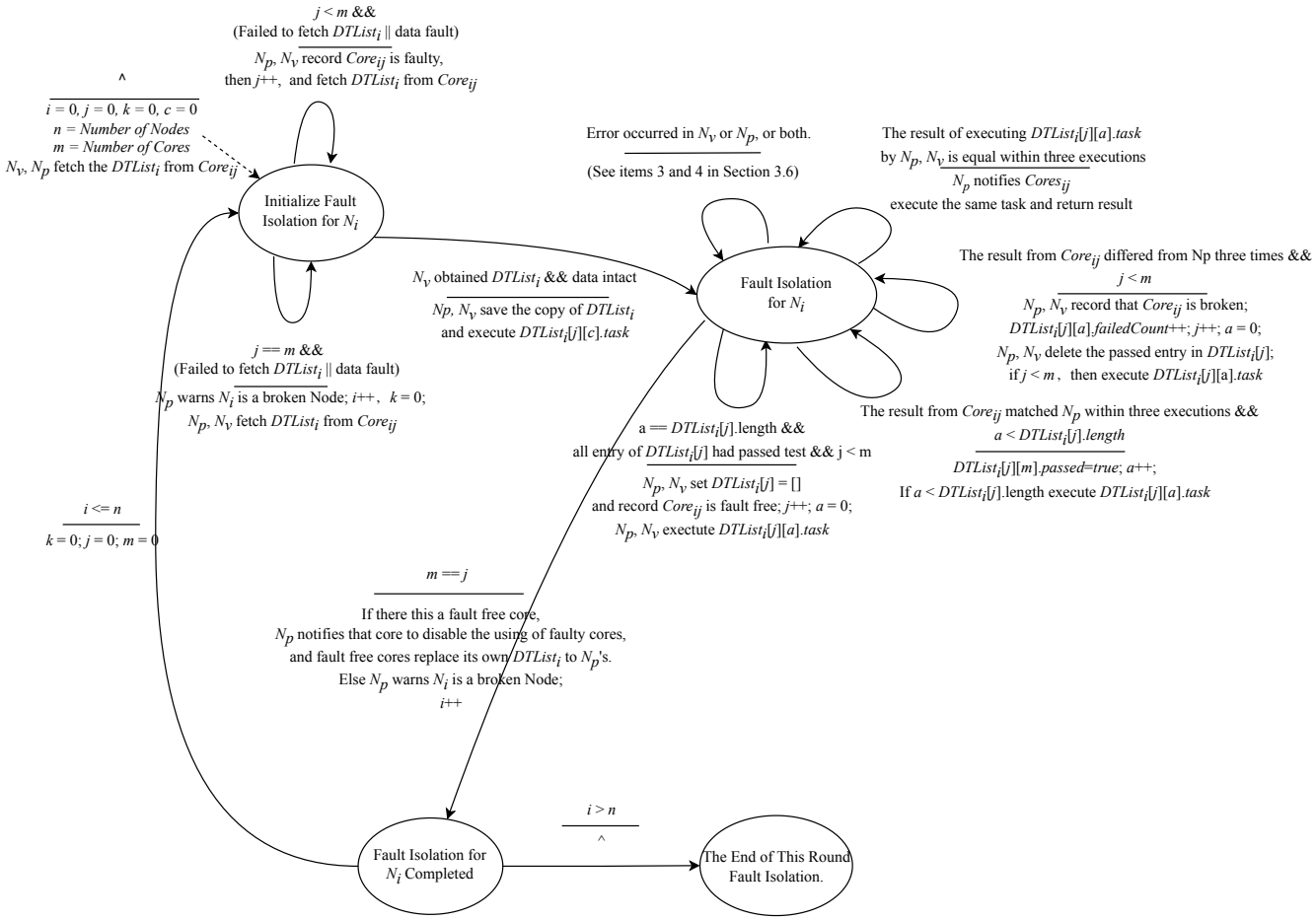

Figure 7. Finite state machine description of the fault-isolation process.

3.6. Formal Correctness Proof of FTI-TMR

First, we articulate the core mechanisms that guarantee the correctness of FTI-TMR:

1. Term Mechanism: The term mechanism synchronizes the state of all cores along the logical time dimension. It effectively isolates cores whose states are corrupted due to crashes or restarts, ensuring that leader election and state synchronization occur only among honest and temporally synchronized cores, thereby enhancing the system's crash resilience.
2. Majority Consensus: Since faulty cores are limited to accidental behavior rather than malicious behavior, all healthy cores honestly compute and broadcast their real SS values. Assuming that more than half of the cores are operational, under the constraints of the term mechanism and SS recomputation verification, the two most stable nodes, N_p and N_v, tend to obtain the majority of votes. The elected primary leader is recognized by all healthy cores and is physically among the most stable nodes, ensuring the consistency and validity of the election.
3. Dynamic Self-Healing of Leaders: Even if N_p or N_v are initially misselected, or if one of them fails during the fault detection phase (i.e., executing a contention task inconsistently three times), N_p and N_v immediately trigger re-election and introduce a third, highly stable node for arbitration and replacement. If no replacement is possible,

the original N_p and N_v are removed and a new leader is re-elected (see the heartbeat mechanism).

4. Heartbeat Mechanism: During permanent-fault isolation, N_p and N_v broadcast heartbeats to the other nodes to indicate the node currently under inspection. The nodes in the system initiate a new election after a random delay of 150–300 ms (ensuring only one node initiates the election) if any of the following conditions is met:
   - A heartbeat from N_p or N_v is not received within the timeout period.
   - N_p and N_v inspect different nodes at the same time.
   - The term is inconsistent.

   Once a majority of the nodes agrees, the original N_p and N_v are removed in the next election, and the system proceeds to elect new leaders.
5. Logical Isolation: For a node, N_i, given the set of permanent faulty cores, $\mathcal{F}_i$, and fault-free cores, $\mathcal{A}_i$ (including a $Core_{ia}$), Logical Isolation mitigates fault propagation. N_p directs all cores in $\mathcal{A}_i$ to adhere to schedules from $Core_{ia}$, assigning tasks only within $\mathcal{A}_i$ and never to $\mathcal{F}_i$. Concurrently, $Core_{ia}$ tries to instruct $\mathcal{F}_i$ not to execute tasks. Consequently, any attempt by $\mathcal{F}_i$ to assign tasks to $\mathcal{A}_i$ is rejected, effectively isolating the accidental behavior of the faulty cores.

3.6.1. Assumptions and Definitions

As Section 3.2 suggests, $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$ denotes the set of all nodes in the system. $\mathcal{A} \subseteq \mathcal{N}$ denotes the set of fault-free nodes. $\mathcal{F} = \mathcal{N} \setminus \mathcal{A}$ denotes the set of faulty nodes, and the set of faulty cores of a node is $\mathcal{F}_i$. Its set of fault-free cores is $\mathcal{A}_i$. $Out(Node_i)$ is a task result of $Node_i$.

Definition (Healthy Behavior)**.** *We define five events first:*

- E_{1x}*: the term of* $Core_x$ *is consistent with the majority of the system.*
- E_{2x}*: if* $Core_x$ *is a leader, its heartbeats are sent on time. And its actions are consistent the other leader.*
- E_{3x}*: the* SS_i *of* $Core_x$ *is in* $[0, 1]$*.*
- E_{4x}*: the* SS_i *of* $Core_x$ *equal the computed from its parameters.*
- E_{5x}*:* $Core_x$ *computes the right result of a task, which is assumed as* $Out(Core_x) = 3$*.*

Then, let $NormalLike_x = E_{1x} \cap E_{2x} \cap E_{3x} \cap E_{4x}$*.* $Normal_x = E_{1x} \cap E_{2x} \cap E_{3x} \cap E_{4x} \cap E_{5x}$*.*

Assumption (Faulty Core, Bad Behavior)**.** *For simplicity, let* $Out(Core) \in [1, 2, 3, 4, 5]$ *and the right result of any task be number* 3*.*

$\forall faultycoreCore_f, and healthycoreCore_h$ *: let* $Core_f$ *obtain each result evenly; then,* $P\{Out(Core_f) = 3\} = 0.2$ *and* $P\{Out(Core_h) = 3\} \approx 1$*.*

$P(NormalLike_f) \ll P(NormalLike_h)$*,* $P(Normal_f) \ll P(Normal_h)$*.*

Assumption (Majority Healthy)**.** *Under the Accidental-Behavior Fault Tolerance (ABFT) model, the system satisfies* $|\mathcal{A}| \geq \lfloor n/2 \rfloor + 1$*; i.e., more than half of the nodes are fault-free.*

Definition (Correct System State)**.** *The system is said to be operating correctly if and only if*

1. *The elected primary leader,* N_p*, and backup leader,* N_v*, are both fault-free, i.e.,* $N_p, N_v \in \mathcal{A}$*;*
2. *All faulty cores can be isolated eventually.*

3.6.2. Main Theorem

Theorem. *Under the ABFT model, if Assumptions Majority Healthy and Faulty Core, Bad Behavior hold, then the FTI-TMR algorithm is more likely to make fault-free cores into the leader.*

Proof. Based on the Faulty Core, Bad Behavior Assumption, we can easily know that $\forall Node_x \in \mathcal{A}, Node_y \in \mathcal{F}, P(SS_x = SS_{max}) > P(SS_y = SS_{max})$. If a faulty core residing on a node wants to keep its leader position in a fault isolation phase, then the probability of this is

$$\begin{aligned} &P(Faultycoresholdleaderposition) = \\ &\quad \prod_{i=1}^{n} P\{NormalLike_p \wedge NormalLike_v \mid (N_p \in \mathcal{F} \vee N_v \in \mathcal{F})\} \times \\ &\quad \prod_{j=1}^{D_i} P\{Out(Core_p) = Out(Core_v) \mid (N_p \in \mathcal{F} \vee N_v \in \mathcal{F})\} \end{aligned} \tag{4}$$

where n is the number of nodes in the system, and D_i is the number of disputed tasks requiring verification. The leaders must remain *NormalLike* and consistent with the other leader during each inspection round. Assuming independent inspection outcomes across rounds, and based on Assumption (Faulty Core, Bad Behavior), we can conclude that Equation (4) is significantly smaller than Equation (5).

$$\begin{aligned} &P(Healthycoresholdleaderposition) = \\ &\quad \prod_{i=1}^{n} P(NormalLike_p \wedge NormalLike_v \mid N_p \in \mathcal{A} \wedge N_v \in \mathcal{A}) \times \\ &\quad \prod_{j=1}^{D_i} P(Out(Core_p) = Out(Core_v) \mid N_p \in \mathcal{A} \wedge N_v \in \mathcal{A}) \end{aligned} \tag{5}$$

Therefore, based on the Raft *MajorityHealthy*, *Term* and *Heartbeat* mechanisms, we know that, in each isolation phase, there will be only two nodes becoming leaders (once the primary leader is elected, the system selects the secondary leader). We prove that these two nodes, which hold these positions, are more likely to be fault-free. □

Theorem. *Under fault-free leadership, the number of isolated faulty cores in the FTI-TMR system is stochastically stable around*

$$\lfloor \frac{\bar{p_{iso}}}{\bar{p_{iso}} + \bar{p_{reg}}} \cdot |\mathcal{F}_{total}| \rfloor$$

within bounded time, where

- $\bar{p}_{iso}$ *is the average probability of isolating a faulty core;*
- $\bar{p}_{reg}$ *is the average probability of reactivating a faulty core;*
- $\bar{p}_{reg} \ll \bar{p}_{iso}$*;*
- $|\mathcal{F}_{total}|$ *is the total number of broken cores in the system.*

Proof. Let $\bar{p_{iso}}$ denote the average probability of isolating a faulty core; here, $\bar{d}$ is the number of the average disputed task:

$$\begin{aligned} \bar{p_{iso}} \approx \binom{\bar{d}}{1} P^3\{ &\\ &\neg NormalLike_i \vee Output(N_i) \neq Output(N_p) \mid \\ &(N_i \in \mathcal{F}) \wedge (N_p \in \mathcal{A}) \\ &\} \approx \min(0.8^3 \bar{d}, 1) \end{aligned} \tag{6}$$

$\bar{p_{reg}}$ is the average probability of reactivating the faulty core, in which

$$
\begin{aligned}
p_{\bar{reg}} \lessapprox \prod_{j=1}^{\bar{d}} P^3\{ & \\
& NormalLike_i \wedge Output(N_i) = Output(N_p) \mid \\
& (N_i \in \mathcal{F}) \wedge (N_p \in \mathcal{A}) \\
& \} \approx 0.2^{3\bar{d}}.
\end{aligned} \tag{7}
$$

Under the exponential backoff strategy, $p_{\bar{reg}}$ is far less than $p_{\bar{iso}}$.

We model this process as a Markov chain. Let X_t denote the number of non-isolated faulty cores in the system at the time t-th system isolation phase, with state space $\{0, 1, \ldots, |\mathcal{F}_{total}|\}$, where $|\mathcal{F}_{total}| = \sum_{i=1}^{n} |\mathcal{F}_i|$ and $X_0 = |\mathcal{F}_{total}|$. We write the expected evolution as

$$
E[X_{t+1} \mid X_t = s] = (1 - p_{\bar{iso}}) \cdot s + (|\mathcal{F}_{total}| - s) \cdot p_{\bar{reg}}. \tag{8}
$$

Then, we have

$$
\begin{aligned}
E[X_t - X_{t+1} \mid X_t = s] &= p_{\bar{iso}} \cdot s - (|\mathcal{F}_{total}| - s) \cdot p_{\bar{reg}} \\
&= (p_{\bar{iso}} + p_{\bar{reg}}) \cdot s - |\mathcal{F}_{total}| \cdot p_{\bar{reg}}.
\end{aligned} \tag{9}
$$

We let

$$
s_{min} = \lceil \frac{p_{\bar{reg}}}{p_{\bar{iso}} + p_{\bar{reg}}} \cdot |\mathcal{F}_{total}| \rceil, \tag{10}
$$

which makes $E[X_t - X_{t+1} \mid X_t = s_{min}] \geq 0$. Let $S = [s_{min} + 1, s_{min} + 2, ..., |\mathcal{F}_{total}|]$, $S \subseteq Z^+$. And let T be the random variable that denotes the first point in time, t, for which $X_t = s_{min}$ over truncated state space, $S \cup \{s_{min}\}$.

There exists a real number, $\delta = \frac{1}{|\mathcal{F}_{total}|} > 0$, such that

$$
E[X_t - X_{t+1} \mid X_t = s] \geq \delta s \tag{11}
$$

holds for all $s \in S$ with $P(X_t = s) > 0$. From the Multiplicative Drift Theorem [32], we have

$$
\begin{aligned}
E[T \mid X_0 = |\mathcal{F}_{total}|] &\leq \frac{1 + ln\{|\mathcal{F}_{total}|/(s_{min} + 1)\}}{\delta} \\
&= |\mathcal{F}_{total}| + |\mathcal{F}_{total}| ln\{|\mathcal{F}_{total}|/(s_{min} + 1)\}.
\end{aligned} \tag{12}
$$

For the states in $S = [0, \ldots, s_{\min} - 1]$, the drift is in the opposite direction, which is also toward $s_{\min}$. Therefore, the FTI-TMR tends to isolate $\lfloor \frac{p_{\bar{iso}}}{p_{\bar{iso}} + p_{\bar{reg}}} \cdot |\mathcal{F}_{total}| \rfloor$ broken cores within the bounded time. □

Corollary. *FTI-TMR tends to maintain the Correct System State.*

Proof. From the above two theorems , we know that healthy cores are more likely to be the leaders. Based on the Raft mechanism, there are typically only two leaders in each isolation phase, and healthy leaders can isolate $\lfloor \frac{p_{\bar{iso}}}{p_{\bar{iso}} + p_{\bar{reg}}} \cdot |\mathcal{F}_{total}| \rfloor$ faulty cores within the bounded time.

Furthermore, by applying exponential backoff, the faulty cores that have been isolated are assigned lower influence and a reduced probability of becoming leaders in subsequent

phases. This mechanism enhances the liveness and stability of the system, preventing recurrent leadership by unstable cores and contributing to the overall correctness of the system state.

We say that FTI-TMR tends to maintain the Correct System State. □

Figure 8 shows the overall finite state machine of the FTI-TMR process, which integrates transient fault handling, stability scoring, leader election, and permanent-fault detection and isolation. The system continuously cycles through these phases to maintain high reliability and fault tolerance.

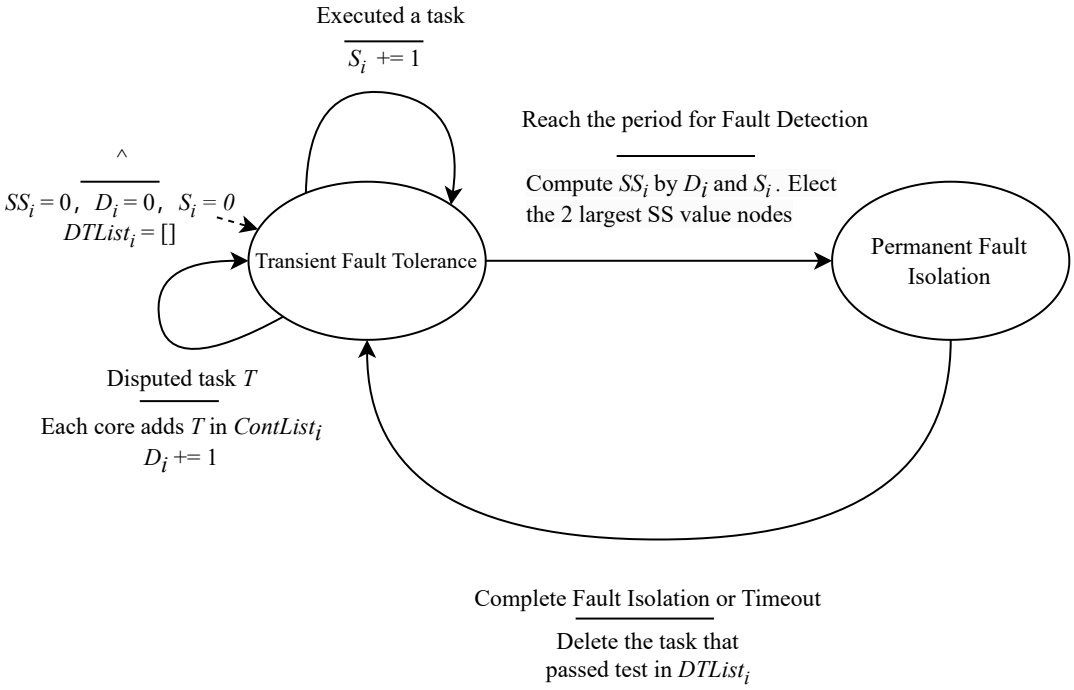

Figure 8. Finite state machine description of the FTI-TMR process.

4. Experiment

To comprehensively evaluate the proposed fault-tolerant scheduling mechanism, we construct a simulation-based experimental environment utilizing representative workloads and rigorous evaluation criteria. Specifically, Section 4.1 (Experimental Setup) outlines our methodology, beginning with Section 4.1.1 (Simulation Framework), which details how we model multicore execution under both transient and permanent fault conditions. Section 4.1.2 (Evaluation Metrics) then introduces the specific metrics adopted to quantify system reliability and execution correctness. Finally, Section 4.2 (Evaluation Methodology) describes the benchmark applications used to test the system.

4.1. Experimental Setup

This section introduces the experimental setup used to evaluate the proposed approach, including the simulation framework, evaluation metrics, and benchmark applications.

4.1.1. Simulation Framework

We simulated a quad-core CPU, running tasks with different methods on our self-developed web-based application. A limitation of this approach is its inability to directly emulate actual hardware-level energy consumption changes and physical fault characteristics. To evaluate the effectiveness of our design in tolerating various faults, we modeled the occurrence of both transient and permanent faults within the simulation framework. For

modeling transient faults, similar to previous work [4,9,33], we used a Poisson distribution with an average occurrence rate of λ.

$$\lambda = \lambda_0 \times 10^{\frac{d \times (1-S)}{1-S_{\min}}} \tag{13}$$

Transient faults are triggered according to Equation (13). In prior work, S represents the voltage/frequency level, selected from a set of eight values $\{\frac{0.85}{1.55}, \frac{0.95}{1.55}, \ldots, \frac{1.55}{1.55}\}$. Due to limitations in our experimental setup, we set $S = \frac{1.45}{1.55}$ based on a comparative analysis with existing studies. Here, $\lambda_0 = 10^{-6}$ faults/second denotes the transient fault rate at the maximum voltage, and $d = 3$ is a technology-dependent constant, both adopted from established research [4,5,9,33]. Accordingly, based on the properties of the Poisson distribution, the probability of a task encountering a transient fault is given by

$$F = 1 - e^{-\lambda \times T} \tag{14}$$

The probability that a task executes without encountering any transient faults is $1 - F = e^{-\lambda \times T}$.

For permanent faults, a faulty core has a near-zero probability of producing a correct task result. Additionally, during critical processes such as elections and fault isolation, it has a 50% probability of performing accidental actions. During the experiments, we configured nine virtual 4-core machines on the simulation platform. Four of these machines contained permanently damaged cores, with the number of faulty cores being 1–4, respectively. The other five machines had no permanent core faults.

4.1.2. Evaluation Metrics

To evaluate the reliability of the entire application system, we adopted the *Probability of Failure (PoF)* [4,5,9,33] as the metric, which reflects the risk level of the system producing erroneous results during execution. To ensure an application is processed correctly, all tasks from that application must ultimately output correct results. According to the mechanism described in Section 2, a task is considered successfully executed if at least two of its three copies encounter no failures. Conversely, if two or more copies of a task fail, then the application execution is considered to have failed. Therefore, *PoF* is calculated as the ratio of failed applications to the total number of executed applications. Since our simulation platform cannot effectively simulate energy consumption, we used the number of executed tasks as a substitute measure for energy consumption.

4.1.3. Applications

We simulated multiple real-world embedded applications using the Standard Task Graph (STG) [34]. The STG application suite covers typical scenarios such as robotic control, sparse matrix solvers, and SPEC fpppp programs. For comparison, we also included a highly parallel application consisting of 200 randomly generated discrete tasks. In each experiment, each node repeatedly executed the same application on a four-core CPU. For each application, we executed it 100 times on every node. Although a limited number of runs is unlikely to trigger transient faults, this setup is acceptable as our study focuses primarily on the detection and isolation of permanent faults.

4.2. Evaluated Methods

We evaluated the following methods:

- Conventional TMR (C-TMR): In the C-TMR scheme, the system consistently executes three copies of each task, as described in Section 2.1.

- Enhanced Two-Phase TMR (TP-TMR+): This method incorporates a straightforward enhancement to tolerate permanent faults at the cost of increased energy consumption. By employing an improved offline scheduling strategy, it avoids assigning multiple copies of the same task to the same core, thereby achieving fault tolerance against permanent failures.
- Reactive TMR (R-TMR): Based on execution histories across different cores, this approach attempts to disable faulty cores and reassign their tasks to functional ones.
- Fault Tolerance and Isolation TMR (FTI-TMR): Our proposed fault-tolerant algorithm leverages execution history and other information to vote for the most stable devices within the interconnected system. More reliable nodes are then tasked with error detection and isolation in others, leading to improved fault tolerance and isolation capabilities. In our experiments, the initial fault isolation cycle was set to two rounds of application execution. After each isolation phase, the cycle length grows exponentially until it reaches a maximum of 100 rounds, at which point the growth stops. Overhead operations generated by the algorithm, such as voting and fault isolation, are also recorded as executed tasks. When the FTI-TMR mechanism reached its fault detection cycle, the values of f_i, β and F_0 were set to 0, and SS_i was calculated according to Equation (3).

4.3. Results and Analysis

Figure 9 shows the number of executed tasks for each method across nine nodes in the absence of permanent faults, normalized against the execution count of C-TMR. As observed, C-TMR yields the highest number of executed tasks. Under low transient fault rates, TP-TMR+, R-TMR, and FTI-TMR seldom execute the third task copy, thereby reducing the computational load by approximately one-third compared to C-TMR, with an average task execution count around 66.84% of that of C-TMR.

FTI-TMR executes slightly more tasks than TP-TMR+ and R-TMR, as it performs additional operations for voting and fault detection at each error-checking cycle. Specifically, FTI-TMR's task count is about 0.76% higher than that of the state-of-the-art R-TMR.

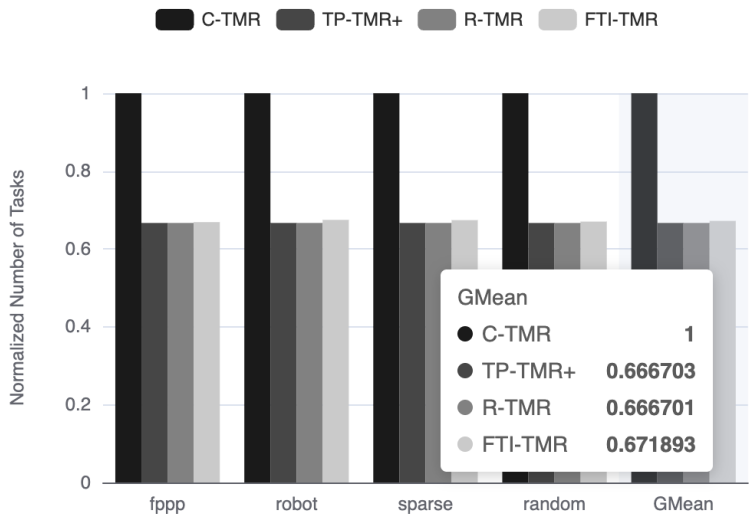

Figure 9. Number of executed tasks per method under no permanent faults on 9 nodes (normalized to C-TMR).

In scenarios with permanent faults, we configured nine four-core nodes, among which four nodes had one–four permanently faulty cores, respectively, and five nodes remained fault-free. Figure 10 presents the normalized task counts relative to C-TMR under such conditions. All three methods—TP-TMR+, R-TMR, and FTI-TMR—show increased task

counts compared to the fault-free case. FTI-TMR exhibits the smallest increase, followed by R-TMR and then TP-TMR+. As summarized in Table 3, R-TMR can detect and isolate any number of faulty cores within the system as long as at least two fault-free cores are available. This is because, in the presence of at least two fault-free cores, R-TMR's scheduling strategy can always allocate a task's two copies to these fault-free cores. The third task copy might be scheduled to a faulty core. Obviously, when there are fewer than two fault-free cores, R-TMR failes rapidly. On contrast, FTI-TMR can handle up to m faulty cores, as also demonstrated in our online visual experiments. When all cores in a node become faulty, FTI-TMR can report the node as entirely faulty. Nodes in which all cores are faulty require frequent execution of the third task copy; therefore, FTI-TMR's task count increases under permanent faults, yet it still reduces the number of executed tasks by about 4.92% compared to the R-TMR.

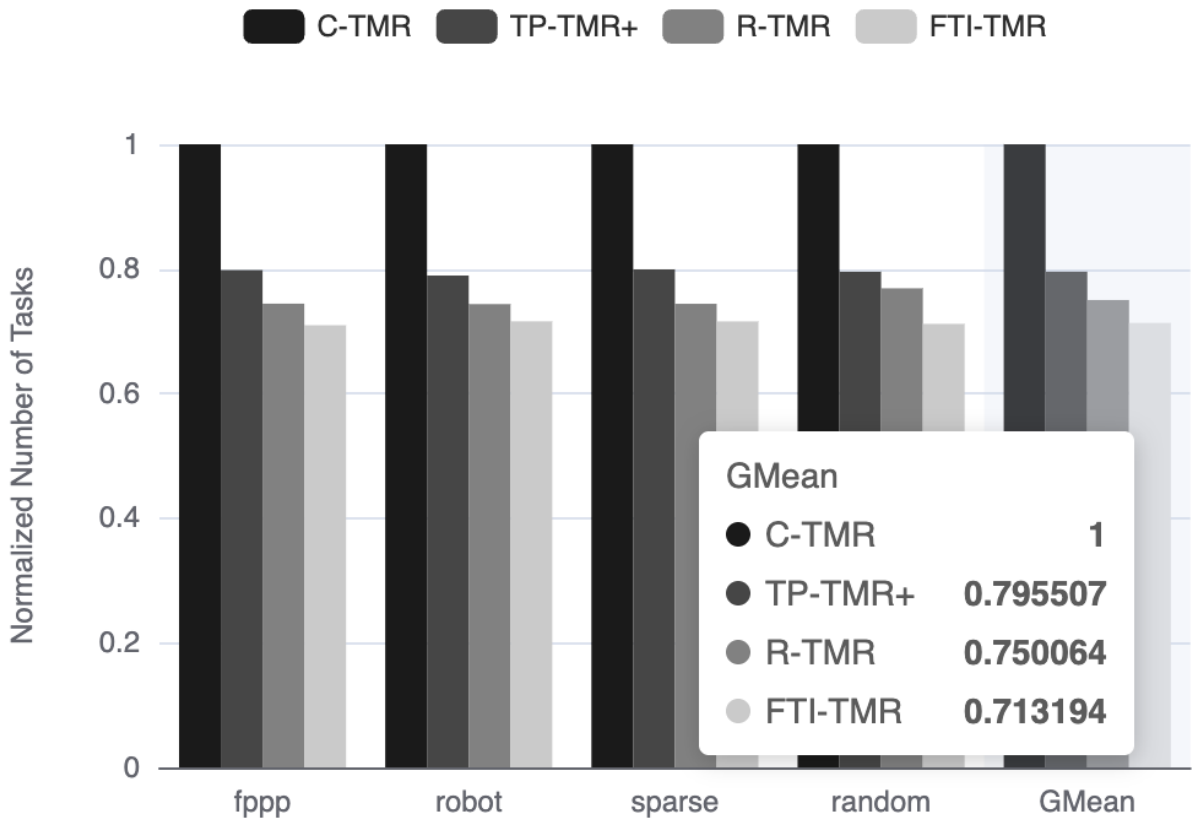

Figure 10. Number of executed tasks per method on 9 nodes with permanent faults (5 fault-free, 4 with 1–4 cores broken, normalized to C-TMR).

Table 3. Ability of each method to handle permanent core failures in m-core nodes.

Method	Isolating < m−1 Permanent Faulty Cores	Isolating m−1 Permanent Faulty Cores	Detecting m Permanent Faulty Cores
C-TMR	×	×	×
TP-TMR+	×	×	×
R-TMR	✓	×	×
FTI-TMR	✓	✓	✓

We further compare the *Probability of Failure* (*PoF*) under this configuration, as listed in Table 4. The R-TMR scheme slightly increases the *POF* under random application

workloads. This comes at the cost of delaying the allocation of two task copies to fault-free cores, thereby postponing the detection and isolation of the faulty core.

Table 4. Probability of Failure (PoF) comparison on 9 nodes with permanent faults (5 fault-free, 4 with 1–4 cores broken).

Application	Probability of Failure			
	C-TMR	**TP-TMR+**	**R-TMR**	**FTI-TMR**
fppp	0.3378	0.3333	0.2244	0.1211
robot	0.3344	0.3356	0.2244	0.1156
sparse	0.3344	0.3367	0.2244	0.1167
random	0.3344	0.3333	0.2988	0.1156
Average	0.3353	0.3347	0.2431	0.1172

FTI-TMR achieves the lowest *PoF*, benefiting from its superior fault detection and isolation capability. However, when a node is fully faulty, FTI-TMR cannot shut it down and only notifies about its failure, resulting in a *PoF* of 11.72% in such cases. On average, FTI-TMR achieves a reduction of approximately 51.79% in the *PoF* compared to R-TMR.

5. Conclusions

As a widely adopted fault tolerance mechanism, Triple Modular Redundancy (TMR) ensures high reliability through triple-redundant computation and majority voting. However, traditional TMR incurs significant task overheads, making it challenging to deploy directly in resource-constrained systems such as embedded or avionics applications. To mitigate this, various improved schemes have been proposed—including task scheduling and power management optimizations—among which Two-Phase TMR (TP-TMR) and Reactive TMR (R-TMR) are notable examples.

TP-TMR introduces a task-optimized Triple Modular Redundancy method that executes the third task copy on demand. Building upon this, R-TMR allocates tasks to different cores (the Enhanced Two-Phase TMR) and incorporates additional lightweight hardware to identify and isolate permanent faults, causing additional hardware overheads. R-TMR has difficulties in detecting and isolating multiple simultaneous permanent faults. They both remain confined to the isolated node paradigm, relying on local hardware.

To overcome these limitations, this paper proposes Fault Tolerance and Isolation TMR (FTI-TMR), a method designed for interconnected systems. During normal operation, FTI-TMR leverages the Enhanced Two-Phase TMR (TP-TMR+) mechanism to both tolerate faults and minimize task execution. To detect and isolate permanent faults, the method introduces a stability score that assesses the stability of each node, allowing the two most reliable nodes to periodically detect and isolate permanent faults in the other nodes. This approach achieves superior permanent-fault coverage, significantly improving reliability and reducing the number of tasks executed.

A key limitation of this paper is that our work was done on a web simulation platform. It cannot capture the physics of hardware-level energy drain or reflect how permanent faults really behave. The results presented in this work are also limited to the assumed system model, where interconnected nodes communicate through reliable and fault-free TCP channels. Faulty components are considered honest but faulty and non-Byzantine. Moreover, the reported fault-coverage results assume that a majority of nodes remain fault-free. Nevertheless, we think the core idea behind FTI-TMR—having reliable nodes actively perform diagnosis on others—is not limited to CPU fault tolerance based on TMR. With rapid advancements being made in artificial intelligence and autonomous systems, it is feasible to apply this same “reliable node” concept to multi-robot teams. The idea would

be to have the more dependable robots monitor, diagnose, and maybe even repair other units in the field. This could be a step toward building truly autonomous systems.

Author Contributions: Conceptualization, Y.H. and C.W.; methodology, Y.H.; software, Y.H.; validation, Y.H. and C.W.; formal analysis, Y.H.; investigation, Y.H. and C.W.; resources, Y.H. and C.W.; data curation, Y.H.; writing—original draft preparation, Y.H.; writing—review and editing, Y.H. and C.W.; visualization, Y.H.; supervision, C.W.; project administration, C.W.; funding acquisition, C.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data can be found in https://github.com/Jamin2025/FTI-TMR-paper-code/tree/master/results (accessed on 30 July 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Buttazzo, G.C. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*; Springer: Berlin/Heidelberg, Germany, 2011.
2. Solouki, M.A.; Angizi, S.; Violante, M. Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques. *IEEE Access* **2024**, *12*, 180939–180967. [CrossRef]
3. Safari, S.; Ansari, M.; Khdr, H.; Gohari-Nazari, P.; Yari-Karin, S.; Yeganeh-Khaksar, A.; Hessabi, S.; Ejlali, A.; Henkel, J. A Survey of Fault-Tolerance Techniques for Embedded Systems from the Perspective of Power, Energy, and Thermal Issues. *IEEE Access* **2022**, *10*, 12229–12251. [CrossRef]
4. Salehi, M.; Ejlali, A.; Al-Hashimi, B.M. Two-Phase Low-Energy N-Modular Redundancy for Hard Real-Time Multi-Core Systems. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 1497–1510. [CrossRef]
5. Mireshghallah, F.; Bakhshalipour, M.; Sadrosadati, M.; Sarbazi-Azad, H. Energy-Efficient Permanent Fault Tolerance in Hard Real-Time Systems. *IEEE Trans. Comput.* **2019**, *68*, 1539–1545. [CrossRef]
6. Analysis, O. M2M Connections Market Outlook 2026–2034: Market Share and Growth Analysis by Connection Type (Wired, Wireless), Technology, Deployment Model, End-User Industry, 2026. Available online: https://www.researchandmarkets.com/reports/6243165/m2m-connections-market-outlook-market-share (accessed on 3 March 2026).
7. Balaguera, J.D.G.; Condia, J.E.R.; Santos, F.F.D.; Reorda, M.S.; Rech, P. *Understanding the Effects of Permanent Faults in GPU's Parallelism Management and Control Units*; Association for Computing Machinery: New York, NY, USA, 2023. [CrossRef]
8. Lanzieri, L.; Martino, G.; Fey, G.; Schlarb, H.; Schmidt, T.C.; Wählisch, M. A Review of Techniques for Ageing Detection and Monitoring on Embedded Systems. *ACM Comput. Surv.* **2024**, *57*, 24. [CrossRef]
9. Ejlali, A.; Al-Hashimi, B.M.; Eles, P. Low-Energy Standby-Sparing for Hard Real-Time Systems. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2012**, *31*, 329–342. [CrossRef]
10. Preparata, F.P.; Metze, G.; Chien, R.T. On the Connection Assignment Problem of Diagnosable Systems. *IEEE Trans. Electron. Comput.* **1967**, *EC-16*, 848–854. [CrossRef]
11. Malek, M. *A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems*; Association for Computing Machinery: New York, NY, USA, 1980. [CrossRef]
12. Sengupta, A.; Dahbura, A.T. On Self-Diagnosable Multiprocessor Systems: Diagnosis by the Comparison Approach. *IEEE Trans. Comput.* **1992**, *41*, 1386–1396. [CrossRef]
13. Yuan, C.; Zou, C.; Wu, J.; Feng, H.; Li, J. IFDA: Intermittent Fault Diagnosis Algorithm for Augmented Cubes Under the PMC Model. *Appl. Sci.* **2025**, *15*, 8197. [CrossRef]
14. Zhu, D.; Melhem, R.; Mosse, D.; Elnozahy, E. Analysis of an energy efficient optimistic TMR scheme. In Proceedings of the Tenth International Conference on Parallel and Distributed Systems, 2004—ICPADS 2004, Newport Beach, CA, USA, 9 July 2024; pp. 559–568. [CrossRef]
15. Melhem, R.; Mosse, D.; Elnozahy, E. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. Comput.* **2004**, *53*, 217–231. [CrossRef]
16. Ongaro, D.; Ousterhout, J. *In Search of an Understandable Consensus Algorithm*; USENIX Association: Berkeley, CA, USA, 2014.
17. Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst.* **1998**, *16*, 133–169. [CrossRef]
18. Cloud Native Computing Foundation. Etcd: A Distributed Reliable Key-Value Store for the Most Critical Data of a Distributed System. Available online: https://etcd.io/ (accessed on 18 October 2025).
19. HashiCorp. Consul: Service Mesh and Service Discovery. Available online: https://www.consul.io/ (accessed on 18 October 2025).

20. Ferrari, R.M.G.; Parisini, T.; Polycarpou, M.M. Distributed Fault Detection and Isolation of Large-Scale Discrete-Time Nonlinear Systems: An Adaptive Approximation Approach. *IEEE Trans. Autom. Control* **2012**, *57*, 275–290. [CrossRef]
21. Kwao, V.; Raptis, I. Particle Filter-Based Fault Diagnosis with Hidden Markov Models for Nonlinear and Uncertain Dynamical Systems. *IEEE Trans. Control Syst. Technol.* **2026**, *34*, 1964–1978. [CrossRef]
22. Xu, B.; Peng, C.; Wang, Y.; Chu, F. Dynamic Line Modeling and Cyclic-Small-Gain Synthesis for Plug-and-Play Fault Detection in Islanded Microgrids. *IEEE Trans. Autom. Sci. Eng.* **2026**, *23*, 11997–12009. [CrossRef]
23. Pourreza, M.; Narasimhan, P. When Timeouts Fail: Revisiting Fault Detection under Resource Stress in Edge Computing. In Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing, New York, NY, USA, 1–4 December 2026. [CrossRef]
24. Aydar, M.B.; Baghaee, S.; Firuzi, K.; Uysal, E.; Yıldırım, G. GridAI: Edge AI Smart Grid Fault Detection with Goal-Oriented Communication. In Proceedings of the 24th Annual International Conference on Mobile Systems, Applications and Services Workshops, New York, NY, USA, 21–25 June 2026; pp. 269–274. [CrossRef]
25. Klutke, G.; Kiessler, P.; Wortman, M. A critical look at the bathtub curve. *IEEE Trans. Reliab.* **2003**, *52*, 125–129. [CrossRef]
26. Dechgummarn, Y.; Fuangfoo, P.; Kampeerawat, W. Predictive Reliability Analysis of Power Distribution Systems Considering the Effects of Seasonal Factors on Outage Data Using Weibull Analysis Combined with Polynomial Regression. *IEEE Access* **2023**, *11*, 138261–138278. [CrossRef]
27. Al-Essa, L.A.; Muhammad, M.; Tahir, M.H.; Abba, B.; Xiao, J.; Jamal, F. A New Flexible Four Parameter Bathtub Curve Failure Rate Model, and Its Application to Right-Censored Data. *IEEE Access* **2023**, *11*, 50130–50144. [CrossRef]
28. Jones, H.W. Monitoring and Modeling the Failure Rate Using the Bathtub Curve. In Proceedings of the 2026 Annual Reliability and Maintainability Symposium (RAMS), Miramar Beach, FL, USA, 26–29 January 2026. [CrossRef]
29. Lamport, L.; Shostak, R.; Pease, M. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* **1982**, *4*, 382–401. [CrossRef]
30. Yang, T.; Gerasoulis, A. List scheduling with and without communication delays. *Parallel Comput.* **1993**, *19*, 1321–1344. [CrossRef]
31. Ventroux, N.; Blanc, F.; Lavenier, D. A Low Complex Scheduling Algorithm for Multi-processor System-on-Chip. In Proceedings of the Parallel and Distributed Computing and Networks, Innsbruck, Austria, 15–17 February 2005.
32. Doerr, B.; Johannsen, D.; Winzen, C. Multiplicative drift analysis. In Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, New York, NY, USA, 7–11 July 2010. [CrossRef]
33. Guo, Y.; Zhu, D.; Aydin, H. Reliability-aware power management for parallel real-time applications with precedence constraints. In Proceedings of the 2011 International Green Computing Conference and Workshops, Orlando, FL, USA, 25–28 July 2011; pp. 1–8. [CrossRef]
34. Tobita, T.; Kasahara, H. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *J. Sched.* **2002**, *5*, 379–394. [CrossRef]