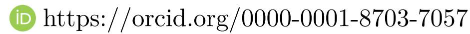
Recovery Time Considerations in Real-Time Systems Employing Software Fault Tolerance

Anand Bhat
Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA
anandbha@andrew.cmu.edu

https://orcid.org/0000-0001-8703-7057

Soheil Samii
General Motors R&D, Warren, MI, USA and Linköping University, Sweden
soheil.samii@gm.com, soheil.samii@liu.se

Ragunathan (Raj) Rajkumar
Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA
rajkumar@andrew.cmu.edu

Abstract

Safety-critical real-time systems like modern automobiles with advanced driving-assist features must employ redundancy for crucial software tasks to tolerate permanent crash faults. This redundancy can be achieved by using techniques like active replication or the primary-backup approach. In such systems, the recovery time which is the amount of time it takes for a redundant task to take over execution on the failure of a primary task becomes a very important design parameter. The recovery time for a given task depends on various factors like task allocation, primary and redundant task priorities, system load and the scheduling policy. Each task can also have a different recovery time requirement (RTR). For example, in automobiles with automated driving features, safety-critical tasks like perception and steering control have strict RTRs, whereas such requirements are more relaxed in the case of tasks like heating control and mission planning. In this paper, we analyze the recovery time for software tasks in a real-time system employing Rate-Monotonic Scheduling (RMS). We derive bounds on the recovery times for different redundant task options and propose techniques to determine the redundant-task type for a task to satisfy its RTR. We also address the fault-tolerant task allocation problem, with the additional constraint of satisfying the RTR of each task in the system. Given that the problem of assigning tasks to processors is a well-known NP-hard bin-packing problem we propose computationally-efficient heuristics to find a feasible allocation of tasks and their redundant copies. We also apply the simulated annealing method to the fault-tolerant task allocation problem with RTR constraints and compare against our heuristics.

2012 ACM Subject Classification Software and its engineering → Software fault tolerance, Software and its engineering → Real-time systems software, Computer systems organization → Real-time systems

Keywords and phrases fault tolerance, real-time embedded systems, recovery time, real-time schedulability

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.23

1 Introduction

Advances in sensing, machine learning and semiconductor technology have resulted in a dramatic increase in the amount and complexity of computational resources used in real-time systems. Many of these systems, such as industrial control, aviation and automobiles

© Anand Bhat, Soheil Samii, and Ragunathan (Raj) Rajkumar;
licensed under Creative Commons License CC-BY

[35], are also safety-critical. The increasing complexity of these computational aspects has compounded the need for these systems to remain dependable [9]. For such systems, the ability to tolerate permanent crash faults is integral to their dependable operation.

Conventionally, fault tolerance is achieved by replicating hardware and often using a voting mechanism to determine the output [32]. Unfortunately, this approach is extremely inefficient in terms of cost, weight, space and energy needs for many applications. This is especially true for the automotive industry, where the system reliability requirements can be diverse and the cost constraints are stringent. For example, consider self-driving cars. Several levels of automation (1 to 5) have been defined in [31] to characterize the spectrum of self-driving features. To put such systems in context with redundancy requirements, consider a Level 2 system active on highways only. In such a system, although the driver is not in direct control of the vehicle motion, the driver still plays a supervisory role: the driver will be requested to take over control in case of any subsystem or component failure. In such a system, perhaps only a small subset of all software tasks need redundancy (e.g., braking and steering control, but not propulsion). Now, consider a Level-4 system active in the same operational domain (highways), where the vehicle itself is responsible to bring the system to a safe state in case of failures. Such high levels of automation impose more stringent fault-tolerance requirements in terms of the number of task replicas (or backups).

Similarly, other diverse needs are also evident from the fact that different tasks running in an automobile have different levels of safety criticality. For example, the braking control task is far more safety-critical than (say) a music playback task. This motivates the need for adaptive cost-optimized fault-tolerance solutions to reduce overall resource utilization. Hence, software fault-tolerance techniques like *active replication* [37, 16] or the *primary-backup approach* [6, 2] using hot and cold standbys are more applicable to systems like automobiles [19, 4].

The diversity in the reliability requirements for tasks in a system using software fault tolerance is captured by the *recovery time requirement* (RTR) of each task. The RTR specifies the number of consecutive deadlines of the primary task that a redundant task can afford to miss without the system being considered to have failed. The recovery time requirement for a task varies depending on its safety criticality. Tasks that are safety-critical have a strict (and very low) upper bound on RTR, while others can afford more relaxed values. The *recovery time* is the time a redundant task takes to successfully take over execution on primary task failure. It is influenced by a number of factors like redundant-task type, redundant task priority and network delays. The goal of this paper is to analyze the recovery times achieved by different types of redundant tasks (active/passive) used in software fault-tolerance techniques for real-time systems. The major contributions of this paper are as follows:

1. We derive the bounds on the recovery time of different types of redundancies, i.e, active or passive, used in software fault-tolerance techniques for real-time systems.
2. We derive conditions to map the recovery time requirements of a task to a redundant-task-type assignment.
3. We propose heuristics to determine redundant-task-type assignments and allocate these tasks to different nodes satisfying the recovery time requirements of all tasks while attempting to optimize resource utilization.
4. We apply the Simulated Annealing method to the fault-tolerant task allocation problem and compare its performance to the heuristics proposed.

The rest of this paper is organized as follows. In Section 2, we describe related work. In Section 3, we define our system model and fault model, and describe different types of redundant tasks we consider for our analysis. In Section 4, we quantify recovery time

and derive bounds on the recovery time for each redundancy type. In Section 5, we derive conditions to assign a redundancy type to a task given its recovery-time requirements. In Section 6, we present heuristics to assign tasks to nodes satisfying the recovery time requirements of each task. We also apply the simulated annealing method to the fault-tolerant task allocation problem and compare its performance against the proposed heuristics. We summarize and conclude our findings in Section 8.

2 Related Work

The problem of supporting fault tolerance at the level of task scheduling has been widely studied in the literature. A number of real-time task allocation algorithms in order to tackle this problem in a distributed real-time system [8, 13, 27] have been described in the literature. In [26], Oh et al. present an online allocation heuristic to assign replicas to a minimum number of processors such that all replicas guarantee that task deadlines are met. They also derive the bound on the number of processors required to feasibly schedule a task set using their heuristic. These approaches focus only on active replication, where the redundant software executes regardless of failure modes. The resource consumption of such approaches is impractical for resource-constrained systems like cars, especially as the level of automation increases and multiple failures need to be tolerated. In contrast, we also focus on the primary-backup approach which enables fault-tolerance solutions with optimized resource usage by activating some backups only when failures occur.

Fault-tolerant task allocation using a combination of active replication and the primary-backup approach has been studied in [15] and [3]. Both techniques introduce phasing delays to support backup overlapping and backup deallocation techniques. Neither technique leverages the lower run-time utilization of different types of passive backups to optimize the number of processors used for deployment. Also, all techniques mentioned so far attempt to meet immediate recovery-time requirements. In this paper, we allow each task to specify its own configurable recovery time requirement.

In our previous work [4, 19], we discussed the fault-tolerant task allocation problem which states that no task should be placed on the same node as its primary or another redundant task. We proposed the Tiered Placement Constrained Decreasing (TPCD) and TPCD with cold standby (TPCDC) heuristics to produce allocations respecting this fault-tolerant placement constraint. Both these heuristics assume the type of redundant task to be used as inputs. In this paper, we attempt to determine this parameter given the recovery time requirement of each task. To this end, we present a recovery-time analysis framework, along with an extension to the TPCDC heuristic, TPCDC+R, and two new heuristics to produce a fault-tolerant allocation satisfying the recovery-time requirement of each task.

3 System Model and Problem Definition

3.1 Computation Model

In this paper, we consider a distributed system consisting of N computational nodes, where each node can communicate with every other node in the system by sending messages. We assume a set of n tasks, $(\tau_1, \tau_2, ..., \tau_n)$, where each task is assigned a unique priority based on the Rate-Monotonic Scheduling (RMS) [23] Policy. We assume that the tasks are ordered in non-increasing order of priorities. We assume that a higher-priority task can immediately preempt a lower-priority task. Each task τ_i is assumed to have a worst-case execution time ($WCET$) of C_i, a period of T_i and an implicit deadline $D_i = T_i$. The analysis can be adapted

to other scheduling policies and deadline models (e.g., $D < T$), as long as a response-time analysis is available. Each task τ_i may be blocked by lower-priority tasks for at most B_i units of time as a result of the operation of a concurrency control protocol like the Priority Ceiling Protocol [33]. We assume that the worst-case release jitter, the worst-case time a task τ_i can spend waiting to be released after arrival, is J_i [36].

The schedulability of a task can be evaluated using the response-time analysis presented in [36].

$$\begin{aligned} r_i^{n+1} &= C_i + B_i + \sum_{j=1}^{i-1} \lceil (r_i^n + J_j)/T_j \rceil C_j \\ r_i^0 &= \sum_{j=1}^{i} C_j \end{aligned} \tag{1}$$

Equation (1) represents an iterative solution which starts at r_i^0 and terminates when either $R_i = r_i^{n+1} = r_n$ or $r_i^{n+1} > D_i$. We refer to R_i as the worst-case response time for task τ_i. R_i is measured from the instant the task is released to its completion. The worst-case time from arrival to completion of task i [36], also known as the worst-case completion time ($WCCT$), is given by,

$$WCCT_i = r_i + J_i \tag{2}$$

A task is said to be schedulable if its $WCCT_i \leq D_i$.

3.2 Fault Model

In our model, we are primarily concerned about permanent crash faults [9]. Hardware failures, operating system crashes and process crashes are some examples of crash faults. We assume that these crash faults are fail-silent [5]. In order to tolerate these crash faults, we employ fault tolerance by replication [14]. We consider three types of redundancies:

1. **Active Replica:** In active replication, all redundant copies are identical and treated uniformly. Each replica performs all operations, like accepting and processing application inputs, performing state calculations, performing application calculations and producing output. This implies that, under normal operation, the system needs to support duplicate suppression to filter out duplicate outputs.
2. **Hot Standby:** A hot standby is based on the primary-backup approach. It performs all the operations of the primary task except for producing outputs. On detection of primary failure, the hot standby is promoted to become a primary and begins to produce outputs. Unlike an active replica, a hot standby can run a degraded version of the primary to optimize resource consumption.
3. **Cold Standby:** A cold standby is also based on the primary-backup approach. It can be of two types depending on the type of application. If an application is *stateless*, the cold standby does not perform any operations until it detects primary failure. For applications with state, the cold standby accepts and logs application inputs but does not perform any other operations. It regularly accepts the state from the primary to maintain consistency. On detection of primary failure, the cold standby primes its state first and then begins to produce outputs.

Transient and intermittent faults can be overcome by techniques like simple re-execution, forward recovery [7], and recovery blocks [17]. The impact of these solutions can be accounted for by modifying the analysis of task response times to include additional fault-induced

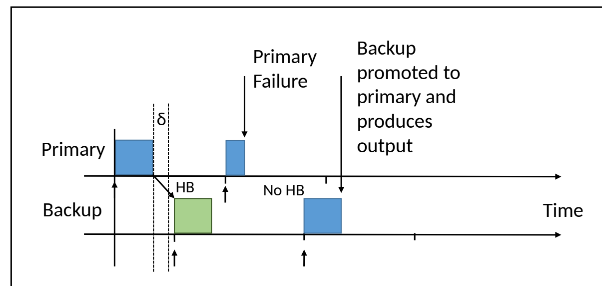


Figure 1 Detecting Primary Failure.

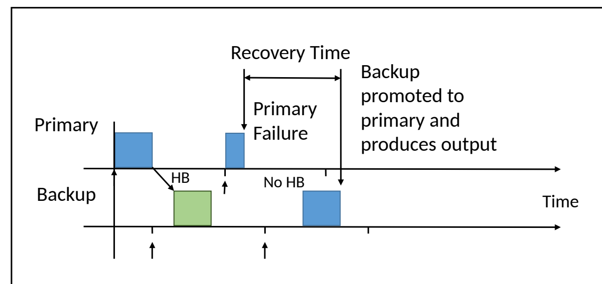


Figure 2 Defining Recovery Time.

processing requirements [7]. In this paper, we focus on permanent faults, though the analysis for transient faults from [7] can be incorporated into our framework. Similar fault models have also been used in the automotive sector [19] and [22].

In our model, a task and its replicas have the same period and a task can be assigned one or more replicas based on the application requirements. The system designer can decide which tasks are considered *critical* for the application and which are considered *non-critical*. In this paper, we assume that non-critical tasks do not have replicas and can be terminated in order to allow a cold standby to execute when a primary fails. For fault detection, we assume that the replicas monitor the status and health of the primary, for example, by using heartbeats and producing outputs when necessary [19, 4]. This is illustrated in Figure 1. We assume that the underlying communication framework is reliable[1], i.e., it guarantees that a message will either be delivered within a fixed message delivery bound δ or not be delivered at all. Common communications protocols like CAN/CAN-FD [10], FlexRay [28] and many variants of real-time Ethernet [1, 34] can support these guarantees. The successful reception of a heartbeat indicates to the replica that the primary is operational.

3.3 Problem Statement

▶ **Definition 1.** *Recovery time* (*RT*) is the time elapsed from the instant of primary failure to the instant when a redundant task is able to produce the desired output. This duration is shown in Figure 2.

[1] Safety-critical real-time systems must deal with communication failures. The communication layer can utilize solutions like redundant CANbus links, dual FlexRay configurations with built-in support for fault tolerance, and replicated ethernet switches. In the interests of brevity, we abstract away the details of such solutions with our assumption of a reliable communication layer in this paper.

The choice of the type of redundant task to be used has a major impact on a task's recovery time. For example, an active replica can virtually provide seamless recovery since it runs alongside the primary. The hot and cold standbys, on the other hand, have to first detect primary failure. In addition, the cold standby needs to then prime its state, which results in an even longer recovery time.

The number of redundant copies assigned to each task is also an important design parameter. Every task can be assigned m ($m \in \mathbf{N}$) redundant copies. It is important to note that different tasks can utilize different redundancy types (i.e., active, hot or cold). The number of replicas and their types are system parameters which are application-specific. Their choice determines the number of failures a given task can tolerate and how quickly a task can recover from a failure. The former is a system designer's choice and the latter can be captured by specifying a recovery-time requirement for each task.

▶ **Definition 2.** *Recovery time requirement* (RTR) is the maximum number of consecutive deadlines of the primary task that the system can afford to miss before the redundant task must recover in accordance with Definition 1.

We first determine which type of redundant task is appropriate for a given task to meet its recovery-time requirement. The benefit of using replicas is maximal when a task and any of its redundant copies obey the placement constraint of not being co-located on the same node. To this end in [20], Kim et al. defined the *Fault-tolerant Partitioned Scheduling* problem as one of assigning independent tasks to nodes where every member of a group, i.e., a primary task and its copies, would not be co-located on the same node. This ensures that, when nodes fail independently, they do not result in application failures. The bin-packing problem [25] of allocating fault-tolerant tasks is known to be NP-hard [18], and heuristics were proposed in [4] to address this problem. In this paper, we extend these heuristics to ensure that the recovery-time requirements of tasks are also satisfied.

We assume that task I/O dependencies[2] and ensuring input consistency between a primary and its redundant copies are considered by the system designer in assigning the RTR of each task in the system.

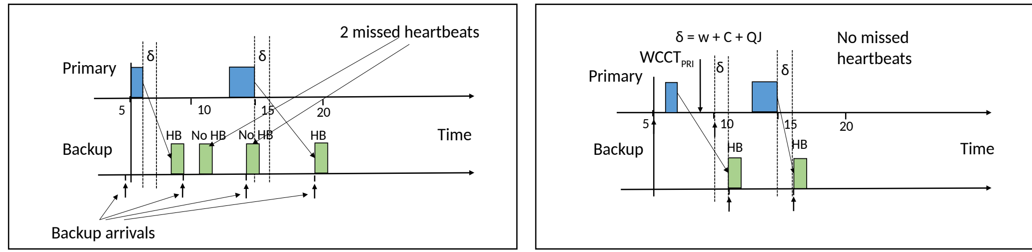
To summarize, the goals of this paper are as follows. Given N nodes, and a task set $\tau = \{T_1, T_2, \ldots, T_n\}$, where every task has an application-dependent recovery-time requirement RTR_i,

1. derive bounds on the recovery time for each redundant-task type,
2. decide a redundant-task type, i.e., active, hot or cold, and
3. find an allocation where all tasks satisfy their recovery-time requirements while minimizing the number of nodes used for allocation.

4 Recovery Time Analysis for Passive Backups

In the previous section, we saw that an active replica can be seamlessly recovered from, since other replicas are running in parallel. In this section, we derive the recovery time bounds for hot and cold standbys.

[2] Detailed task models capturing I/O dependencies are certainly needed, and will be part of our future work. For example, task I/O dependencies can be factored into our analysis by constructing composite (virtual) tasks formed by combining tasks with I/O dependencies.



(a) Backup not following the Primary. **(b)** Backup following the Primary.

Figure 3 Motivation for Backup following the Primary.

4.1 Backup Following the Primary

Previous work [4] has shown that the bounds on the recovery time for passive backups can be reduced if the backup task execution follows the execution of the primary. The intuition for this can be seen in Figures 3a and 3b. As seen in Figure 3a, if the backup can execute at any time independent of its primary, it is possible for a backup to miss up to two heartbeats without primary failure. Hence, the backup must wait for three consecutive missed heartbeats to declare failure of the primary and initiate recovery, resulting in a longer recovery time. In contrast, when the backup follows the execution of the primary, it needs only a single missed heartbeat to detect primary failure.

For the backup to follow the primary, the following requirements must be satisfied:

1. *Global Time Synchronization*: To ensure that the backup follows the primary, the release time of the backup w.r.t that of the primary must be explicitly controlled. Since fault-tolerant task allocation requires primaries and replicas to run on distinct nodes, the nodes must be time-synchronized. This constraint can be relaxed in a system which allows tasks to be released with offsets at boot up and has negligible clock drift.
2. *Network Schedulability Analysis*: In order to calculate the optimal release instant for the backup, network delays must be characterized. [3] The worst-case network response time δ_m for message m can be represented as,

$$\delta_m = w_m + QJ_m + C_m \tag{3}$$

where,

- The queuing jitter QJ_m corresponds to the longest time between the initiating event and the message being queued, ready to be transmitted on the network.
- The queuing delay w_m corresponds to the longest time that the message can remain in the device queue, before commencing successful transmission on the network.
- The transmission time C_m corresponds to the longest time that the message can take to be transmitted. In the case of standbys, the transmission time depends on the standby type. Cold standbys need to accept state, and normally require longer transmission times than hot standbys.

4.2 Recovery Time Bounds for Hot Standbys

A hot standby produces an output immediately after it detects primary failure as described in Section 3 and shown in Figure 4. Let $WCCT_{pri}$ be the $WCCT$ for the primary. Let

[3] Popular automotive network technologies, like CAN [10] and FlexRay [28], have response-time analyses to bound the worst-case message delivery time.

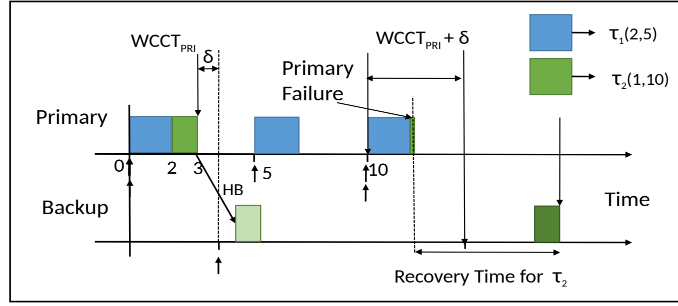


Figure 4 Recovery Time Bounds for Hot Standby τ_2.

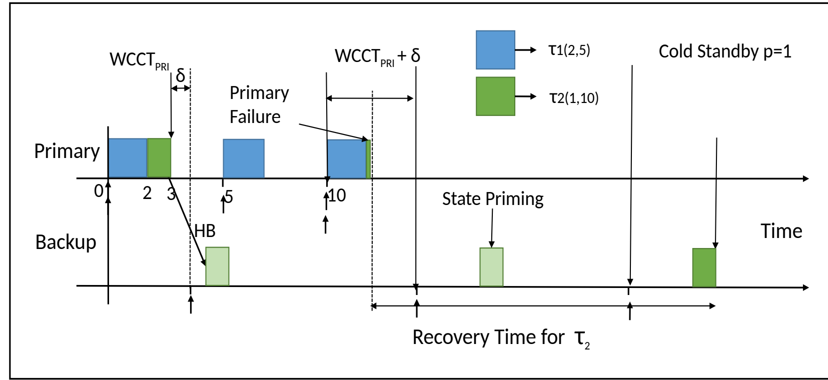


Figure 5 Recovery Time Bounds for Cold Standby τ_2.

$WCCT_{hot}$ be the completion time of the backup corresponding to the failure of the primary. The total time from the release of the primary to the execution of the backup would be $WCCT_{pri} + \delta_{hot} + WCCT_{hot}$. Hence, the recovery time is

$$RT_{Hot} = WCCT_{pri} + \delta_{hot} + WCCT_{hot} \tag{4}$$

4.3 Recovery Time Bounds for Cold Standbys

A cold standby takes longer to recover from a failed primary as described in Section 3, since it does not produce any state of its own, but instead receives regular state updates from the primary. This is illustrated in Figure 5. It only logs application inputs which it uses to prime state for future use. These logs can be cleared once a primary state is applied. Let p denote the number of periods the cold standby needs to prime state and produce output. A cold standby for a stateless application does not need to prime any state, and hence, in this case, $p = 0$. For applications with state, the value of p depends on two factors:

1. The frequency of state transfer from the primary to the standby: The higher the frequency of state transfer, the fresher is the state of the cold standby and hence lower is the number of periods required for state priming (i.e., a lower value for p).
2. Priming state is highly application-dependent. Some applications may make temporal corrections of the most recent state using appropriate extrapolations. Other applications may iterate through all the logged inputs between the last received state and the time instant the failure is detected, and, in each iteration, re-calculate the state to finally produce output based on fresh state. In this paper, we assume that, for applications with state, the value of p is provided by the application designer.

Thus, for a cold standby to recover from a primary failure, the recovery time would be

$$RT_{Cold} = WCCT_{pri} + \delta_{cold} + pT + WCCT_{cold} \tag{5}$$

5 Redundant-Task Type Assignment To Tasks

In this section, we identify the types of redundant task assignments that can satisfy a given RTR constraint. As described in Section 3.3 an active replica can be seamlessly recovered from, since other replicas are running in parallel, hence it can satisfy any RTR requirement.

5.1 Hot standby

5.1.1 $RTR = 0$

For a hot standby to recover from primary failure and maintain $RTR = 0$, the recovery time, RT_{hot}, should be less than or equal to T, i.e., the redundant task must recover before the deadline of its primary. Hence, from Equation (4) we have,

$$RT_{Hot} \leq T \quad \Rightarrow \quad WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq T \tag{6}$$

With the worst-case values for the terms in Equation (6),

$$T + \delta_{hot} + T \nleq T$$

Hence, with the worst-case values for $WCCT$, a hot standby cannot satisfy $RTR = 0$.

5.1.2 $RTR > 0$

Consider the case where $RTR = n$ and $n \in \mathbb{N}_{>0}$, allowing the task to tolerate up to n missed deadlines when the primary fails.

In the case of a hot standby, $RTR = n$ can be satisfied if $RT_{Hot} < (n+1)T$.
From Equation (4),

$$WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq (n+1)T \tag{7}$$

Considering $n \geq 2$ and the worst-case values for the terms in the above equation,

$$\begin{aligned} WCCT_{pri} + \delta + WCCT_{bkp} &\leq 3T \\ T + \delta + T &\leq 3T \end{aligned} \tag{8}$$

Assuming $\delta < T$, a hot standby can meet $RTR \geq 2$ (if it is schedulable).

5.2 Cold standby

5.2.1 $RTR = 0$

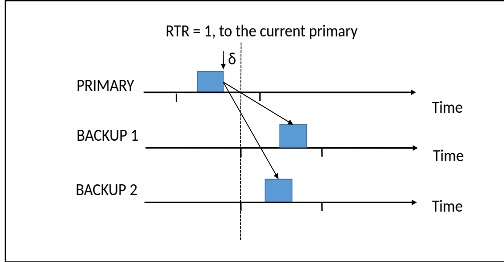
For a cold standby to satisfy $RTR = 0$, the recovery time should be less than or equal to T. From Equation (5),

$$WCCT_{pri} + \delta_{cold} + pT + WCCT_{cold} \leq T \tag{9}$$

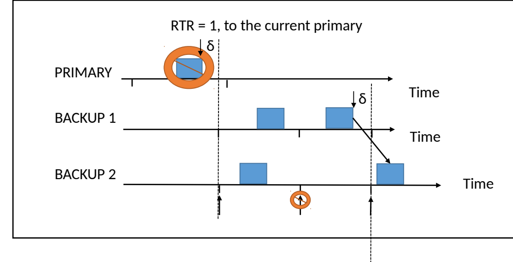
Equation (9) must be satisfied for a cold standby to meet $RTR = 0$. However, if $p \neq 0$, a cold standby cannot satisfy $RTR = 0$.

Table 1 Conditions for Redundant Task Selection.

Standby Selection		
RTR(n)	**Condition**	**Standby Assignment**
0	$WCCT_{pri} + \delta_{cold} + WCCT_{cold} \leq T$	Cold ($p = 0$)
0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq T$	Hot
0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} > T$	Active
> 0	$WCCT_{pri} + \delta + WCCT_{bkp} + pT \leq (n+1)T$	Cold
> 0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq (n+1)T$	Hot
> 0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} > (n+1)T$	Active



(a) Multi-Level Backups.



(b) Release Time Correction.

Figure 6 Support for Multi-Level Backups.

5.2.2 $RTR > 0$

In the case of a cold standby, $RTR = n$ can be satisfied if $RT_{Cold} < (n+1)T$ in the worst case.

From Equation (5),

$$WCCT_{pri} + \delta + WCCT_{bkp} + pT \leq (n+1)T \tag{10}$$

Table 1 summarizes all the conditions above for standby selection. We see that, for certain conditions, multiple options are available for redundant-task type assignment. We describe our approach to redundant task selection in case of multiple available options in Section 6.

5.3 Multi-Level Backups

As shown in Figure 6a, a single primary can have more than one backup. Both backups in the figure are released such that they follow the primary to satisfy the primary's recovery-time requirement. We assume that the order of promotion to primary is statically configured (which in practice is easily achieved by the use of configuration parameters, or using node IDs). Suppose that the first backup in Figure 6a is designated to take over execution first after primary failure. On primary failure, it is not guaranteed that the current second backup would always satisfy the recovery time requirements of the first which would now become the new primary. In order for the second-level backup to now satisfy the RTR of the new primary, the release time of the task needs to be corrected and this is shown in Figure 6b. Also, since we are delaying the release time of the task, and deadlines are therefore correspondingly postponed, the deferred start does not affect the overall schedulability of the task set [30].

Algorithm 1 TPCDC+R.

1: **procedure** TPCDC+R $(\Gamma = \{\tau_0^0, \tau_0^1, ...\tau_1^0...\tau_n^0, ...\})$ ▷ $(\tau_j^i : j \rightarrow TaskId, i \rightarrow TierOrder)$
2:     **for** each task τ_j in Γ **do**
3:         $\Psi_i \leftarrow \tau_j^i$ ▷ Create tiers consisting of tasks with redundancies of the same order
4:     **for** each tier Ψ_i in Ψ **do**
5:         Sort tasks in descending order of their utilizations
6:         **for** each task τ_i in Ψ_i **do**
7:             Check recovery time to primary and assign redundant-task type
8:             Task Assignment$(\alpha) \leftarrow$ BFD-P(τ_i)
9:     Apply lower run-time utilizations for cold standbys
10:     Allocate the tasks that do not have redundancies
11:     **return** α ▷ Return the task set assignment

---

6 Task partitioning with Recovery Time Constraints

In Section 3, we presented the fault-tolerant task allocation problem. We now extend this problem to include the constraint that every backup task satisfies the recovery-time requirement of the primary. Given our focus on resource-constrained environments, we present heuristics to address this problem while trying to minimize the number of processors used for allocation. Based on the recovery-time bounds of Section 4.2, we derived conditions to determine the standby type in Section 5. In this section, we look at how the redundant-task type assignment can be incorporated in the task allocation scheme to satisfy the recovery time requirement of each primary.

The TPCD heuristic [4] produces an allocation satisfying the fault-tolerant placement constraint while attempting to minimize the number of nodes used. TPCD breaks the task set into tiers based on the backup order to place members of a replica group as far away from each other in the task order as possible. This reduces the chances of a task facing a placement conflict. In each tier, TPCD arranges tasks in descending order of utilization values, since, members of larger groups have a greater probability of running into a placement conflict. TPCD then allocates the tiers from the highest-order tier to the lowest-order tier. The TPCDC heuristic extends TPCD to leverage lower cold-standby utilizations. Any non-critical task can be terminated in order to allow a cold standby to execute when a primary fails. TPCDC initially treats all standbys as hot standbys from a utilization standpoint.

6.1 The TPCDC+R Heuristic

We now extend TPCDC by introducing an explicit check for RTR. This TPCDC+R heuristic is shown in Algorithm 1. Before assigning a task to a node, we ensure that every task (primary or copy) on that node satisfies RTR constraints. In order to determine the recovery time of a redundant task, we must first assign the redundant-task type using Table 1. Since cold standbys at run-time have very low utilization values, it allows for an optimization where non-safety critical tasks can be assigned to processors with cold standbys which can be terminated in case the cold standby needs to take over primary execution. Hence, if multiple redundant task options are available, we prioritize cold standbys over hot standbys and active replicas because they are the most resource-efficient. Next, hot standbys do not normally produce outputs. Hence, the overhead for duplicate suppression is avoided and hot

Algorithm 2 TRTI.

```
1: procedure TRTI (Γ = {τ_0^0, τ_0^1, ....τ_1^0...τ_n^0, ...})      ▷ (τ_j^i : j → TaskId, i → TierOrder)
2:     for each task τ_j in Γ do
3:         Ψ_i ← τ_j^i    ▷ Create tiers consisting of tasks with redundancies of the same order
4:     for each tier Ψ_i in Ψ do
5:         Sort tasks in ascending order of RTR constraints
6:     for each task τ_i in Ψ_i do
7:         Check recovery time to primary and assign redundant-task type
8:         Task Assignment(α) ← BFD-P(τ_i)
9: Apply lower run-time utilizations for cold standbys
10: Allocate the tasks that do not have redundancies
11: return α                                  ▷ Return the task set assignment
```

Algorithm 3 RTT.

```
1: procedure RTT(Γ = {τ_0^0, τ_0^1, ....τ_1^0...τ_n^0, ...})
2:     for each task τ_j in Γ do
3:         Ψ_i ← τ_j^i                 ▷ Create tiers consisting of tasks of same RTR
4:     for each tier Ψ_i in Ψ do
5:         TPCDC+R(Ψ_i)
6:     return α                        ▷ Return the task set assignment
```

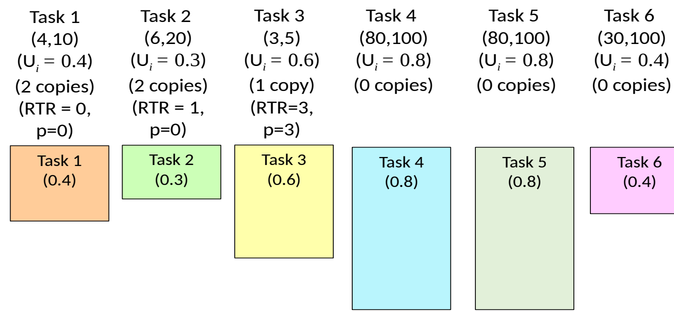
standbys can potentially run a degraded version of the primary with lower utilization values. However, they may have a scheduling penalty since they need to satisfy RTR constraints. Therefore, the heuristic first checks if the hot standby satisfies the RTR constraint of the task. If so, it assign a hot standby. Else, it chooses an active replica instead of opening a new node for assignment.

It must be noted that the choices among three redundant-task types would be different if the goal was different. For example, if communication bandwidth is constrained, the cold standby overheads for state transfer need to be factored in.[4]

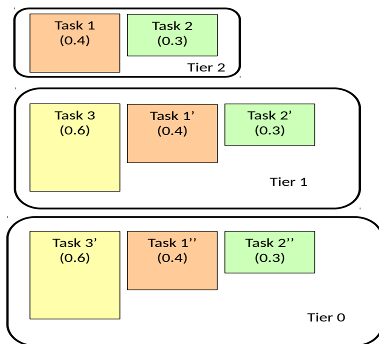
As stated before, we prioritize cold standbys over hot standbys or active replicas. Figure 8a shows the distribution of standby types produced by TPCDC+R. We plot the percentage of active, hot or cold redundant task assignments against the number of primary tasks in each task set. The results are averaged across 50,000 tasksets, where tasks are randomly generated. Each task is randomly assigned 0,1 or 2 redundancies, an RTR constraint from 0 to 5, and a value for p (i.e., periods for cold standby priming) from 0 to 5.

TPCDC+R prioritizes tasks with higher utilization values by assigning them first in the task allocation order for each tier. This introduces additional placement constraints for tasks which have tight RTR requirements. An example occurs when a task with low utilization with strict RTR requirements gets placed later in the allocation order. As a result, cold standbys may become unschedulable forcing the use of active replicas, which in turn can cause new nodes to be added.

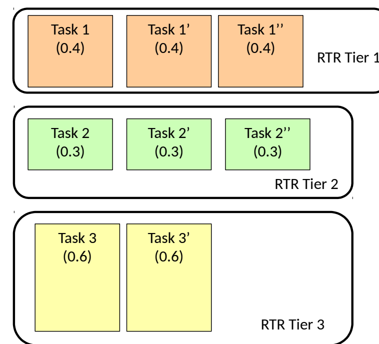
[4] We will consider this overall system resource optimization problem as part of our future work.



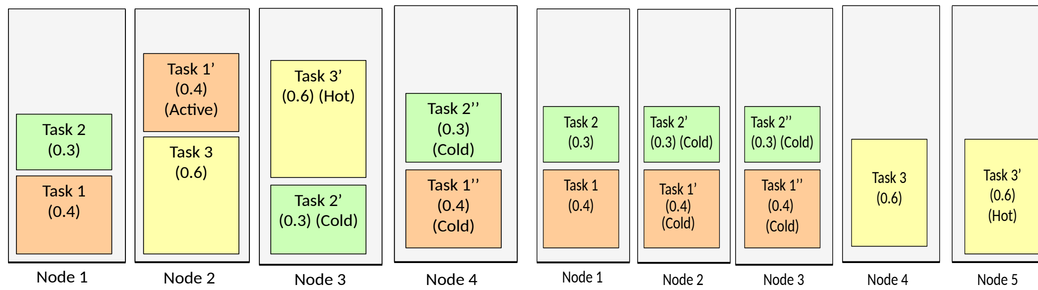
(a) Input Task Set.



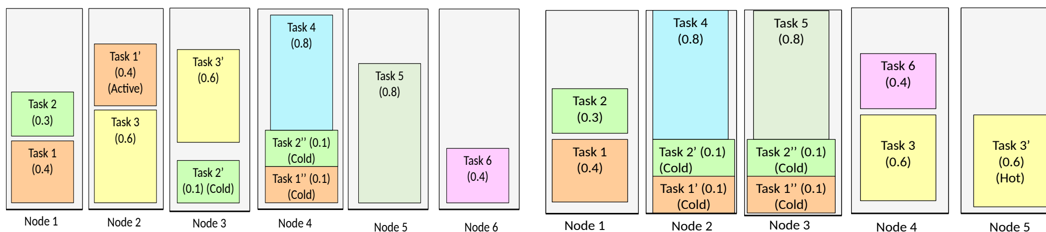
(b) TPCDC+R Tiering.



(c) RTT Tiering.

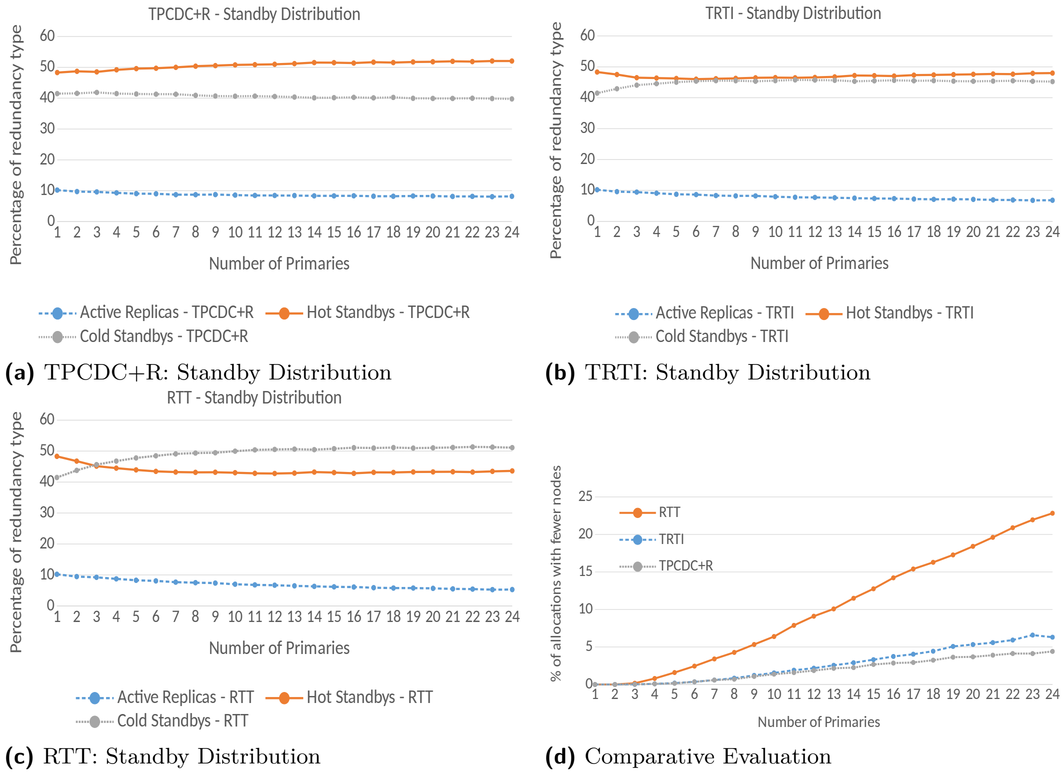


(d) TPCDC-R+ critical task allocation. **(e)** RTT critical task allocation.



(f) TPCDC-R+ non-critical task allocation. **(g)** RTT non-critical task allocation.

Figure 7 Example: TPCDC-R+ vs RTT (Best Viewed In Color).



(a) TPCDC+R: Standby Distribution

(b) TRTI: Standby Distribution

(c) RTT: Standby Distribution

(d) Comparative Evaluation

Figure 8 Evaluation: RTT vs TRTI vs TPCDC+R

To address this problem, we introduce two new heuristics based on TPCDC+R that prioritize RTR constraints in the task allocation order.

1. In the first heuristic, we order tasks within each tier of TPCDC by their RTR requirements instead of utilization values. We refer to this extension as the *Tiered RTR constraint Increasing* (TRTI) heuristic. Algorithm 2 captures this TRTI heuristic.

2. In the second heuristic, we divide tasks into groups with different RTR requirements and allocate each group using the TPCDC heuristic separately. We refer to this as the *RTR-tiered* (RTT) heuristic. Algorithm 3 presents this heuristic.

Figure 7 depicts an example highlighting how prioritizing RTR constraints in the task allocation order can improve resource utilization by comparing the outputs of the TPCDC+R and the RTT heuristics for the input task set in Figure 7a. As shown in Figure 7b, TPCDC+R breaks the critical tasks into tiers based on the number of backups and orders tasks within a tier based on their utilization values. In contrast, RTT breaks the tasks into tiers based on their RTR constraints. Figures 7e and 7d show that the RTT heuristic allocates a greater number of cold standbys compared to the TPCDC-R heuristic. This, in turn, results in an allocation with fewer nodes as seen in Figures 7f and 7e. Notice that, when allocating the non-critical tasks, we consider the lower utilization values for the cold standbys.

Figures 8b and 8c show the standby distributions for TRTI and RTT heuristics. Both the heuristics result in a larger number of cold standby allocations than for the TPCDC+R heuristic.

6.2 Evaluation and Discussion

In this section, we evaluate and compare the performance of the TPCDC+R, TRTI and RTT heuristics. We also evaluate the impact of the increased cold standby allocation on the number of nodes used for allocations using the new heuristics. We plot the percentage of task sets for which a heuristic produces an allocation with fewer nodes, i.e., uses at least one node less for allocation compared to the other two heuristics. Figure 8d presents the results for 50,000 randomly-generated tasksets generated using Stafford's `Randfixedsum` algorithm [11] for total utilization values ranging from 0.1 to number of primaries and random period values ranging from 1 to 10^4. Each task is randomly assigned 0, 1 or 2 copies, an RTR constraint from 0 to 5, and a value for p (i.e., periods for cold standby priming) from 0 to 5. As the figure shows, RTT produces an allocation with fewer nodes on average when compared to the TRTI and TPCDC+R. For task sets with 24 primaries, it produces an allocation with fewer nodes than TRTI and TPCDC+R for almost 23% of the task sets. This is consistent with the intuition that increasing the number of cold standbys reduces CPU resource utilization. Also, as the number of primaries increase, this trend becomes more significant as we have more cold standby assignments to leverage. Moreover, both heuristics that prioritize the RTR constraints perform better than the TPCDC+R heuristic. It is important to note that increasing the number of cold standbys will result in additional network latencies since they need to have state information sent to them from their primaries. For the purpose of these experiments, we assume that the delays incurred for state transfer are short. For a network-constrained system, it may prove to be more advantageous to have a lower number of cold standbys.

7 Applying Simulated Annealing to the Fault-Tolerant Task Allocation Problem

In the previous section, we saw that the RTT heuristic on average produces a better solution than TPCDC+R and TRTI. In this section, we look at further improving on the RTT heuristic solution by utilizing the simulated annealing method to solve the fault-tolerant task allocation problem instead.

Simulated annealing is a general-purpose combinatorial optimization technique first proposed by Kirkpatrick et al. [21]. The fault-tolerant task assignment problem can be stated as an optimization problem as follows,

Given n tasks $(\tau_1, \tau_2, ..., \tau_n)$, with *utilization* $(u_1, u_2, ..., u_n)$, where $u_i \leq 1$, find the number of nodes M of size 1 that are needed to pack all tasks such that a primary task and its corresponding redundant copies obey the placement constraint of not being co-located on the same node and optimizing the following cost function [12]

$$cf = \text{Maximize} \quad \sum_{j=1}^{M} (\sum_{i \in k_j} u_i)^2 \tag{11}$$

where, k_j represents the set of tasks in bin j.

The simulated annealing algorithm for fault-tolerant task allocation is shown in Algorithm 4. The algorithm starts by using the RTT heuristic to create an initial allocation, α. We use this as the *initial state* of the system. To obtain a new state α' from the initial state we randomly perform one of the two operations described in Section 7.1. While performing either of these operations, we ensure that the placement constraints for all tasks remain satisfied. We also ensure that the new allocation is schedulable. Here, we apply a greedy

Algorithm 4 Simulated Annealing.

1: **procedure** ANNEAL $(\Gamma = \{\tau_0^0, \tau_0^1, ...\tau_1^0...\tau_n^0, ...\})$
2: Task Assignment(α) = RTT(Γ)
3: $T \leftarrow T_\infty$
4: **while** $T > T_0$ **do**
5: **repeat**
6: α' = RANDOMLYMODIFYCURRENTSOLUTION(α)
7: $\Delta C = cf(\alpha) - cf(\alpha')$ ▷ From Eqn 11
8: $\eta = \text{RANDOM}(0, 1)$
9: $P(\Delta C) = e^{(-\Delta C/T)}$
10: **if** $\Delta C < 0$ or $P(\Delta C) > \eta$ **then**
11: $\alpha = \alpha'$
12: **until** thermal equilibrium
13: $T \leftarrow F(T)$
14: **return** α ▷ Return the task set assignment

optimization: if a valid operation results in an empty bin, we remove it from the allocation[5]. The value of the objective function is calculated for this new state. Let ΔC represent the change in the cost function, i.e, $\Delta C = cf(\alpha) - cf(\alpha')$. This state is unconditionally accepted if $\Delta C < 0$. If not, the Metropolis condition [24] is applied and the state is accepted with a probability according to the following acceptance function $P = e^{(-\Delta C/T)}$. We start with a large value for initial temperature $T = T_\infty$. When there is no appreciable change in the value of the cost function across a few chains of computation or a maximum number of iterations is reached, we lower the temperature. The annealing terminates when the temperature T reaches a low-enough value, T_o and the current best α is returned as the solution. We derive the values for T_∞ and T_o for the fault-tolerant task allocation problem in Section 7.2.

7.1 Generating Random Solutions

In order to create random solutions from a given solution, we apply the following two operations [29].

1. We randomly move a single task from a randomly-selected node k to another randomly selected node l.

   ▶ **Lemma 3.** *The maximum reduction ΔC_{max} for the cost function in Equation 11, for a system of two nodes, k and l, by moving a task from node k to l occurs when $U_k = 1$ and $U_l = 0$, where U_k and U_l are the total utilization values of the respective nodes.*

   Proof. Let u_t represent the utilization of the task that is moved from bin k to l. Let U'_k and U'_l be the transformed utilization values after a task is moved from node k to l. Hence, $U'_k = U_k - u_t$ and $U'_l = U_l + u_t$ and ΔC for this operation can be represented as,

$$\begin{aligned} \Delta C = {U_k}^2 + {U_l}^2 - {U'_k}^2 - {U'_l}^2 &= {U_k}^2 + {U_l}^2 - (U_k - u_t)^2 - (U'_l + u_t)^2 \\ &= 2 * U_k * u_t - 2 * U_l * u_t - 2 * u_t^2 \end{aligned} \tag{12}$$

[5] In our experiments, we found no significant improvement in the quality of solutions obtained by retaining an empty bin

From Equation (12), ΔC is maximum when the positive terms are maximized and the negative terms are minimized. U_l only appears in the second term which is negative, and U_k appears only in the first term which is positive. Hence ΔC is maximized when $U_k = 1$ and $U_l = 0$ corresponding to their maximum and minimum possible values. ◀

For the fault-tolerant task allocation problem, moving a task from one bin to another can result in a different redundant-task-type assignment resulting in different run-time utilizations. Let the factor s capture this utilization change. The associated change in the cost function for this operation is given by,

$$\begin{aligned}\Delta C =& U_k^2 + U_l^2 - [(U_k - u_t)^2 + (U_l + s * u_t)^2] \\ =& 2 * U_k * u_t - u_t^2 - 2 * U_l * s * u_t - (s * u_t)^2\end{aligned} \tag{13}$$

From Lemma 3, the maximum value of ΔC, which represents the largest reduction in the cost function, occurs when a task is moved from a completely-packed node to a completely-empty node. Since we apply a greedy optimization of removing empty bins, we consider $U_l = \epsilon$. Hence,

$$\Delta C_{max1} \approxeq 2 * u_t - u_t^2 - (s * u_t)^2 \tag{14}$$

2. We randomly select two tasks currently located in two different bins and swap them.

▶ **Lemma 4.** *The maximum reduction ΔC_{max} for the cost function in Equation 11, for a system of two nodes, k and l, by swapping two tasks occurs when one of the nodes has $U = 1$ and the other has $U = \epsilon$.*

Proof. Let U_k and U_l be the total utilization values of the respective nodes. Let u_{t1} represent the utilization of the task that is moved from bin k to l and u_{t2} represent the utilization of the task that is moved from bin l to k. Let U'_k and U'_l be the transformed utilization values after the tasks are swapped. Hence, $U'_k = U_k - u_{t1} + u_{t2}$, $U'_l = U_l + u_{t1} - u_{t2}$ and ΔC for this operation can be represented as,

$$\begin{aligned}\Delta C &= {U_k}^2 + {U_l}^2 - {U'_k}^2 - {U'_l}^2 \\ &= {U_k}^2 + {U_l}^2 - (U_k - u_{t1} + u_{t2})^2 - (U_l + u_{t1} - u_{t2})^2 \\ &= 2 * U_k * u_{t1} - 2 * U_k * u_{t2} + 2 * u_{t1} * u_{t2} - {u_{t1}}^2 - {u_{t2}}^2 \\ &+ 2 * U_l * u_{t2} - 2 * U_l * u_{t1} + 2 * u_{t1} * u_{t2} - {u_{t1}}^2 - {u_{t2}}^2 \\ &= 2 * U_k * (u_{t1} - u_{t2}) - 2 * U_l * (u_{t1} - u_{t2}) - 2 * (u_{t1} - u_{t2})^2 \\ &= 2 * (U_k - U_l) * (u_{t1} - u_{t2}) - 2 * (u_{t1} - u_{t2})^2\end{aligned} \tag{15}$$

From Equation (15), ΔC is maximum when $U_k - U_l \approxeq 1$, since $0 < U_k, U_l \leq 1$. Since we are swapping tasks between two nodes, a node cannot be empty. Hence, ΔC is maximized when one node has $U = 1$ and the other $U = \epsilon$. ◀

For our fault-tolerant task allocation problem, let the factors s_{t1} and s_{t2} capture the utilization changes after the swap. The associated change in the cost function for this operation is given by,

$$\begin{aligned}\Delta C =& U_k^2 + U_l^2 - [(U_k - u_{t1} + s_{t2} * u_{t2})^2 + (U_l + s_{t1} * u_{t1} - u_{t2})^2] \\ =& 2 * U_k * u_{t1} - 2 * U_k * s_{t2} * u_{t2} + 2 * u_{t1} * s_{t2} * u_{t2} - u_{t1}^2 - (s_{t2} * u_{t2})^2 + \\ & 2 * U_l * u_{t2} - 2 * U_l * s_{t1} * u_{t1} + 2 * u_{t2} * s_{t1} * u_{t1} - u_{t2}^2 - (s_{t1} * u_{t1})^2\end{aligned} \tag{16}$$

From Lemma 4, the cost function is maximized when one bin has $U = 1$ and the other has $U = \epsilon$. Hence,

$$\begin{aligned} \Delta C_{max2} \approxeq & 2 * u_{t1} - 2 * s_{t2} * u_{t2} + 2 * u_{t1} * s_{t2} * u_{t2} - u_{t1}^2 - (s_{t2} * u_{t2})^2 + \\ & + 2 * u_{t2} * s_{t1} * u_{t1} - u_{t2}^2 - (s_{t1} * u_{t1})^2 \to U_k = 1, U_l = \epsilon \end{aligned} \quad (17)$$

Given a task set, the value of $\Delta C_{max} = max(\Delta C_{max1}, \Delta C_{max2})$ can be easily calculated by substituting actual values into Equations (14) and (17) for all combinations of tasks.

7.2 Selecting an Annealing Schedule

The annealing schedule is described by quantitative choices for the three parameters: the starting value of the temperature, T_∞, the stopping value of the temperature T_o, and the decrement function $F(T)$ which determines the profile of the temperature from the beginning till the end of the annealing process.

The starting temperature, T_∞, for a good annealing schedule, is usually determined by monitoring the acceptance ratio at each temperature. The upper bound for acceptance ratio a_h (the fraction of generated states that are accepted), is arbitrarily fixed at some high value such as 0.9 and the temperature is increased to a value where this acceptance ratio is achieved [29]. Given that we can calculate ΔC_{max} for a given task set, we can calculate the value T_∞, which can accommodate even the largest reduction in the cost function at high temperatures, as follows.

$$a_h = e^{(-\Delta C_{max}/T_\infty)} \Rightarrow ln(1/a_h) = \Delta C_{max}/T_\infty \Rightarrow T_\infty = \Delta C_{max}/ln(1/a_h) \quad (18)$$

Similarly, T_o can be calculated for the lower bound of the acceptance ratio a_l.

$$T_o = \Delta C_{max}/ln(1/a_l) \quad (19)$$

In our experiments, we also found that $F(T) = 0.9 * T$ works well for the problem at hand.

7.3 Evaluation

In this section, we compare the performance of the simulated annealing approach with that of the RTT heuristic. We plot the execution time of the simulation annealing approach and the RTT heuristic against the number of the primaries in the task set. Figure 9 presents the results averaged across 5000 randomly-generated task sets. Each task is randomly assigned 0, 1 or 2 redundancies, an RTR constraint from 0 to 5, and a value for p (i.e., periods for cold standby priming) from 0 to 5. Note that the Y-axis is in log scale. Our heuristics are faster than the simulated annealing approach by more than 2 orders of magnitude. We also plot the number of nodes utilized by each technique per iteration against the number of primaries in the task set. Figure 10 presents the results for 5,000 randomly-generated tasksets generated using Stafford's `Randfixedsum` algorithm [11] for total utilization values ranging from 0.1 to number of primaries and random period ranges from 1 to 10^4. As Figures 9 and 10 show, though the simulated annealing algorithm takes longer to complete, it produces an allocation with fewer nodes on average when compared to RTT. This approach can be used for generating offline static allocations and in other non-time-sensitive contexts. In contrast, our heuristics can be used for run-time admission control and other environments that are time-sensitive.

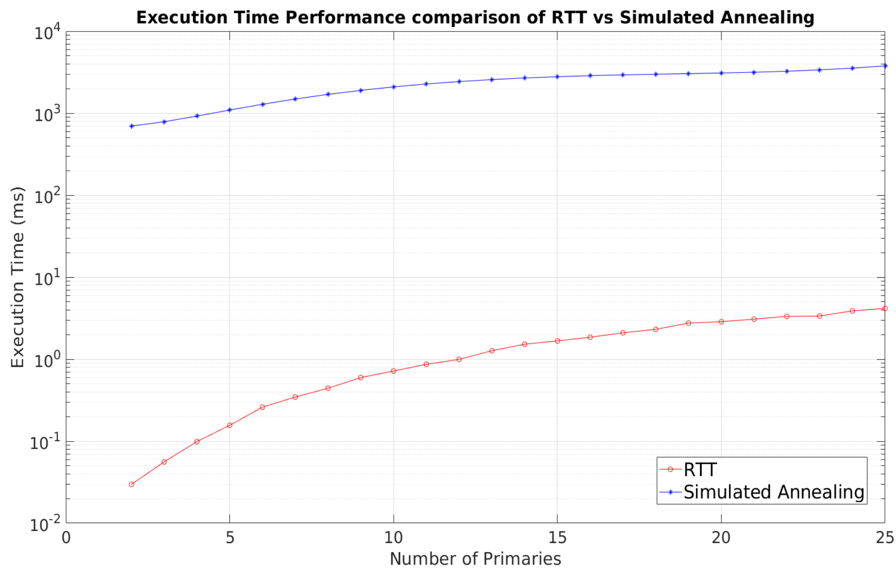


Figure 9 Execution-Time Evaluation.

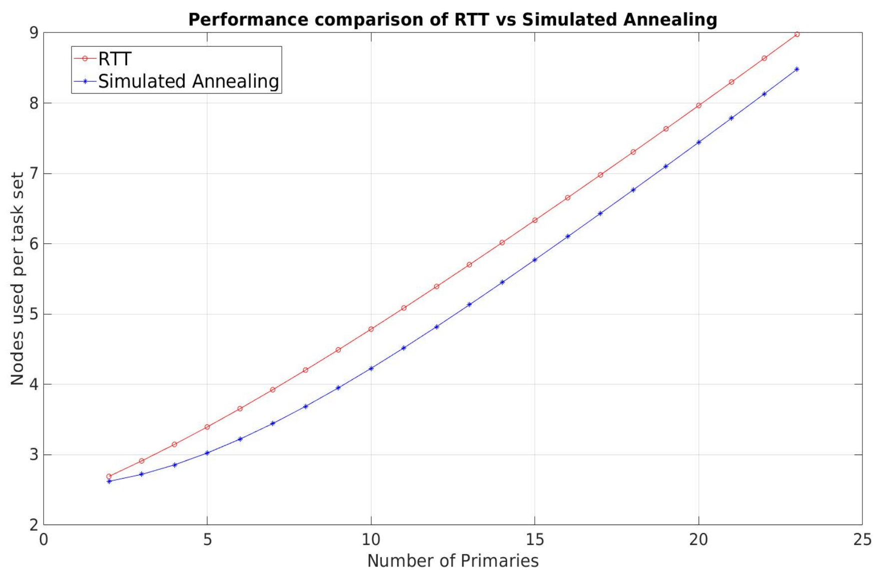


Figure 10 Resource Utilization Evaluation.

8 Concluding Remarks

In this paper, we considered software fault-tolerance techniques for safety-critical real-time systems and derived the bounds on the recovery time of different types of redundant tasks: active replication and primary backups with hot and cold standbys. We also derived conditions to map the recovery time requirements (RTR) of a task to a specific assignment of a redundant-task type. We extended the fault-tolerant task allocation problem to include these RTR constraints, and proposed the TPCDC+R heuristic to satisfy these constraints. Finding a core weakness in TPCDC+R, we then presented two additional heuristics called Recovery-Time Tiered (RTT) and Tiered Recovery-Time Constraint Increasing (TRTI) which

prioritize the RTR constraints in the task allocation sequence. These two heuristics on average produce allocations with fewer nodes than the TPCDC+R heuristic because they yield more assignments of resource-efficient cold standbys. Overall, the RTT heuristic, which tiers tasks based on their RTR values to prioritize the allocation of tasks with strict RTR requirements first, performs the best. Finally, we used the simulated annealing method to solve the fault-tolerant task allocation optimization problem and showed that it produces allocations utilizing fewer computing resources than the proposed heuristics, at the cost of substantial run-time.

References

1. IEEE802.1cb-frame replication and elimination for reliability, howpublished = `http://www.ieee802.org/1/pages/802.1cb.html`, note = Accessed: 2018-01-12.
2. KapDae Ahn, Jong Kim, and SungJe Hong. Fault-tolerant real-time scheduling using passive replicas. In *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 98–103, Dec 1997. `doi:10.1109/PRFTS.1997.640132`.
3. A. A. Bertossi, L. V. Mancini, and A. Menapace. Scheduling hard-real-time tasks with backup phasing delay. In *2006 Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 107–118, Oct 2006. `doi:10.1109/DS-RT.2006.33`.
4. A. Bhat, S. Samii, and R. Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 87–98, April 2017. `doi:10.1109/RTAS.2017.33`.
5. F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao. Implementing fail-silent nodes for distributed systems. *IEEE Transactions on Computers*, 45(11):1226–1238, Nov 1996. `doi:10.1109/12.544479`.
6. Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed Systems (2Nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. URL: `http://dl.acm.org/citation.cfm?id=302430.302438`.
7. A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 29–33, Jun 1996. `doi:10.1109/EMWRTS.1996.557785`.
8. J. J. Chen, C. Y. Yang, T. W. Kuo, and S. Y. Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 249–258, April 2007. `doi:10.1109/RTAS.2007.30`.
9. Jean claude Laprie and Brian Randell. Fundamental concepts of computer systems dependability. In *In Proceedings of the 3rd IEEE Information Survivability, Boston, Massachusetts, USA, October 2000*, pages 24–26, 2001.
10. Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, Apr 2007. `doi:10.1007/s11241-007-9012-7`.
11. Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.
12. Krzysztof Fleszar and Khalil S. Hindi. New heuristics for one-dimensional bin-packing. *Comput. Oper. Res.*, 29(7):821–839, 2002. `doi:10.1016/S0305-0548(00)00082-4`.

13 S. Gopalakrishnan and M. Caccamo. Task partitioning with replication upon heterogeneous multiprocessor systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 199–207, April 2006. `doi:10.1109/RTAS.2006.43`.

14 Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *Reliable Software Technologies — Ada-Europe '96*, pages 38–57, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

15 P. Guo and Z. Xue. Improved task partition based fault-tolerant rate-monotonic scheduling algorithm. In *2016 International Conference on Security of Smart Cities, Industrial Control System and Communications (SSIC)*, pages 1–5, July 2016. `doi:10.1109/SSIC.2016.7571812`.

16 K Hasimoto, Tatsuhiro Tsuchiya, and T Kikuno. Effective scheduling of duplicated tasks for fault tolerance in multiprocessor systems. *IEICE TRANSACTIONS on Information and Systems*, E85-D:525–534, 03 2002.

17 J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. *A Program Structure for Error Detection and Recovery*, pages 53–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985. `doi:10.1007/978-3-642-82470-8_7`.

18 David Johnson. Near-optimal bin packing algorithms. *Ph.D. Dissertation, MIT, MA*, 08 2010.

19 J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. Safer: System-level architecture for failure evasion in real-time applications. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 227–236, Dec 2012. `doi:10.1109/RTSS.2012.74`.

20 J. Kim, K. Lakshmanan, and R. Rajkumar. R-batch: Task partitioning for fault-tolerant multiprocessor real-time systems. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1872–1879, June 2010. `doi:10.1109/CIT.2010.321`.

21 S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

22 Kay Klobedanz, Jan Jatzkowski, Achim Rettberg, and Wolfgang Mueller. Fault-tolerant deployment of real-time software in autosar ecu networks. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro C. Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification*, pages 238–249, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

23 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. `doi:10.1145/321738.321743`.

24 N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *jcp*, 21:1087–1092, jun 1953. `doi:10.1063/1.1699114`.

25 Dong-Ik Oh and T.P. Bakker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15(2):183–192, Sep 1998. `doi:10.1023/A:1008098013753`.

26 Yingfeng Oh and Sang H. Son. Enhancing fault-tolerance in rate-monotonic scheduling. *Real-Time Systems*, 7(3):315–329, Nov 1994. `doi:10.1007/BF01088524`.

27 C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Fault-tolerant distributed deployment of embedded control software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(5):906–919, May 2008. `doi:10.1109/TCAD.2008.917971`.

28 Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39(1):205–235, Aug 2008. `doi:10.1007/s11241-007-9040-3`.

29 R.L. Rao and S.S. Iyengar. Bin-packing by simulated annealing. *Computers and Mathematics with Applications*, 27(5):71–82, 1994. doi:10.1016/0898-1221(94)90077-9.

30 Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, Mar 2004. doi:10.1023/B:TIME.0000016129.97430.c6.

31 Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems., .

32 C. Schonfeld. Redundancy approaches in spacecraft computers. In *28th Israel Annual Conference on Aviation and Astronautics*, pages 148–156, 1986.

33 L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990. doi:10.1109/12.57058.

34 D. Thiele, P. Axer, and R. Ernst. Improving formal timing analysis of switched ethernet by exploiting fifo scheduling. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015. doi:10.1145/2744769.2744854.

35 Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William “Red” Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. *Autonomous Driving in Urban Environments: Boss and the Urban Challenge*, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-03991-1_1.

36 A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292(8), September 1993. URL: http://digital-library.theiet.org/content/journals/10.1049/sej.1993.0034.

37 Thomas Wolf and Alfred Strohmeier. Fault tolerance by transparent replication for distributed ada 95. In Michael González Harbour and Juan A. de la Puente, editors, *Reliable Software Technologies — Ada-Europe’ 99*, pages 412–424, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.