A New Fault-Tolerant Technique for Improving Schedulability in Multiprocessor Real-time Systems

R. Al-Omari Arun K. Somani G. Manimaran
Dependable Computing & Networking Laboratory
Dept. of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011, USA
romari@iastate.edu arun@iastate.edu gmani@iastate.edu

Abstract

In real-time systems, tasks have deadlines to be met despite the presence of faults. Primary-Backup (PB) scheme is one of the most common schemes that has been employed for fault-tolerant scheduling of real-time tasks, wherein each task has two versions and the versions are scheduled on two different processors with time exclusion. There have been techniques proposed for improving schedulability of the PB-based scheduling. One of the more popular ones include Backup-Backup (BB) overloading, wherein two or more backups can share/overlap in time on a processor. In this paper, we propose a new schedulability enhancing technique, called primary-backup (PB) overloading, in which the primary of a task can share/overlap in time with the backup of another task on a processor. The intuition is that, for both primary and backup of a task, the PB-overloading can assign an earlier start time than that of the BB-overloading, thereby increasing the schedulability. We conduct schedulability and reliability analysis of PB- and BB-overloading techniques through simulation and analytical studies. Our studies show that PB-overloading offers better schedulability (25% increase in the guarantee ratio) than that of BB-overloading, and offers reliability comparable to that of BB-overloading. The proposed PB-overloading is a general technique that can be employed in any static or dynamic fault-tolerant scheduling algorithm.

1 Introduction

Due to the critical nature of tasks in a real-time system, it is essential that every task admitted in the system completes its execution even in the presence of faults. Therefore, fault-tolerance is an important requirement in such systems. Scheduling multiple versions of tasks on different processors can provide fault-tolerance [1]-[8]. One of the approaches that is used for fault-tolerant scheduling of real-time tasks is the Primary-Backup (PB) model, in which two versions of a task are scheduled on two different processors and an acceptance test is used to check the correctness of the execution result [2, 3, 7]. The backup version is executed only if the output of the primary fails the acceptance test, otherwise it is deallocated from the schedule.

In the context of scheduling, the term "overloading" refers to scheduling of more than one task (version) on the same/overlapping time interval on a processor. Fault-tolerant scheduling algorithms [2, 3] have employed overloading technique as a means to conserve system resources in order to improve the schedulability of the system. For PB-based fault-tolerant scheduling, a technique called *backup overloading*, we call it Backup-Backup (BB) overloading was proposed in [3] and was improved in [2]. In BB-overloading, two or more backups can be overloaded. In this paper, we propose a new technique, called primary-backup (PB) overloading, wherein the primary of a task can be overloaded with the backup of another task. The objective of PB-overloading is to improve the schedulability with minimal reduction in the reliability of the system.

The rest of the paper is organized as follows. In Section 2, we define the task, scheduler, and fault models. In Section 3, we discuss backup overloading and related work. In Section 4, we propose the PB-overloading technique. In Section 5, we present the analytical and the simulation studies of the PB- and BB-overloading techniques. In Section 6, we make some concluding remarks.

2 Task, Scheduler, and Fault Models

Task model: Tasks are aperiodic, i.e., task characteristics are not known a priori. Every task T_i has the following attributes: arrival time (a_i), ready time (r_i), worst case computation time (c_i), and deadline (d_i). Each task T_i has two versions, namely primary version (Pr_i) and backup version (Bk_i). Both versions have identical attributes. Tasks are non-preemptable.

Scheduler model: In our dynamic multiprocessor scheduling, all the tasks arrive at a central processor, called the scheduler, from where they are distributed to other processors in the system for execution. These processors are identical and connected through a shared medium. The communication between the scheduler and the processors is through *dispatch queues* wherein the schedule is stored. Each processor has its own dispatch queue. The scheduler will be running in parallel with the processors, scheduling the newly arriving tasks and periodically updating the dispatch queues.

Terminology:

Definition 1: $st(Pr_i)$ is the scheduled start time and $ft(Pr_i)$ is the scheduled finish time of primary version of task T_i. Similarly, $st(Bk_i)$ and $ft(Bk_i)$ denote the same for the backup version of T_i.

Definition 2: $Proc(Pr_i)$ is the processor onto which the primary version of task T_i is scheduled. Similarly, $Proc(Bk_i)$ denotes the same for the backup version of T_i.

Definition 3: $S(Pr_i)$ is the time interval (slot) in which primary version of a task T_i is scheduled (i.e., $S(Pr_i) = [st(Pr_i), ft(Pr_i)]$). Similarly $S(Bk_i)$ denote the same for the backup version.

Definition 4: A task is said to be feasible in the fault-tolerant schedule if it satisfies the following constraints.

1. The primary and backup versions of a task should satisfy deadline and time exclusion constraints.

$$r_i \leq st(Pr_i) \leq ft(Pr_i) \leq st(Bk_i) \leq ft(Bk_i) \leq d_i.$$

2. The primary and backup versions of a task should be mutually exclusive in space in the schedule (i.e., $Proc(Bk_i) \neq Proc(Pr_i)$). This is necessary to tolerate permanent processor failures.

Fault model: Each processor, except the scheduler[1], may fail due to hardware fault which results in tasks' failures. The faults can be transient or permanent and are independent. Each independent fault results in failing of only one processor. The following assumptions form the fault model.

Assumption 1: The maximum number of processors that are expected to fail at any instant of time in a group of processors is assumed to be one (The concept of forming processors into groups will be explained later). This is a reasonable assumption given that the group size is very small, which is the case in our techniques wherein the maximum group size is three.

Assumption 2: The occurrence of faults in each processor follows a Poisson distribution with parameter λ_i.

Assumption 3: Mean time to failure ($MTTF$) of a group is defined as the expected time for which the group operates before the first failure occurs. Time to second fault ($TTSF$) is the time at which the second fault can occur without affecting the operation of the system [3].

[1]For example, the scheduler can be made fault-tolerant by executing it on more than one processor.

Note that, $MTTF$ is imposed on the system by the occurrence of a fault, whereas $TTSF$ is the ability of the system to react to fault. Smaller the $TTSF$, better the fault-tolerant operation of the system. In a PB-based fault-tolerant scheduling, the minimum required value of $TTSF$ (see Section 5.1.2) is always greater than or equal to $(d_i - r_i)\ \forall\ T_i$. Obviously, $TTSF$ must be much smaller than $MTTF$ for the system to remain operational continuously. To satisfy this condition we assume that the $(d_i - r_i)\ \forall\ T_i$ are much smaller than the typical $MTTF$ value of the system. This is to ensure that the backup of a task will always execute successfully, if its primary fails.

Assumption 4: There exists fault-detection mechanisms such as fail-signal and acceptance test to detect processor and task failures. The scheduler will not schedule tasks to a known faulty processor.

3 Backup overloading and related work

Backup overloading [3] is defined as scheduling backups of multiple primaries onto the same or overlapping time interval on a processor. Figure 1 shows two primaries (Pr_1 and Pr_2) that are scheduled on processors 1 and 3, respectively, and their backups (Bk_1 and Bk_2) are scheduled in an overloading manner on processor 2. The following are the conditions under which backups can be overloaded on a processor:

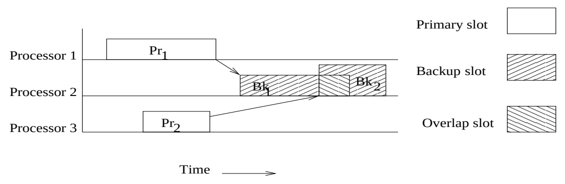

Figure 1. BB-overloading

Condition 1: The primaries of the backups being overloaded must be scheduled onto different processors.

Condition 2: At most one of these primaries is expected to encounter a fault.

Condition 3: At most one version of a task is expected to encounter a fault. In other words, if the primary of a task fails, its backup always succeeds.

Condition 1 is needed to handle permanent faults. Condition 2 is needed to ensure that at most one backup is required to be executed among the overloaded backups. Condition 3 is needed to ensure that at least one version of each task is executed without any fault. Condition 2 can be guaranteed by Assumption 1. Condition 3 is guaranteed by Assumption 3 which states that the $MTTF$ of the system is much greater than $TTSF$ of the fault-tolerant technique.

3.1 Related work and motivation for our work

In this section, we first discuss existing BB-overloading techniques and then discuss the motivation for PB-overloading.

BB-overloading: No-grouping [3]: A scheduling algorithm with BB-overloading was proposed in [3] for fault-tolerant dynamic scheduling of real-time tasks in multiprocessor systems. In this algorithm, a backup has $n-1$ choices for overloading (except the processor onto which its primary is scheduled), where n is the number of processors in the system. Thus, the number of backups that could potentially be overloaded (in a time slot) is $n-1$. Although this algorithm has a potential to offer higher schedulability due to its maximum flexibility in overloading, it can tolerate at most one failure at any point of time. Which is too optimistic and becomes unrealistic for larger n as $MTTF$ becomes smaller.

BB-overloading: Static grouping [2]: This algorithm statically divides the system processors into disjoint logical groups and allows backup overloading to take place only within the group. That is, the processors onto which the backups are overloaded/scheduled belong to the same logical group where their primaries are scheduled. Note that, the number of backups that can be overloaded in a time slot is $k-1$, where k is the number of processors in a logical group. Although this algorithm restricts the flexibility of overloading, it allows at most g faults in the system (where g is the number of logical groups) one in each logical group, thus improving the reliability of the system. In other words, static grouping introduces a trade-off between system utilization (schedulability) and reliability. Here, since the group size is taken to be 3 or 4, the assumption of having only one fault at a time in a group is reasonable. Thus, static grouping improves the probability of satisfying Conditions 2 and 3.

BB-overloading: Dynamic grouping [9]: In an attempt to offer schedulability close to that of no-grouping and reliability close to that of static grouping, we have proposed a method called *dynamic grouping* [9], which dynamically divides the processors into logical groups. The creation and expansion of a group take place when a task is scheduled, and removing and shrinking of a group take place when a task finishes its execution. The number of groups and the size of the groups will vary dynamically in contrast to static grouping where these quantities are fixed. Moreover, in static grouping, a processor can be a member of only one group. Whereas in dynamic grouping, a processor can be a member of more than one group which allows efficient overloading. In [9], we have shown that dynamic grouping offers better schedulability than static grouping and comparable to that of no-grouping, and offers the same reliability that of static grouping (and better than that of no-grouping). Since the number of processors in a group is very small, the dynamic grouping also has better chances of satisfying Conditions 2 and 3.

The motivation for our work is to propose a better technique than BB-overloading to improve the schedulability of the system. We call this technique Primary-Backup overloading. In our work, we quantify the schedulability and reliability gain/loss offered by the proposed technique over the BB-overloading technique.

4 Primary-Backup overloading

In primary-backup overloading (PB-overloading), the primary of a task can be scheduled onto the same or overlapping time interval (i.e., overloaded) with the backup of another task on a processor. This technique has a potential to offer better schedulability than BB-overloading and is based on the following intuition.

> For a primary, PB-overloading can assign an earlier *start time* than that of BB-overloading, because the primary can be overloaded on an already scheduled backup. This, in turn, helps its backup to find an earlier start time, thus resulting in a better chance of meeting the task deadline.

In other words, PB-overloading has a better chance of making a task feasible (refer to Definition 4) in the schedule compared to BB-overloading.

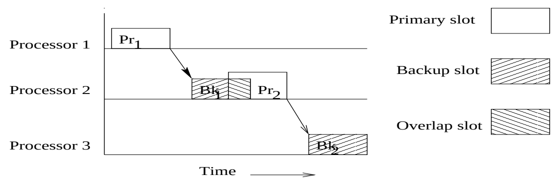

Figure 2. PB-overloading

Figure 2 shows a primary Pr_1 that is scheduled on processor 1 and its backup Bk_1 is scheduled on processor 2. Also it shows another primary Pr_2 that is overloaded with Bk_1 on processor 2 and its backup Bk_2 is scheduled in processor 3. With respect to this example, we state the following conditions that govern the PB-overloading, which are equivalent to the conditions governing BB-overloading.

Condition 1: Pr_1 and Bk_2 should be scheduled onto different processors.

Condition 2: $ft(Bk_2) - st(Pr_1) \leq$ the expected minimum interval between two faults.

Condition 3: $Proc(Pr_1)$, $Proc(Bk_1) = Proc(Pr_2)$, and $Proc(Bk_2)$ should be in the same group.

Condition 1 is needed to handle permanent faults. Conditions 2 and 3 are needed to ensure that at most one version among these tasks is likely to encounter a fault. It can be noted that grouping mechanisms such as no-grouping, static grouping, and dynamic grouping are applicable to PB-overloading as well. Also, it is to be noted that, both BB- and PB-overloading techniques can co-exist in a single scheduling algorithm. However, we believe that this will significantly increase the cost (complexity) of scheduling, which may not offer a proportionate increase in schedulability. Therefore, in this paper, we do not explore the co-existence of BB- and PB-overloading techniques.

4.1 Group dynamics with PB-overloading

In the PB-overloading, a backup can be in one of two states. In the first state (green), a primary is allowed to overload with the green backup. In the second state (red), a primary is not allowed to overload with the red backup. The states are controlled by the following rules.

Rule 1: If a primary is scheduled without overloading with another backup, its backup has a green state.

Rule 2: If a primary is scheduled with overloading on another backup, its backup has a red state.

Rule 3: A red backup becomes green if the backup that was overloaded with its primary is de-allocated (i.e., the primary for that backup has executed successfully). Otherwise, the backup stays red.

Rule 4: A green backup becomes red if its primary has failed to execute successfully. Otherwise, it stays green.

These rules ensure that tasks are scheduled onto fault-free groups (i.e., groups that did not encounter a fault previously), which will increase the probability of satisfying the fault model. Also, these rules ensure that the "PB-overloading chain" will not contain more than two tasks at the same time.

Example PB-overloading dynamics: Figure 3 shows an example that illustrates the working of PB-overloading. Figure 3a shows that the primary of task T_1 is scheduled on processor 1, and its backup (green state) is scheduled on processor 2. Then, these two processors will form a logical group (group 1) that will stay until one of these versions executes successfully. Figure 3b shows the same situation as in Figure 3a, but this time the primary of T_2 is scheduled to overload with Bk_1 on processor 2, and its backup (red state) is scheduled on processor 3. This results in expanding the group to have three processors. Figure 3c shows the situation when Pr_1 has executed successfully. Therefore, Bk_1 is de-allocated which results in shrinking group 1 to have two processors (processors 2 and 3), and Bk_2 becomes in green state. On the other hand, Figure 3d shows the situation after Pr_1 has failed to execute successfully. Therefore, Pr_2 is de-allocated and Bk_1 becomes in red state. The group will stay until Bk_1 finishes its execution which results in shrinking group 1 to have two processors (processors 2 and 3) which will stay until Bk_2 also finishes its execution.

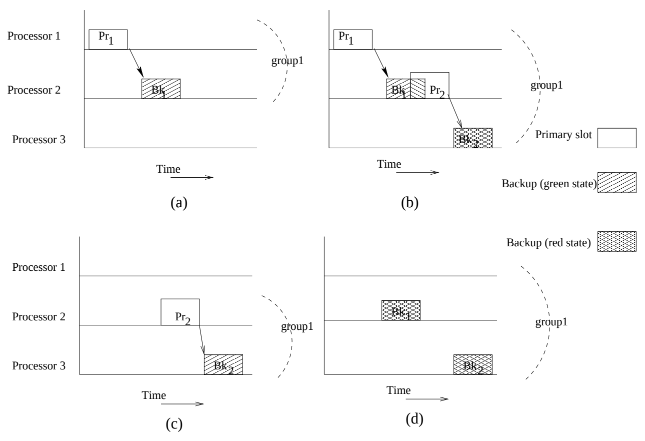

Figure 3. Dynamics of PB-overloading

5 Performance studies

We have conducted schedulability and reliability analysis of the BB- and PB-overloading techniques through extensive analytical and simulation studies.

5.1 Analytical studies

In this section, we study analytically the effectiveness of the proposed overloading technique in the system schedulability and reliability.

5.1.1 Schedulability analysis

The capability of a technique to schedule tasks can be measured in terms of the probability of a task being rejected. In this section, We show that the probability of rejection in BB-overloading is higher than in PB-overloading. The analysis follows the methodology used in [2, 3]. We use the following assumptions:

1. All tasks have unit worst case computation time, i.e., $c_i = 1$. **2.** Backup slots are preallocated in the schedule. **3.** FIFO scheduling strategy is used. **4.** Task deadlines follow (discrete) uniform distribution in the interval $[W_{min}, W_{max}]$ relative to their ready times. We call this, *deadline window*. If $P_{win}(w)$ is the probability that an arriving task has a relative deadline w, then $P_{win}(w) = \frac{1}{W_{max} - W_{min} + 1}$, where $W_{min} \leq w \leq W_{max}$. **5.** Task arrivals follow (discrete) uniform distribution with mean $A_{av} = \frac{A_{max}}{2}$ in the interval $[0, A_{max}]$. If $P_{ar}(k)$ is the probability of k tasks arriving at a given time, then $P_{ar}(k) = \frac{1}{A_{max}+1}$, where $0 \leq k \leq A_{max}$.

Pre-allocation strategies: The pre-allocation strategies presented here are for a system with three processors (i.e., one group). When the system size is greater than three, one of the grouping techniques (static or dynamic) can be used to divide the system into groups and the pre-allocation strategy is applied on a group basis.

A simple pre-allocation policy for BB-overloading (as given in [3]) is to reserve a slot for backups every n (n is the number of processors, which is 3) time slots on each processor. Backup slots on the three processors can be staggered (Figure 4). In this policy, if a backup slot is pre-allocated at time t on processor P_i, then any task scheduled to run at time $t-1$ on P_j, $j \neq i$, can use this slot for its backup. Because, the task scheduled to run on P_i at time $t-1$ cannot have its backup slot on the same processor (space exclusion), then this task can use the backup slot at time $t+1$, which is on $P_{(i+1) mod\ 3}$. In other words, for a task T_i, Bk_i is scheduled immediately after Pr_i with probability 0.5 and is scheduled two slots later than Pr_i with probability 0.5.

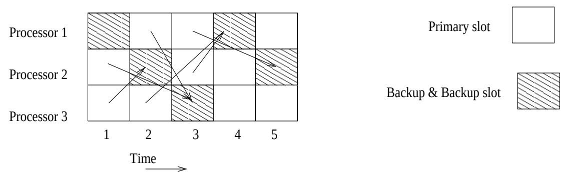

Figure 4. Pre-allocation strategy for BB-overloading

To define a pre-allocation strategy for PB-overloading we need to identify three different types of time (0, 1 ,and 2), wherein any time t has a type i if $(t-1)\ mod\ 3 = i$. In our pre-allocation strategy, at any time t, the number of primaries that can be scheduled to start at that time is s_0 if t is of type 0, s_1 if t is of type 1, and s_2 if t is of type 2. Figure 5 shows that $s_0 = 2$ (e.g. $t = 1$), $s_1 = 3$ (e.g. $t = 2$), and $s_2 = 1$ (e.g. $t = 3$). In our pre-allocation strategy if a primary Pr_i is scheduled on processor P_i at time t its backup is scheduled at time $t+1$ in processor $P_{(i+1)\ mod\ 3}$. In other words, for a task T_i, Bk_i is scheduled immediately after Pr_i with probability 1.

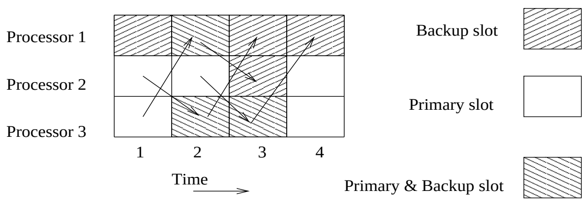

Figure 5. Pre-allocation strategy for PB-overloading

Analysis: Using FIFO scheduling is equivalent to maintaining a task queue, Q, to which arriving tasks are appended. Given that the number of tasks that can be scheduled on each time unit is known, then the position of a task in the Q indicates its scheduled start time.

In the pre-allocation strategy for the BB-overloading, two tasks can be scheduled on each time t (one slot is reserved for backups). If at the beginning of time slot t, a task T_i is the q^{th} task in Q, then T_i is scheduled to execute at time slot $t + g_q^{BB}$. Where g_q^{BB} is the time at which a task, whose position in the Q is q ($q = 1, 2, \ldots 2W_{max}$), will be executed and is defined as

$$g_q^{BB} = \lfloor \frac{q}{2} \rfloor. \qquad (1)$$

In the pre-allocation strategy for the PB-overloading s_0, s_1, or s_2 tasks can be scheduled on a given time slot t depending on whether t is of type 0, 1, or 2, respectively. In this technique, the time g_q^{PB} is defined as

$$g_q^{PB} = (i+j+l)\ if\ \left[\sum_{c=1}^{i} s_0 + \sum_{c=1}^{j} s_1 + \sum_{c=1}^{l} s_2\right] \leq q-1, \quad and\ |i-j| \leq 1, |j-l| \leq 1, |l-i| \leq 1. \qquad (2)$$

where $i \geq j \geq l$ if t is of type 0, $j \geq l \geq i$ if t is of type 1, and $l \geq i \geq j$ if t is of type 2.

When a task T_i arrives at time t, its schedulability depends on the length of Q and on the relative deadline w_i of the task. In BB-overloading, if T_i is appended at position q of Q and $w_i \geq g_q^{BB}$, then the primary task, Pr_i, is guaranteed to execute before time $t + w_i$. Otherwise, the task is not schedulable since it will miss its deadline. Moreover, if $w_i \geq g_q^{BB} + 2$, then Bk_i is also guaranteed to execute before time $t + w_i$. In PB-overloading, if T_i is appended at position q of Q and $w_i \geq g_q^{PB}$, then the primary task, Pr_i, is guaranteed to execute before time $t + w_i$. Otherwise, the task is not schedulable since it will miss its deadline. Moreover, if $w_i \geq g_q^{BB} + 1$, then Bk_i is also guaranteed to execute before time $t + w_i$.

The dynamics of the above schemes can be approximately modeled using discreet time Markov process. For simplicity of presentation, a system without deadline is modeled first i.e., a system in which no tasks are rejected. Such a system may be modeled by a linear discreet time Markov chain in which each state represents the number of tasks in Q and each transition represents the change in the length of the Q in one unit of time. The probabilities of different transitions may be calculated from the rate of task arrival, and the average number of tasks executed in a unit of time. The average number of tasks executed at a unit of time is 2 for the BB-overloading and is also 2 ($= \frac{s_0+s_1+s_2}{3}$) for the PB-overloading. If S_u represents the state in which Q contains u tasks and $u \geq 2$, then the probability of a transition from S_u to S_{u-2+k} is $P_{ar}(k)$. This is because, at any time t, k tasks can arrive and 2 tasks get executed for the two schemes. If $u < 2$, then only u tasks are executed, then there is a state transition from S_u to S_k with probability $P_{ar}(k)$. For example, Figure 6 shows the transitions out of state S_u ($u > 2$) in the Markov chain assuming that $A_{max} = 6$.

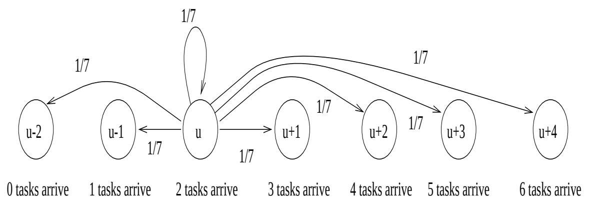

Figure 6. Transitions out of state S_u for a linear Markov chain

Now we consider the case of tasks with deadlines. When the k arriving tasks have finite deadlines, some of these tasks may be rejected. Let $P_{q,k}$ be the probability that one of the k tasks is rejected when the queue size is q. For the BB-overloading, the value of $P^{BB}_{q,k}$ is the probability that the relative deadline of the task is smaller than $g^{BB}_b + \sigma$, where $b = q + k/2$ (average case) and σ is the extra time needed to schedule the backup and is equal to 1 or 2 with a probability of 0.5. For the PB-overloading, the value of $P^{PB}_{q,k}$ is the probability that the relative deadline of the task is smaller than $g^{PB}_b + 1$, where $b = q + k/2$ (average case) and the extra one time unit is needed to schedule the backup. Then, $P^*_{q,k} = P_{win}(w < \bar{t})$ which is defined as

$$P^*_{q,k} = \begin{cases} 0 & \bar{t} < W_{min} \\ 1 - \sum_{w=\bar{t}}^{W_{max}} P_{win}(w) & W_{min} \le \bar{t} \le W_{max} \\ 1 & \bar{t} > W_{max} \end{cases} \tag{3}$$

where $* = BB$, and $\bar{t}$=$g^{BB}_b + \sigma$ for the BB-overloading scheme, or $* = PB$ and $\bar{t}$=$g^{PB}_b + 1$ for the PB-overloading scheme. Hence, when the queue size is q, the probability, $P^*_{rej}(r,k,q)$, that r out of k tasks are rejected is

$$P^*_{rej}(r,k,q) = C^k_r \left(P^*_{q,k}\right)^r \left(1 - P^*_{q,k}\right)^{k-r} \tag{4}$$

where C^k_r is the number of possible ways to select r out of k elements.

Up until now, we have not considered backup de-allocation in the model. Backup de-allocation means that if at time t no fault has occurred, then the backups pre-allocated at time slot $t+1$ may be used to schedule a new task. In other words, if k tasks arrive during slot t, and $k > 0$, then one of these tasks can be scheduled in the de-allocated backup slot, and the remaining $k-1$ tasks can be treated as above. The effect of backup de-allocation may be analyzed by changing k by $k-1$ in the previous analysis. More specifically in the equations for g^*_b, where $b = q + (k-1)/2$.

Results: From the analysis, we notice that the performance of a technique depends on the probability of rejecting a task ($P^*_{q,k}$) for that technique. Figure 7 shows the probability of rejecting a task as the number of task arrival (k) varies for both faulty and fault-free cases. The figure shows that

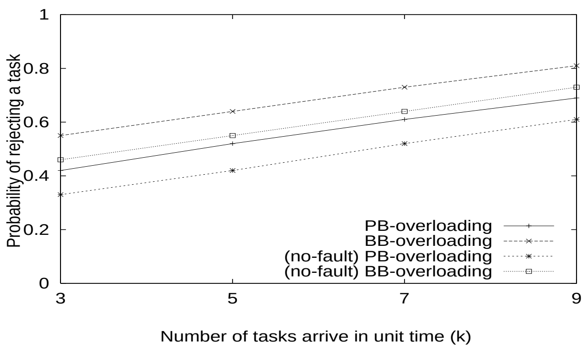

Figure 7. Effect of task load on $P^*_{q,k}$ ($W_{max} = 5$,$W_{min} = 3$,$A_{max} = 9$)

PB-overloading technique has a lesser probability to reject a task for all values of k for both faulty and fault-free cases. The figure also shows that the probability of rejecting a task increases as task load increases for both techniques.

5.1.2 Reliability analysis

The capability of a fault-tolerant technique is assessed based on the number of and the frequency of faults that it can tolerate. The resilience of the system can be measured in terms of the time it takes for the system to be able to tolerate the second fault after the first fault using a given fault-tolerance technique. This latency is called the time to second fault ($TTSF$) [3]. Higher the $TTSF$, poor the performance of the fault-tolerant technique.

In the previous section, we have shown that PB-overloading technique offers better schedulability than BB-overloading. Here, we show that $TTSF$ offered by the PB-overloading is bounded by twice that of the BB-overloading. Note that, a typical value for $TTSF$ is much smaller than $MTTF$ (Assumption 3 of the fault model) and hence both the overloading techniques offer a similar reliability from the practical point of view. Theorems 1 and 2 below quantify the $TTSF$ offered by BB-overloading and PB-overloading respectively.

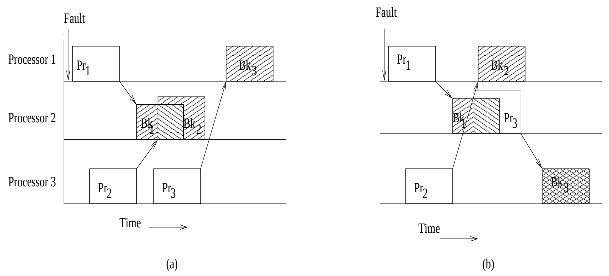

Figure 8. Tolerating a second fault

Theorem 1: In BB-overloading if a permanent fault occurs at time t in processor P_i, the technique will continue to operate if another fault does not occur in the group before time $\overline{t}$, where

$$\overline{t} > \max\{max_j\{ft(Bk_j) : Proc(Pr_j) = P_i\}, \\ max_j\{ft(Pr_j) : Proc(Bk_j) = P_i\}\} \quad (5)$$

Proof: If a permanent fault occurs at time t in P_i, any task arriving later than t will be scheduled (both primary and backup) on the $n-1$ non-faulty processors (Assumption 4 of the fault model). Thus, such tasks are guaranteed to complete even if a second fault occurs. If a task T_j is already scheduled when the first fault occurs, then the following two cases arise:

case 1: $Proc(Pr_j) = P_i$ or $Proc(Bk_j) = P_i$: In this case, the restriction on $\overline{t}$ guarantees that Bk_j or Pr_j will execute successfully before the second fault occurs. For example, in Figure 8a, primary Pr_1 is scheduled on P_1 and its backup Bk_1 on P_2. If a fault occurs on P_1 before Pr_1 executes, then a fault can be tolerated on P_2 only after Bk_1 completes. If a second fault occurs on P_2 before $ft(Bk_1)$, both copies of T_1 would be faulty. Using a similar logic, a second fault can be tolerated on P_3 only after Pr_3 completes. The maximum of all such combinations gives the minimum time at which the second fault can occur.

case 2: $Proc(Pr_j) \neq P_i$ and $Proc(Bk_j) \neq P_i$: In this case, T_j is guaranteed to complete even if a second fault occurs unless Bk_j overlaps with a backup Bk_k whose primary Pr_k is scheduled on P_i (for example, in Figure 8a, if $i = 1$, $j = 2$, and $k = 1$). Due to the first fault, Bk_k is activated and hence Bk_j cannot be used (since it overlaps with Bk_k). Therefore, a second fault cannot be tolerated on $Proc(Pr_j)$ (P_3 in Figure 8a) until Pr_j has executed. However, this case is covered by the first part of the restriction on $\overline{t}$ (in Equation 5), according to which the second fault cannot be tolerated before Bk_k has executed. Since Bk_j and Bk_k overlap, Pr_j is scheduled earlier than $ft(Bk_j)$ in the system. This means that the second fault can be tolerated only after Pr_j finishes its execution successfully.

Theorem 2: In PB-overloading if a permanent fault occurs at time t in processor P_i, the technique will continue to operate if another fault does not occur in the group before time $\overline{t}$, where

$$\overline{t} > \max\{max_j\{ft(Bk_j) : Proc(Pr_j) = P_i\}, \\ max_j\{ft(Pr_j) : Proc(Bk_j) = P_i\}, \\ max_j\{ft(Bk_k) : Proc(Pr_k) = Proc(Bk_j), \\ S(Pr_k) \cap S(Bk_j) \neq \emptyset,\ and\ Proc(Pr_j) = P_i\}\} \quad (6)$$

Proof: The first two parts in Equation 6 have the same proof as the BB-overloading. For the third part consider the following case.

case 3: $Proc(Pr_j) \neq P_i$ and $Proc(Bk_j) \neq P_i$: In this case, T_j is guaranteed to complete even if a second fault occurs unless Pr_j overlaps with a backup Bk_k whose primary Pr_k is scheduled on P_i (for example, in Figure 8b, if $i = 1$, $j = 3$, and $k = 1$). Due to the first fault, Bk_k is activated and hence Pr_j cannot be used (since it overlaps with Bk_k). Therefore, a second fault cannot be tolerated on $Proc(Bk_j)$ (P_3 in Figure 8b) until Bk_j has executed. This case is covered by the third part of the restriction on $\overline{t}$ (in Equation 6), according to which a second fault can occur only after all backups, whose primaries were overlapped with backups of primaries on the faulty processor, have executed. The maximum of all such combinations gives the minimum time at which the second fault can occur.

$TTSF$ can be calculated as $\overline{t} - t$. In the worst case, $TTSF_{PB} \leq 2 \times TTSF_{BB}$. For example in Figure 8a, if the fault was transient and it affected Pr_1 only, then the BB-overloading technique will continue to operate if another fault does not occur before $ft(Bk_1)$. In the worst case this interval is $[r_1, d_1]$. On the other hand in Figure 8b, the PB-overloading technique will continue to operate if another fault does not occur before $ft(Bk_3)$. In the worst case, this interval is $[r_1, d_3]$ which is approximately $2 \times [r_1, d_1]$.

5.2 Simulation studies

For our simulation studies, we have used Spring scheduling [10], a well known dynamic scheduling algorithm for scheduling real-time tasks, as the base algorithm to incorporate the BB- and PB-overloading techniques. The guarantee ratio (GR) is used as the performance metric for the capability of a technique to schedule tasks, which is defined as: $GR = \frac{the\ number\ of\ tasks\ guaranteed}{the\ number\ of\ tasks\ arrived}$. The $TTSF$ is used as the performance metric for measuring the capability of a technique to tolerate faults. For each point in the performance curves, a total of 20,000 tasks arrived in the system.The number of task have been chosen to have approximately 95% confidence interval within ± 0.005 around the results. The tasks for the simulation are generated as follows:

1. The worst case computation time of a task (primary version) is chosen uniformly between $20\ sec.$ and $40\ sec.$ **2.** The deadline of a task T_i (primary version) is uniformly chosen between $r_i + 2 * c_i$ and $r_i + 5 * c_i$. **3.** The inter-arrival time of tasks follows exponential distribution with mean θ. **4.** The inter-arrival time of faults follows exponential distribution with mean λ, with a minimum value of $400\ sec.$. This value is chosen to be greater than both $TTSF_{PB}$ and $TTSF_{BB}$ to satisfy Assumption 3. **5.** The task load L is defined as the expected number of task arrivals per mean service time and its value is approximately equal to $\frac{1}{\theta}C$, where C is the mean computation time. **6.** The number of processors in the system is chosen to be five.

Results: The task load (L) is varied in Figure 9. As expected, increasing L decreases the guarantee ratio for all the techniques and the difference in performance between the techniques is maximum when the task load is greater than 0.5. This is because, for low loads, the task load is less than the system capacity and hence the techniques tend to behave similarly (i.e., there are no/less overloading take place). The figure also shows that the guarantee ratio offered by PB-overloading is better than BB-overloading and no-overloading for all task loads. The reason is that the PB-overloading can assign a start time for a primary earlier than that of BB-overloading because the primary can be overloaded on an already scheduled backup, which, in turn, helps its backup to find an earlier start time, thus resulting in a better chance of meeting the deadline.

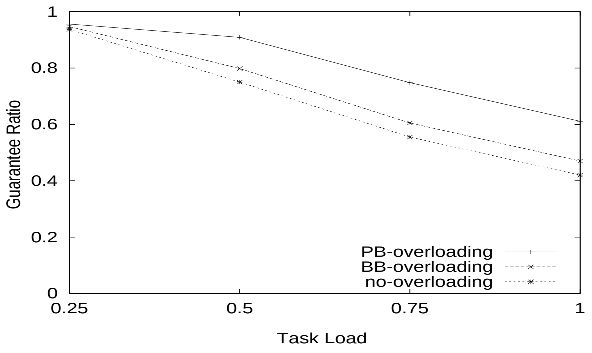

Figure 9. The effect of task load on the guarantee ratio

Figure 10 shows the effect of varying task load on the $TTSF$ for the overloading techniques. The figure shows that the difference in the $TTSF$ between the PB- and BB-overloading techniques is significant at high load ($L = 1$) compared to at low load ($L = 0.25$). This is because, for low loads ($L = 0.25$), most of the tasks are schedulable even without overloading, thus making the techniques behave similarly. The figure also shows that $TTSF_{PB}$ increases more rapidly than $TTSF_{BB}$ as task load increases. This is because, $TTSF_{PB}$ is approximately twice that is offered by the BB-overloading technique and also due to better chances of overloading with increasing load. Whereas, the value of $TTSF_{BB}$ is approximately equal to the value that is offered by the no-overloading. The slight increase in $TTSF_{BB}$ is partly due to the increase in the number of tasks that are affected by the faults as the task load increases.

6 Conclusions

In this paper, we have proposed a new technique, called primary-backup (PB) overloading, for fault-tolerant scheduling of real-time tasks in multiprocessor systems. The proposed PB-overloading technique can work with grouping schemes - such as no-grouping, static grouping, and dynamic grouping - and can easily be incorporated into any scheduling algorithm. We have shown through both analytical and simulation studies that PB-overloading offers better schedulability (25% gain) than BB-overloading for all the cases studied. We have also shown that the $TTSF$ (a reliability metric) of PB-overloading is upper bounded by twice that of BB-overloading, which is a much smaller value than the $MTTF$ of the system. Hence, PB-overloading is an effective technique than BB-overloading for many practical reliability requirement.

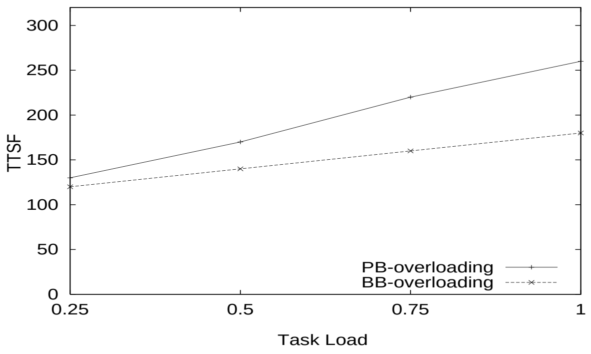

Figure 10. The effect of task load on the guarantee ratio

References

[1] S. Tridandapani, A. K. Somani, and U. Reddy, "Low overhead multiprocessor allocation strategies exploiting system spare capacity for fault detection and location," *IEEE Trans. on Computers*, vol. 44, no. 7, pp. 865-877, July 1995.

[2] G. Manimaran and C. Siva Ram Murthy, " A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1137-1152, Nov. 1998.

[3] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 3, pp. 272-284, Mar. 1997.

[4] J.H. Purtilo and Pankaj Jalote, "An environment for developing fault-tolerant software," *IEEE Trans. on Software Eng.*, vol.17, no.2, pp.153-159, Feb. 1995.

[5] L.V. Mancini, "Modular redundancy in a message passing system," *IEEE Trans. on Software Eng.*, vol.12, no.1, pp.79-86, Jan. 1986.

[6] Y. Oh and S. Son, "Multiprocessor support for real-time fault-tolerant scheduling," In Proc. *IEEE Workshop on Architectural Aspects of Real-time Systems*, Dec. 1991.

[7] K. Kim and J. Yoon, "Approaches to implementation of reparable distributed recovery block scheme," In Proc. *IEEE Fault-tolerant Computing Symposium*, pp.50-55, 1988.

[8] C.M. Krishna and K.G. Shin, "On scheduling tasks with quick recovery from failure," *IEEE Trans. on Computers*, vol.35, no.5, pp.448-455, May 1986.

[9] R. Al-Omari, G. Manimaran, and A. K. Somani,"An Efficient backup-overloading for fault-tolerant scheduling of real-time tasks," in *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 1291-1295, 2000.

[10] J.A. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time operating systems," *ACM SIGOPS, Operating Systems Review*, vol. 23, no. 2, pp. 77-83, Jan. 1995.