R-BATCH: Task Partitioning for Fault-tolerant Multiprocessor Real-Time Systems

Junsung Kim, Karthik Lakshmanan and Ragunathan (Raj) Rajkumar
Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA
{*junsungk, klakshma, raj*}*@ece.cmu.edu*

***Abstract*—Many emerging embedded real-time applications such as SCADA (Supervisory Control and Data Acquisition), autonomous vehicles and advanced avionics, require a high degree of dependability. Dealing with tasks having both hard real-time requirements and high reliability constraints is a key challenge faced in such systems. This paper addresses the problem of guaranteeing reliability requirements with bounded recovery times on *fail-stop* processors in fault-tolerant multiprocessor real-time systems. We classify tasks based on their recovery-time requirements into (i) *Hard Recovery*, (ii) *Soft Recovery*, and (iii) *Best-Effort Recovery* tasks. Then, the notion of a *Hot Standby* for Hard Recovery tasks along with a *Cold Standby* for Soft Recovery and Best-Effort Recovery tasks is introduced. In order to maximize the benefits of using a *Hot Standby*, replicas should not be co-located on the same processor. For this purpose, we propose a task allocation algorithm for *Hot Standby* replicas called R-BFD (Reliable Best-Fit Decreasing) that uses *37%* fewer number of processors than BFD-P (Best-Fit Decreasing augmented with placement constraints). For tasks with more relaxed recovery-time constraints, however, additional optimization can be applied by using a *Cold Standby* that gets activated only when failures occur. Given a system reliability requirement and hence a maximum number of processor failures to tolerate, the required resource over-provisioning for *Cold Standby* replicas from multiple processors can be consolidated. An algorithm called R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) reduces the required number of processors by up to *45%* compared to R-BFD-based pure *Hot Standby* replication technique.**

***Keywords*-real-time systems; fault tolerance; embedded systems; bin-packing algorithms; task replication;**

I. Introduction

Embedded real-time systems are growing in terms of both scale and complexity. An emphasis on scalability, extensibility, and flexibility has led to complex electrical/electronic multiprocessor architectures. In a variety of applications such as industrial control, avionics, and automotive systems, such complexity can lead to unavoidable failures in both hardware and software. Furthermore, developers/designers may not be able to predict when and where faults can happen. System-level dependability is therefore a key concern in evolving embedded real-time systems. An emerging application of such systems is autonomous driving. For example, the Urban Challenge winning autonomous vehicle, Boss [1], used several embedded processors and ten Intel Core2Duo processors due to high computing power requirements. However, CMOS scaling for performance improvements has decreased the reliability of processors [2]. This could potentially have catastrophic effects if not taken into account, for example, unmanned vehicles can lose driving capability due to processor failures.

Two different notions, *fault-tolerant* and *fail-safe*, are applicable for characterizing the system behavior under failures. A fault-tolerant system requires that a system/user does not recognize a failure occurrence during the operation in terms of functionalities. Several conventional replication methods such as hardware redundancy, software redundancy, and re-execution can be used to build fault-tolerant systems. A fail-safe system requires a different type of fault handling. It allows failures, but must not generate an unsafe system state by overriding a proper procedure when a failure occurs. Developing a fail-safe system requires us to consider specific failure scenarios. As described earlier, enumerating all possible failure scenarios is not an easy task for complex systems, therefore, we focus on achieving the more robust property of fault tolerance.

Systems that interact with the physical world such as autonomous vehicles should adhere to the strict timing constraints imposed by their operating environment. In such real-time systems, tasks are conventionally modeled as a periodic sequence of jobs that are releasing every T units of time, where each job needs to finish within a relative deadline of D time-units from its release. Dealing with unpredictable failures in such systems is not a trivial job. Therefore, instead of assuming a priori failure scenarios, dynamically handling failures with bounded recovery time is desirable for real-time systems. By handling failures within a required timing boundary, the Time-To-Recovery is bounded, and the system need not be stopped. Tasks with the requirement that they complete within D units of time from release, even under the presence of failures (i.e. recover and complete within the original deadline), are denoted as *Hard Recovery Tasks*. Tasks with more relaxed recovery-time requirements are denoted as *Soft Recovery Tasks*. Optional tasks that are not critical for system operation and do not require bounded recovery times are denoted as *Best-Effort Recovery Tasks*.

The recovery-time requirement imposed by Hard Recovery Tasks does not allow much room for re-executing the failed jobs. For such tasks, a practical solution is to use multiple *Hot Standby* replicas that can take over the functionality under the presence of failures. Jobs of these Hot Standby replicas are released in parallel with those of the primary task, and they have the same deadline as the primary. It is not required that the Hot Standby replicas execute whenever the primary executes, however, they have the same relative deadline as the primary. This relaxed synchronicity between Hot Standby and Primary enables a more practical solution.

We assume a *fail-stop* failure model [3], where a working

replica can assume control by detecting the lack of output from the primary. The replica can immediately provide the output since it would also have the output by the original deadline. In order to maximize task reliability, a process and its *Hot Standby* replicas should not be co-located on the same processor. We refer to this as the *placement constraint*. For this purpose, we develop a task allocation algorithm called R-BFD (Reliable Best-Fit Decreasing) that optimizes for allocating tasks with Hot Standby replicas having such placement constraints.

Cold Standby replicas reduce the resource over-provisioning costs further by getting activated only under failure conditions. The Cold Standby replicas use the task state information, which can be stored in shared memory or obtained during subsequent execution, and are used to recover soft recovery or best-effort recovery tasks. The benefit of Cold Standbys is that it leads to lesser consumption of resources under normal conditions. However, the recovery time bounds under Cold Standby will be much larger than those guaranteed by Hot Standby. Using the system reliability requirement and the maximum number of processors that can fail during system operation, we can reduce the resource over-provisioning required for Cold Standby replicas of tasks allocated across different processors. Using this observation, we develop an algorithm called R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) that uses virtual tasks to consolidate and capture the resource requirements for Cold Standby replicas.

The rest of this paper is organized as follows. Section II will describe the related research. Section III will discuss the timing properties of different task replication mechanisms in a multiprocessor environment, and will summarize our approach. The proposed task partitioning algorithm will be described in Section IV, and evaluated in Section V. Finally, we provide our concluding remarks in Section VI.

II. Related Work

Real-time scheduling algorithms for uniprocessors have been studied extensively for guaranteeing timeliness. For example, Liu and Layland proposed a static real-time scheduling algorithm, RMS (Rate Monotonic Scheduling), which prioritizes periodic tasks according to their rates since [4]. Emerging demands on computational capability have driven various multiprocessor scheduling algorithms for supporting similar properties on multiprocessor environments. Multiprocessor scheduling is therefore a well-studied problem in real-time systems literature [5, 6, 7, 8]. Existing solutions are broadly classified into global [9] scheduling with unrestricted task migration, partitioned [10] scheduling with strictly no task migration, and hybrid [11, 12] with restricted task migration. Although each of these approaches has its own benefits, in this work we are primarily interested in the partitioned approach due to the high cost associated with migrating tasks across processors. We allow on-demand activation to deal with failed processors, where we explicitly capture the associated timing cost of restoring the task state on a different processor.

A wealth of literature exists on the topic of fault-tolerant computing [13, 14, 15]. *Active* and *Passive* replication are standard mechanisms to improve system reliability. These replication-based approaches affect on research [16, 17, 18] in real-time domain. The key distinction made in real-time contexts is systems with strict bounds on system recovery time. A successful recovery is one where the system not only resumes its normal operation but does so within a pre-specified recovery time. These recovery time requirements are typically derived from the physical environment in which the system operates. For example, an autonomous vehicle cannot stall significantly in the middle of a highway during system recovery.

Fault-tolerant scheduling in multiprocessor systems has also received attention in [19]. In [20], the FFD (First-Fit Decreasing) is augmented with placement constraints to allocate replicated tasks. Most closely related to our work is [21], where the authors proposed a BFD (Best-Fit Decreasing) with placement constraints as a practical solution, which we will call BFD-P (BFD with Placement constraint). Our work differs from theirs in two ways: (i) we propose a new bin-packing heuristic using a cluster of replicated tasks which performs better than BFD-P, and (ii) we deal with the allocation of *Cold Standby* replicas that are passive entities, which can be potentially consolidated across processors.

III. Design Implications

In this paper, we will find a reliability-enforced allocation while reducing the processor required. In other words, we will tolerate ρ permanent processor failures while minimizing number of processors which can support the given reliability requirement of a target system. Under the assumption that applications are executed periodically, transient failures can be recovered by exploiting the periodic nature of tasks. For the scheduling policy, we use RMS [4] with harmonic tasks to achieve full processor utilization [22] when required. Regarding processor failures, we use a fail-stop model, in other words, we assume that a failure stops a processor completely, and completed tasks always generate correct results. Selected tasks on those stopped processors will be recovered in different processors. We also impose a *placement constraint*, which requires that replicas must be on different processors. If two or more processors fail simultaneously, the system may be able to recover through multiple Hot Standbys for hard recovery tasks and a combination of Hot Standby and Cold Standbys for soft recovery tasks.

A. System Assumptions

In order to model the behavior of each task, we assume a set of tasks, Γ. Γ is composed of n tasks, τ_1, τ_2, ..., and τ_n. These tasks are divided into three subsets of Γ, *Hard Recovery Task* set, Γ_H, *Soft Recovery Task* set, Γ_S, and *Best-effort Recovery Task* set, Γ_B, and these categories will be defined in Section III-B.

A task τ_i is represented as (C_i, T_i, D_i, α_i), where C_i is the worst-case execution time, T_i is the period, D_i is the deadline relative to the release time, and α_i represents the ratio of recovery time to deadline. The *recovery time* is defined as the time instant relative to the release time of τ_i, within which jobs of τ_i should be recovered. For example, if $\alpha_i = 1$ or $\tau_i \in \Gamma_H$, the failed job should be recovered within the original deadline, D_i, which is

equal to T_i. A task τ_i has the response time denoted as R_i, where $C_i \leq R_i \leq T_i$ is satisfied because R_i is the duration from the instant of job release to the moment of job completion of τ_i.

Every task τ_i can have $\psi(i)$ Hot Standby replicas, which can be represented by $\tau^h_{i,1}$, $\tau^h_{i,2}$, ...,$\tau^h_{i,\psi(i)}$. Either τ_i or $\tau^h_{i,0}$ denotes the primary of τ_i. Since our objective is to tolerate ρ processor failures, each task τ_i also can have $\zeta(i)$, which is $\rho + 1 - \psi(i)$ Cold Standby replicas, which can be represented as $\tau^c_{i,1}$, $\tau^c_{i,2}$, ...,$\tau^c_{i,\zeta(i)}$. In addition, u_i is the utilization of τ_i, defined as $\frac{C_i}{T_i}$.

A set of processors, P, is used for running a task set, Γ. P is composed of m homogeneous processors, P_1, P_2, ..., P_m. Then, Π_i is the set of processors which have τ_i and its Hot Standby replicas. Each element of Π_i, Π_{ij} is the processor allocated to τ_{ij}, j^{th} Hot Standby replica of task τ_i. Therefore, the placement constraint is expressed as $\forall i$, $\Pi_{ij} \neq \Pi_{ik}$, where $j \neq k$. Each processor P_k has its own failure rate, f_k, which denotes the probability of a permanent failure. We assume homogeneous processors for convenience of presentation, $f_1 = f_2 = ... = f_m = f$.

B. Recovery-timing Requirements and Our Approaches

Real-time systems are composed of multiple tasks with strict timing constraints. In such systems, fault recovery entails not only restoring functionality but also doing so within a bounded time. A classical example would be a feedback control system, where failure to recover within a bounded time could cause the physical state of the controlled system to move into an unsafe state. Automotive and avionics systems are examples of such applications. Depending on the timing constraint imposed on recovery time, we classify tasks into three categories:

- *Hard Recovery:* Tasks that need to meet their original deadlines relative to their release time even under failures are denoted as Hard Recovery tasks. In other words, α_i for a task in this category is 1. Meeting the original deadline provides very little room for recovery and re-execution. These tasks are the most difficult ones to provision for failure recovery.
- *Soft Recovery:* Tasks with relaxed recovery deadlines are denoted as Soft Recovery tasks. In this case, α_i for these tasks is greater than 1, but bounded.
- *Best-Effort Recovery:* Tasks that do not have any recovery deadlines and only require recovery from a functional perspective are called as Best-Effort Recovery. For tasks in this class, α_i may not be bounded.

Task replication is a fundamental technique used to improve system reliability. By introducing task replicas on multiple processors, tasks on failed processors can be recovered. In this paper, we consider two techniques for task replication viz. *Hot Standby* and *Cold Standby* with different timing characteristics.

- *Hot Standby Approach:* This approach uses two or more on-line copies of a certain task. One or more replicas of the task will be running simultaneously. Each replica must be on a different processor, in order to make sure that at least one of them is working when a processor failure occurs. When a failure occurs, a replica will take over the task on a failed processor. For the Hot Standby approach, at least one of the backup replicas should meet the original deadline when a failure occurs. In order to support this tight constraint, we use Hot Standby replicas, whose jobs are released synchronously with that of the *Primary* copy and execute in parallel with the same deadline. Any Hot Standby can be promoted to be the primary after recovery.
- *Cold Standby Approach:* In this case, replicas are not executed until failures occur. This means that they only use up memory, but not processor under normal operation, and they are triggered on demand when failures occur. Because Cold Standby are not running under normal conditions, only the state information of each primary copy needs to be shared among replicas. When failures occur, they should be detected as soon as possible, and the tasks on failed processors should be recovered using replicas within a predefined time. For the purposes of this paper, we consider that faults are detected instantaneously, and all task information is available in memory for fault recovery. This approach can recover both soft recovery tasks and best-effort recovery tasks.

C. Determination of Standby Type

The benefit of using a Hot Standby is the ability to meet the original deadline with less timing penalties than the Cold Standby approach, since all Hot Standbys are running concurrently with a primary task. The primary copy and the backup copy are released at the same time. However, this does not mean that all the replicas complete at the same time. We use the same deadline for every replica, therefore, all replicas are guaranteed to finish by the deadline. The disadvantage of using this scheme is that it requires more resources than single copy execution. Specifically, for tasks with Hot Standbys, additional required resource is $\psi(i)$ times u_i for τ_i.

For the Cold Standby approach, when there are no failures, only the primary copy is executed during the normal operation. Since we assume failures can be detected through properties of fail-stop processors, the failure of a primary copy can trigger the execution of a backup copy. For bounding recovery time, we propose a type of utilizations, *Transient Overload Utilization* (TOU). TOU is the additional utilization while a task is being recovered. In other words, TOU is a required resource for reexecuting a task on a new processor within the remaining time, $\alpha_i T_i - R_i$, after τ_i fails. Let u^t_i denote TOU of τ_i. Then, we have the following *Theorem*.

Theorem 3.1: If $\alpha_i \geq 2$, $u^t_i \leq u_i$ for any task.

Proof: In TOU, u^t_i, the backup copy of τ_i should meet the primary's original deadline by using a Cold Standby, which implies that $\alpha_i T_i - R_i \geq C_i$ should be satisfied. This is because the remaining time for computation after τ_i's execution should be enough for one more execution in the worst case. Therefore, u^t_i is denoted as $\frac{C_i}{\alpha_i T_i - R_i}$. Then, u^t_i is equal to u_i when $\alpha_i = 1 + u_i$ for the best case ($R_i = C_i$) and $\alpha_i = 2$ for the worst case ($R_i = T_i$). Because $2 \geq 1 + u_i$, $u^t_i \leq u_i$ is satisfied when $\alpha_i \geq 2$ for any task. ■

In most embedded systems, only certain tasks are safety-critical. By using a large α_i, u^t_i can be relaxed. The importance of *Theorem* 3.1 is seen in the following example. Suppose that a task

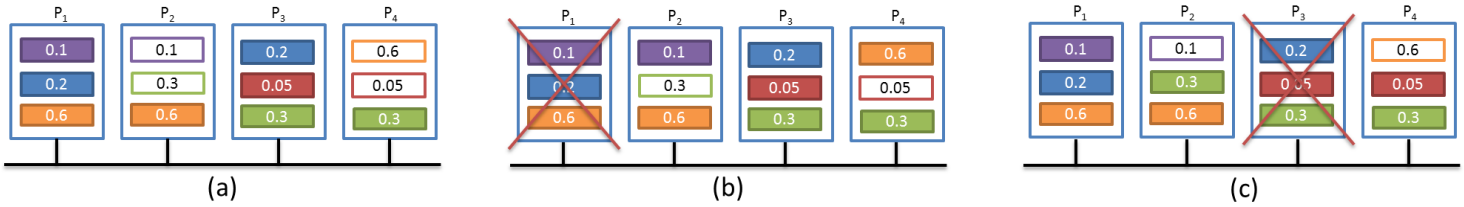


Figure 1. Example operation scenario of Hot Standby and Cold Standby. (a) normal case (b) P_1 has failed, and the tasks on P_1 have been recovered. (c) P_3has failed, and the tasks on P_3 have been recovered.

with $\alpha_i = 1$ uses a Cold Standby. Then, the backup copy of τ_i should meet the condition $T_i - R_i \geq C_i$. Therefore, $u_i^t = \frac{C_i}{T_i - R_i}$. Since $u_i^t \geq u_i$, another processor which runs the Cold Standby of the primary task which is using only Cold Standby should have more utilization value than u_i. Therefore, based on α_i, we can choose the type of Standby. An important observation in the Cold Standby approach is that the processor running the backup should have enough unused utilization. This is also applicable to other processors, which should take over other tasks that are not running at the same time on the failed processor. The reserved slack for Cold Standby replicas can be used by any task but the reserved utilization for Hot Standby cannot be utilized by other tasks.

In this paper, we will use both Hot Standby and Cold Standby within the same system as necessary. In order to decrease the number of processors required to achieve reliability requirements and tolerate simultaneous failures, Cold Standby can be promoted to the Hot Standby if the primary of τ_i fails and the current number of τ_i is less than $\psi(i)$. Since at least one Hot Standby for certain τ_i can meet its original deadline, D_i, we are ready to tolerate another potential failure if a Cold Standby can be promoted to a Hot Standby. By using this, we can tolerate as many as ρ failures for τ_i.

Figure 1 has an example scenario. Suppose a task set $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, and tasks have utilizations, 0.6, 0.3, 0.2, 0.1, and 0.05, respectively. With respect to replicas:

- τ_1, τ_2 have both 1 Cold Standby and 1 Hot Standby
- τ_3 has 1 Hot Standby
- τ_4, τ_5 have 1 Cold Standby each

These tasks are allocated to a set of processors, P, which has four processors. The allocation is shown in Figure 1(a). In Figure 1(b), P_1 has failed, and P_1 holds for τ_1, τ_3, and τ_4. Because τ_1 has one Hot Standby on P_2 and one Cold Standby on P_4, τ_1 is recovered by a Hot Standby on P_2. Then, the Cold Standby on P_4 is promoted to a Hot Standby. τ_3 is recovered by Hot Standby on P_3, and τ_4 is recovered by the Cold Standby on P_2. Similar operations happen when P_3 also has failed in Figure 1(c).

IV. Task Allocation with Hot Standby and Cold Standby Replication

Allocating tasks to processors is a well-known bin-packing problem [23]. In this paper, we consider the design-time allocation of tasks and it is well-known that this allocation is NP-hard [24]. There are several popular heuristics such as BFD, FFD, NFD (Next-Fit Decreasing) , and WFD (Worst-Fit Decreasing). Each of these algorithms regards the utilization value of each task as the object size and allocates each task to a proper processor. In this paper, we will take BFD as a base-line algorithm.

A. Fault-tolerant Partitioned Scheduling

The main difference between conventional and fault-tolerant partitioned scheduling is a placement constraint in order to improve system reliability by spreading copies of the primary and the replicas. For task allocation, the original BFD 1) sorts the tasks in descending order of size, 2) fits the next task into the best processor that it can fit into, 3) adds a new processor if a task does not fit into any current processor, and 4) iterates this procedure until no tasks remain. In the current context, BFD should be modified to satisfy the placement constraint of replicas. Therefore, two steps are changed. For step 1), when the tasks are sorted, the replicas are sorted together with their primaries. For step 2), the processor should be determined using placement constraints. This algorithm is denoted as BFD-P [21] and is listed in Algorithm 3.

B. Task Allocation with Hot Standby using R-BFD

Consider a task set $\Gamma : \{\tau_1, \tau_2, ..., \tau_n\}$, where each task τ_i has a primary copy τ_i and $\psi(i)$ *Hot Standby* replicas $\{\tau_{i,1}^h, ..., \tau_{i,\psi(i)}^h\}$. The placement constraint dictates that none of these copies get co-located on the same processor.

R-PACK is a basic algorithm for allocating a set of n objects $\Omega : \{O_1, ..., O_n\}$ with placement constraints $\Pi : \{\Pi_1, ..., \Pi_n\}$ (see Algorithm 1). It iterates over the objects in the given order, allocating each object O_i to the best-fit processor $\Pi_{ij} \leftarrow P_k$ that satisfies the placement constraint $P_k \notin \Pi_i$. If no existing processor can fit O_i, then a new processor is added for it. The set of processors and updated placement constraints are obtained from R-PACK.

R-BFD sorts the given tasks in decreasing order of sizes (see Algorithm 2). The primary copies are first allocated using R-PACK. The *Hot Standby* replicas are then allocated in batches using R-PACK. The key distinction here is that BFD-P (see Algorithm 3) would allocate the whole set of tasks (including Hot Standby replicas) with placement constraints. By operating in batches, R-BFD can better fill up the space left over in the previous processors, whereas BFD-P would lead to wastage of this space. This effect is shown in Figure 2. Figure 2 shows the result of BFD-P in 2(a) and the result of R-BFD in 2(b) under

Algorithm 1 R-PACK($\Omega : \{O_1, ..., O_n\}$,$\Pi : \{\Pi_1, ..., \Pi_n\}, P : \{P_1, ..., P_m\}$)

for $i = 1$ to n **do**
  ▷ *Only worry about non-empty object list*
  if $O_i \neq \emptyset$ **then**
    For O_i, find a best-fit processor P_k, s.t. $P_k \notin \Pi_i$
    if P_k exists **then**
      $\Pi_i \leftarrow \Pi_i \cup P_k$ ▷ *Allocate to existing processor*
    else
      $\Pi_i \leftarrow \Pi_i \cup P_m$ ▷ *Need to add new processor*
      $P \leftarrow P \cup \Pi_i$
      $m \leftarrow m + 1$
return (P, Π)

Algorithm 2 R-BFD($\Gamma : \{\tau_1, \tau_2, ..., \tau_n\}$)

Sort Γ in descending order of utilization
▷ *Allocate the primary copies first*
$(P, \Pi) \leftarrow$ R-PACK($\Omega \leftarrow \{\tau_1, \tau_2, ..., \tau_n\}$,
  $\Pi \leftarrow \{\Pi_1 \leftarrow \emptyset, ..., \Pi_n \leftarrow \emptyset\}$, $P \leftarrow \emptyset$)
▷ *Allocate the replicas one by one*
for $j = 1$ to $\max_{\forall \tau_k \in \Gamma}(\psi(k))$ **do**
  ▷ *Ignore tasks that do not need j replicas*
  $\forall \tau_i$ s.t. $\psi(i) < j$, $\tau^h_{i,j} \leftarrow \emptyset$
  $(P, \Pi) \leftarrow$ R-PACK($\Omega \leftarrow \{\tau^h_{1,j}, \tau^h_{2,j}, ..., \tau^h_{n,j}\}$, Π, P)
return (P, Π)

the given task set $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ with one Hot Standby and utilization values, 0.6, 0.3, and 0.2, respectively. In this example, R-BFD saves 1 processor by allocating the Hot Standy replicas in batches. Since BFD-P assigns τ_3 to a new processor, placement constraints bring one more processor for $\tau^h_{3,1}$.

C. Analysis

In order to abstract away from the performance of the individual uniprocessor scheduling algorithm used within the processors themselves and focus on the task allocation, we restrict our analysis to an optimal uniprocessor scheduling configuration. Under RMS, we focus on harmonic task sets that can achieve 100% utilization. Generalizing to arbitrary task periods would lead to non-ideal processor utilization due to the relationship between task periods [4], thereby detracting away from the properties of the task allocation algorithm itself. In the following discussion of BFD and R-BFD properties, we focus on harmonic tasks, although the algorithms themselves can be applicable to arbitrary task sets.

Lemma 4.1: For harmonic task sets with $\psi(i) = 0\ \forall i$, BFD heuristic requires at most $M_0 = (\frac{11}{9}(OPT_0) + 4)$ processors, where OPT_0 is the number of processors required by an optimal algorithm.

Proof: Harmonic task sets result in a schedulable utilization bound of 100% in each processor. The task allocation problem is therefore reduced to the standard bin-packing problem, for which BFD has been established to require no more than $\frac{11}{9}(OPT_0) + 4$ processors [25] when the optimal algorithm requires OPT_0 processors. ■

Lemma 4.2: For harmonic task sets with $\psi(i) = k\ \forall k$, where k is a positive integer constant, BFD requires at most $k(\frac{11}{9}(OPT_0) + 4)$ processors, where OPT_0 is the number of processors that would be required by an optimal algorithm to schedule the same task set with $k = 0$.

(a) BFD-P (b) R-BFD

Figure 2. Benefit of using R-BFD

Proof: Let the optimal number of processors required to schedule the same task set assuming $\psi(i) = 0$ be OPT_0. The number of processors required by BFD to schedule the same task set under $\psi(i) = 0$ is M_0. We know that $M_0 \leq \frac{11}{9}(OPT_0) + 4$ (by Lemma 4.1). Due to the sorted order in which BFD considers objects and the placement constraints, when $\psi(i) = k$ the number of processors required by BFD is kM_0. This can be shown from Algorithm 3, which results in k identical processors following every $(k+1)$th processor if $\psi(i) = k$. ■

Lemma 4.3: For harmonic task sets with $\psi(i) = k\ \forall k$, R-BFD requires no more processors than BFD.

Proof: This follows from the fact that Hot Standby replicas in Algorithm 2, R-BFD at least consider the same candidate processors for allocation as in Algorithm 3, BFD-P. R-BFD therefore requires no more processors than BFD-P. ■

Corollary 4.4: For harmonic task sets with $\psi(i) = k\ \forall k$, where k is a positive integer constant, R-BFD requires at most $k(\frac{11}{9}(OPT_0) + 4)$ processors, where OPT_0 is the number of processors that would be required by an optimal algorithm to schedule the same task set with $k = 0$.

Proof: Follows from Lemmas 4.2 and 4.3. ■

D. Dealing with System Reliability Requirements

Consider a uniform multiprocessor system with m processors. Let F denote the system SIL (Safety Integrity Level) [26] requirement specified in terms of the PFD (Probability of Failure on Demand). Let the reliability specification of each individual processor be f, denoting that the processors are designed to have a PFD less than f. Based on F and f, the system designer can estimate ρ, which is the minimum number of additional processors required to satisfy the system reliability. ρ should be greater than the maximum number of processor failures that can be expected in $(m + \rho)$ processors.

$$\rho = \min_p\{p \in Z | F \geq \sum_{j=p}^{m+p} Prob(\text{exactly } j \text{ processor failures})\}$$

$$\rho = \min_p\{p \in Z | F \geq \sum_{j=p}^{m+p} \binom{m+p}{j} f^j (1-f)^{m+p-j}\} \quad (1)$$

Algorithm 3 BFD-P($\Gamma : \{\tau_1, \tau_2, ..., \tau_n\}$)

Sort Γ in decreasing order of utilization
$\tau_{i,0} \leftarrow \tau_i$
for i in 1 to n **do**
  for j in 0 to $\psi(i)$ **do**
    For $\tau_{i,j}$, find a best-fit processor P_k s.t. $P_k \notin \Pi_i$
    if P_k exists **then**
      $\Pi_{ij} \leftarrow P_k$ ▷ *Allocate to an existing processor*
    else
      $\Pi_{ij} \leftarrow P_m$ ▷ *Need to create a new processor*
      $P \leftarrow P \cup \Pi_i$
      $m \leftarrow m + 1$
    $\Pi_i \leftarrow \Pi_{ij} \cup \Pi_i$ ▷ *Update placement constraints*
return (P, Π)

Algorithm 4 R-BATCH: Allocate the given task set Γ to processors P for handling ρ processor failures

$P \leftarrow \emptyset$
$(P, \Pi) \leftarrow$ R-BFD(Γ)
$(\Gamma, \Pi) \leftarrow$ generateVirtualTask(Γ, Π, P, ρ)
$(P, \Pi) \leftarrow$ R-PACK(Γ, Π, P)
return (P, Π)

A system designer may choose a value of ρ greater than the one obtained using Equation (1) depending on design margins.

E. Task Allocation with Cold Standby using R-BATCH

Allocating tasks with *Cold Standby* replication in addition to *Hot Standby* replicas is accomplished by R-BATCH (Reliable-Bin-packing Algorithm for Tasks with Cold Standby and Hot Standby) given in Algorithm 4. The basic idea behind the algorithm is to estimate $\zeta(i) = \rho + 1 - \psi(i)$, the number of Cold Standbys needed for task τ_i. We observe that the Cold Standby replicas for processors other than those hosting τ_i can be consolidated. Suppose a task set $\Gamma = \{\tau_1, \tau_2\}$, and both tasks have 0.6 utilizations. Since they cannot fit into one processor together, two processors are necessary for Γ. In order to tolerate one failure per task by using a Hot Standby, two more processors are required. A Cold Standby, however, can reduce one processor compared to exploiting a Hot Standby by sharing one unused processor.

R-BATCH creates virtual task, $\tau^v_{q,j}$, where q distinguishes each virtual task and j denotes j^{th} Cold Standby replica of tasks covered by $\tau^v_{q,j}$. Then, $\Gamma^v_{q,j}$ is a set of tasks which can be taken over by $\tau^v_{q,j}$. Each virtual task is responsible for handling the Cold Standby replica for multiple processors. As it is assumed that no more than ρ distinct processors will fail during the system runtime, the virtual tasks can consolidate the replication and reduce the number of processors required considerably. The procedure for generating virtual tasks is defined in Algorithm 5.

After generating virtual tasks, the placement constraint should be satisfied. This constraint is considered by following three *Lemmas*.

Algorithm 5 generateVirtualTask($\Gamma : \{\tau_1, ..., \tau_n\}, \Pi : \{\Pi_1, ..., \Pi_n\}, P, \rho$)

$q \leftarrow 0$
for $j = 0$ **to** $\max_{\forall \tau_i \in \Gamma}(\zeta(i))$ **do**
  for all τ_i such that $\tau_i \in \Gamma$ **do**
    if $j < \zeta(i)$ and $\tau^c_{i,j} \notin \Gamma^v$ **then**
      $\Gamma^v_{q,j} \leftarrow \{\tau^c_{i,j}\}$ ▷ *Generate virtual task* $\tau^v_{q,j}$
      $q \leftarrow q + 1$
      $u^v_{q,j} \leftarrow u_i$ ▷ *Set the virtual utilization of* $\tau^v_{q,j}$
      $\Pi^v_{q,j} \leftarrow \Pi^v_{q,j} \cup \Pi_i$ ▷ *Set the placement constraint*
      ▷ *Pick a processor not containing copies of* τ_i
      for all P_k such that $P_k \notin \Pi_i$ **do**
        $alloc \leftarrow 0$
        for all $\forall \tau_p$ such that $P_k \in (\Pi_p - \Pi_i)$ **do**
          ▷ *Allocate* τ_p *to* $\tau^v_{q,j}$ *if possible*
          if $alloc + u_p \leq u^v_{q,j}$ and $\tau^c_{p,j} \notin \Gamma^v$ **then**
            $\Gamma^v_{q,j} \leftarrow \Gamma^v_{q,j} \cup \{\tau^c_{p,j}\}$
            $alloc \leftarrow alloc + u_p$
        if $alloc \neq 0$ **then**
          $\Pi^v_{q,j} \leftarrow \Pi^v_{q,j} \cup P_k$
      $\Gamma^v \leftarrow \Gamma^v \cup \Gamma^v_{q,j}$, $\Pi^v \leftarrow \Pi^v \cup \Pi^v_{q,j}$
return $(\Gamma^v \cup \Gamma, \Pi^v \cup \Pi)$

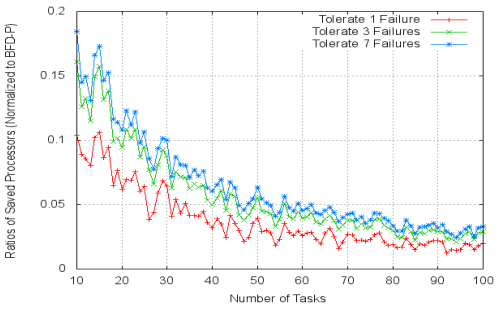
Lemma 4.5: Two virtual tasks $\tau^v_{q,j}$ and $\tau^v_{p,j}$, where $q \neq p$, can be located on the same processor.

Lemma 4.6: Two virtual tasks $\tau^v_{q,i}$ and $\tau^v_{p,j}$, where $q \neq p$ and $i \neq j$, can be located on the same processor if $\Gamma^v_{q,i} \cap \Gamma^v_{p,j} = \emptyset$ is satisfied.

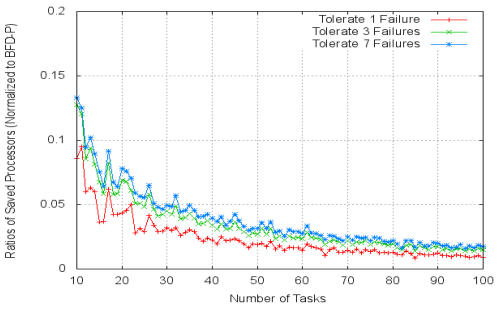
Lemma 4.7: $\tau^v_{q,j}$ and τ_i can be located on a same processor if $\tau_i \notin \Gamma^v_{q,j}$ is satisfied.

V. Evaluation

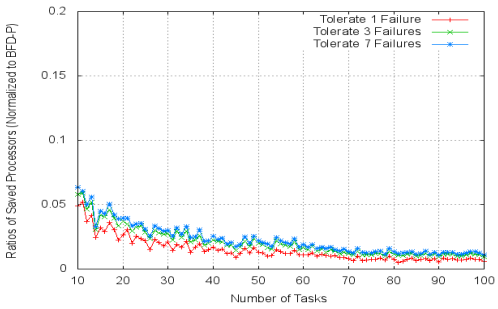
We will now evaluate the performance benefits of using R-BATCH and R-BFD on randomly generated task sets. We analyze task sets with different characteristics by varying the maximum task utilization (u_{max}). The results presented here are at u_{max} values of 0.3, 0.5, and 0.7. The number of tasks is varied from from 10 to 100. We characterize the performance with respect to tolerating 1, 3, and 7 processor failures by introducing 2, 4, and 8 replicas respectively (including the primary) under R-BFD. With R-BATCH, we set the number of Hot Standbys for each task to 1, 2, and 4 Hot Standby replicas including the primary. Any possible remaining failures can be recovered by using a Cold Standby. Because we are using Hot Standby and Cold Standby together, all tasks can be recovered even if all of them have Hard Recovery requirements. Tasks can be promoted from Cold Standby to Hot Standby when failures occur. We provide results from both single-node and 4-node platforms to illustrate the performance benefits, where a 4-node platform has 4 processors per board. Since 4-node platform is augmented at the granularity of boards, it can fail under only a single processor failure out of 4 processors. Each data point is obtained by averaging sum of all results from 50 iterations.



(a) $u_{max} = 0.3$

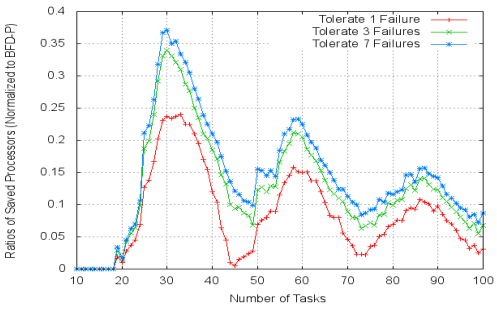


(b) $u_{max} = 0.5$

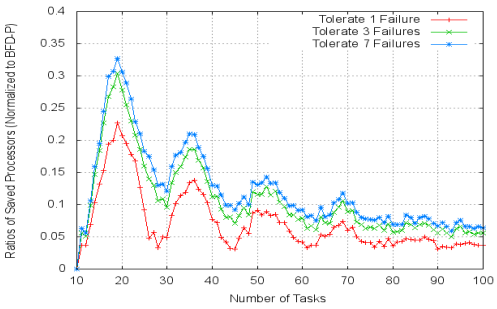


(c) $u_{max} = 0.7$

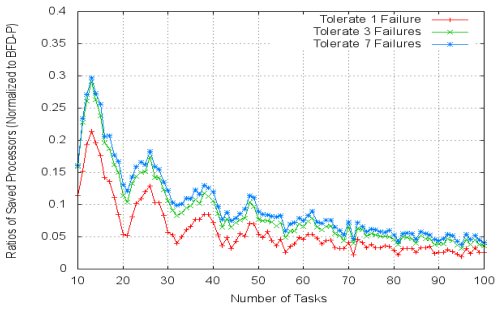
Figure 3. Ratios of saved processors when R-BFD are used on Single-node Case. Results are normalized to BFD-P.



(a) $u_{max} = 0.3$

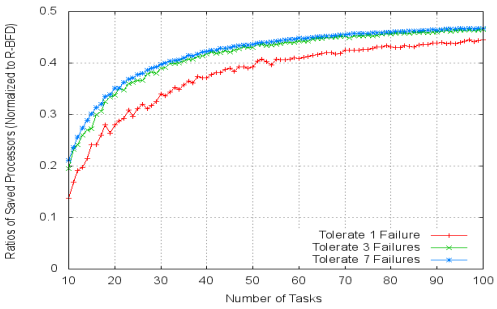


(b) $u_{max} = 0.5$

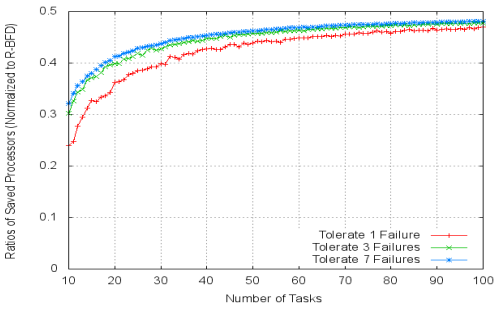


(c) $u_{max} = 0.7$

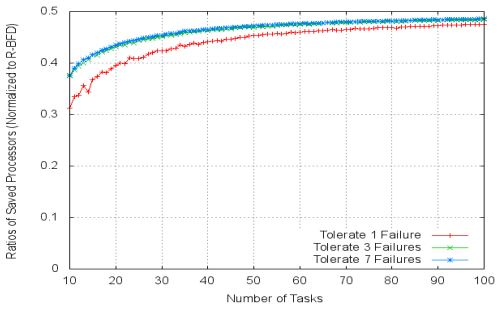
Figure 4. Ratios of saved processors when R-BFD are used on 4-node Case. Results are normalized to BFD-P.



(a) $u_{max} = 0.3$

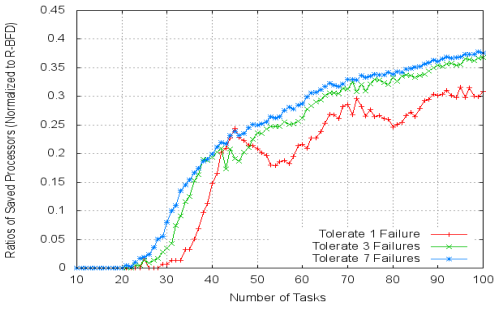


(b) $u_{max} = 0.5$

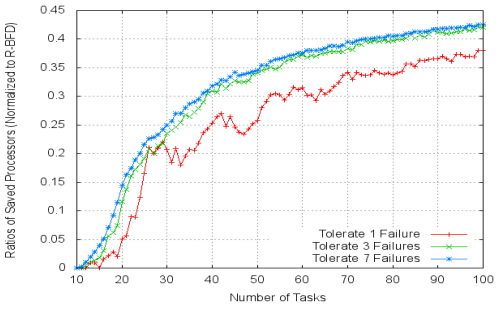


(c) $u_{max} = 0.7$

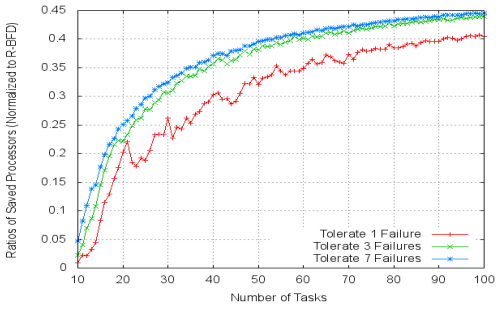
Figure 5. Ratios of saved processors when R-BATCH are used on Single-node Case. Results are normalized to R-BFD.



(a) $u_{max} = 0.3$



(b) $u_{max} = 0.5$



(c) $u_{max} = 0.7$

Figure 6. Ratios of saved processors when R-BATCH are used on 4-node Case. Results are normalized to R-BFD.

Figure 3 and 4 show the number of processors saved by R-BFD over BFD-P, normalized to BFD-P on single-node and 4-node cases. Figure 3(a), 3(b), and 3(c) show the results at different u_{max} values. For small u_{max} and task set sizes, R-BFD is most beneficial. R-BFD can save up to *19%* on the single-node case. For the 4-node case, it can save up to *37%* processors compared with BFD-P. Regarding Figure 4(a), there are no differences when the number of tasks is changed from 10 to 19. This happens due to the nature of the 4-node case. Because we merge 4 processors into a set of processors, we have more space for allocating tasks. Therefore, until a certain number of tasks fills up 4 processors without replication, there is no difference. This is also a reason for the fluctuations seen in Figure 4.

Figure 5 and 6 show the number of processors saved by R-BATCH over R-BFD, normalized to R-BFD. R-BATCH can save up to *45%* additional processors compared to R-BFD. The interesting observation is that benefits get larger when more tasks are used. This is because we can consolidate more tasks by using virtual tasks. Fluctuations in Figure 6(a) are due to the same reasons given above for R-BFD. Another interesting observation is that R-BATCH can save more platforms as u_{max} increases. This is a consequence of the fact that larger virtual tasks can cover more Cold Standbys.

VI. Conclusion

Fault recovery in hard real-time environments requires restoring functionality within pre-specified deadlines. In this work, we have provided a comprehensive solution for guaranteeing reliability requirements with bounded recovery times. We have proposed categorizing tasks based on their recovery time requirement into (i) *Hard Recovery*, (ii) *Soft Recovery*, and (iii) *Best-Effort Recovery*. We then developed a task-partitioning strategy called *R-BFD* for allocating *Hot Standby* replicas to processors in order to improve system reliability. In order to further reduce the resource over-provisioning required for task reliability, we introduced the notion of a *Cold Standby* that consumes processing time only when activated. Our consolidated task allocation algorithm called *R-BATCH* can allocate both *Hot Standby* and *Cold Standby* tasks to meet system-level reliability requirements. Evaluation results suggest that R-BFD saves up to *37%* of the required number of processors, while achieving the same levels of reliability and satisfying the recovery time requirements as the conventional BFD-P heuristic on 4-node platform case. The introduction of *Cold Standby* can save up to *45%* additional processors using R-BATCH, in comparison to R-BFD with pure *Hot Standby* replicas.

References

[1] C. Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. In *The DARPA Urban Challenge*, pages 1–59. 2009.

[2] T. Skotnicki, J. A. Hutchby, T. J King, H. S.P Wong, F. Boeuf, and S. T. Microelectronics. The end of CMOS scaling: toward the introduction of new materials and structural changes to improve MOSFET performance. *IEEE Circuits and Devices Magazine*, 21(1):1626, 2005.

[3] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[4] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, 1973.

[5] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26, 1978.

[6] M.L. Dertouzos and A.K Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15:1497–1506, 1989.

[7] K. Ramamritham, J.A. Stankovic, and P.F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1:184–194, 1990.

[8] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, page 93, Washington, DC, USA, 2001. IEEE Computer Society.

[9] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems Journal*, 38:223–235, 2008.

[10] Nathan Fisher, Sanjoy Baruah, and Theodore P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 118–127, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. *Real-Time Systems, Euromicro Conference on*, 0:239–248, 2009.

[12] Shinpei Kato and Nobuyuki Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 441–450, Washington, DC, USA, 2007. IEEE Computer Society.

[13] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[14] Jean-Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23:39–51, 1990.

[15] Balachandran Natarajan, Aniruddha Gokhale, Shalini Yajnik, and Douglas C. Schmidt. Doors: Towards high-performance fault tolerant corba. *Distributed Objects and Applications, International Symposium on*, 0:39, 2000.

[16] G. Manimaran, C. Siva, and R. Murthy. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Trans. on Parallel and distributed Systems*, 1998.

[17] S. Punnekkat, A. Burns, and R. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83102, 2001.

[18] Sunondo Ghosh, Rami Melhem, Daniel Moss, and Joydeep Sen Sarma. Fault-Tolerant Rate-Monotonic scheduling. *Real-Time Systems*, 15(2):149–181, 1998.

[19] C M Krishna and K G Shin. On scheduling tasks with a quick recovery from failure. *IEEE Trans. Comput.*, 35(5):448–455, 1986.

[20] Alan A. Bertossi, Luigi V. Mancini, and Federico Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Trans. Parallel and Distributed Systems*, 10:934–945, 1999.

[21] Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Shau-Yin Tseng. Real-Time task replication for fault tolerance in identical multiprocessor systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 249–258. IEEE Computer Society, 2007.

[22] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. *Proceedings of the Real-Time Systems Symposium (RTSS'89)*, pages 166–171, 1989.

[23] D. Oh and T. Baker. Utilization bounds for n-processor rate monotonic scheduling with static processor assignment. *Real-Time Systems*, 15:183–192, 1998.

[24] D. S. Johnson. Near optimal allocation algorithms. *Ph.D. Dissertation, MIT, Cambridge, MA*.

[25] DS Johnson, A. Demers, JD Ullman, MR Garey, and RL Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:299, 1974.

[26] *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission (IEC), December 1998.