DETERMINING REDUNDANCY LEVELS FOR FAULT TOLERANT REAL-TIME SYSTEMS

F. WANG, K. RAMAMRITHAM, and J.A. STANKOVIC

CMPSCI Technical Report 93-82

November 1993

Determining Redundancy Levels for Fault Tolerant Real-Time Systems *

Fuxing Wang Krithi Ramamritham John A. Stankovic

Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

November 1993

Abstract

Many real-time systems have both performance requirements and reliability requirements. Performance is usually measured in terms of the value in completing tasks on time. Reliability is evaluated by hardware and software failure models. In many situations, there are tradeoffs between task performance and task reliability. Thus, a mathematical assessment of performance-reliability tradeoffs is necessary to evaluate the performance of real-time fault-tolerance systems.

Assuming that the reliability of task execution is achieved through task replication, we present an approach that mathematically determines the replication factor for tasks. Our approach is novel in that it is a task schedule based analysis rather than a state based analysis as found in other models. Because we use a task schedule based analysis, we can provide a fast method to determine optimal redundancy levels, we are not limited to hardware reliability given by constant failure rate functions as in most other models, and we hypothesize that we can more naturally integrate with on-line real-time scheduling than when state based techniques are used. In this work, the goal is to maximize the total performance index, which is a performance-related reliability measurement. We present a technique based on a continuous task model and show how it very closely approximates discrete models and tasks with varying characteristics.

Key Words — Real-Time Systems, Reliability, Degradable Systems, Fault Tolerance, Functional Variation, Performability

*This work is part of the Spring Project at the University of Massachusetts and is funded in part by the Office of Naval Research under contract N00014-92-J-1048 and by the National Science Foundation under grant CDA-8922572.

1 Introduction

To simultaneously support high performance, flexibility, and reliability requirements of complex systems involves many tradeoffs. In real-time systems, schedulability analysis is often used to *guarantee* that tasks meet their time constraints. A task is guaranteed subject to a set of assumptions, for example, about its worst case execution time, resource needs, and the nature of faults in the system. If these assumptions hold, once a task is guaranteed it can be assumed to meet its timing requirements. Thus, the probability of a task's successful completion is affected by the probability with which the assumptions hold.

Let us consider a simple scenario to illustrate some of the tradeoffs involved. Assume that reliability of task execution is achieved through task replication. If we increase the redundancy level for a task we increase its probability of completing before its deadline, that is, decrease the probability that the task will fail after being guaranteed. However this reduces the number of tasks that get guaranteed in the first place and so increases the penalties due to task rejections. Clearly, there are tradeoffs involved between the fault tolerance of the system, the rewards provided by guaranteed tasks that complete successfully, and the penalties due to tasks that fail after being guaranteed or that fail to be guaranteed. Therefore, ways to maximize rewards while minimizing penalties must be found. In the current state of the art, most approaches rely on state based analysis and are applied to static systems.

The work presented here is an extension to the performance-related reliability assessment, but it is based on a task level analysis rather than a state based model. This allows us to develop a fast method to determine the optimal redundancy levels of tasks without explicitly referring to states and without using any expensive algorithms for exhaustive search. Also, our method is not limited to hardware reliability given by constant failure rate functions as in most other models, since we do not depend on the memoryless property. Further, we hypothesize that using analysis on a task basis more naturally integrates with dynamic, on-line, real-time scheduling. For clarity of presentation, we present our solution in the context of systems where decisions are made statically, i.e., at system design time, even though the approach developed in this paper can be tailored to apply to systems that perform dynamic schedulability analysis.

Consider a system with m processors and n tasks. We must decide what level of redundancy should be assigned to the tasks such that both reliability and performance requirements are met. We use the term *task configuration* or *configuring tasks* to refer to the problem of determining a task's redundancy level. Once a task redundancy level is determined, a task is said to be

guaranteed if the given number of replicas of the task are all scheduled to complete before the task's deadline.

In particular, suppose a task T_i provides a reward V_i if it completes successfully once it is guaranteed, a penalty P_i if it fails after being guaranteed, and a penalty Q_i if it is not guaranteed. Let R_i be the reliability of a guaranteed task T_i and F_i be its failure probability, with $R_i = 1 - F_i$. R_i is mainly affected by the redundancy level for a task T_i and the failure model for the processors. Then, we define a performance index for the system such that it takes the tasks' penalties, rewards, and reliabilities into account. The performance index PI_i for task T_i is defined as

$$PI_i = \begin{cases} V_i R_i - P_i F_i & \text{if } T_i \text{ is guaranteed} \\ -Q_i & \text{if } T_i \text{ is not guaranteed.} \end{cases}$$

The performance index PI for the task set is defined as

$$PI = \sum_{i=1}^{n} PI_i.$$

Thus PI accounts for both performance requirements and reliability requirements of real-time tasks. It provides us a base for achieving a mathematical assessment of performance-reliability tradeoffs.

Given this definition of performance index, we need to know the reliability for each scheduled task. Tasks' reliabilities are affected by faults in both software and hardware. Many fault-tolerance structures have been developed to tolerate these two types of faults. For example, *task replication* and *N-modular-redundancy* are commonly used to tolerate hardware faults, and *recovery block* and *N-version programming* are commonly used to tolerate software faults. The quantitative models for hardware reliability are well established [3], while the quantitative models for software reliability are still not fully understood. Because of this we focus on hardware faults and the related quantitative reliability models.

In general, performance-related reliability models use a state-based approach by assigning some kind of performance value or reward to a system's various working configurations. Using continuous-time Markov chain model, a degradable multiprocessor is expressed as an n-state process with state space as 0, 1, $\cdots$, n. State 0 represents the system failed state and state 1 through n represent various working configurations. Each working state i is associated with a reward rate r_i. Solving this Markov chain model yields the probability that the system is in different working state at time t.

Beaudry introduced measures such as computation reliability and computation availability for degradable multiprocessors [1]. Beaudry's model is built on the Markov reward process. The concepts in Beaudry's model were generalized by Meyer, who introduced performability [9], which is the probability distribution function of accumulated system performance. Lee and Shin introduced an active reconfiguration strategy for a degradable multimodule computing system with a static set of tasks [6]. They recognized that the system should reconfigure itself not only when a failure occurs, but also when it spends a certain amount time without failure. Their model is also a state-based approach which is represented as a Markov reward process. In [10] Muppala, Woolet, and Trivedi have combined two approaches for modeling soft and hard real-time systems. Their approach is based on the addition of transitions to the Markov model of a system's behavior for modeling a system failure due to the missing of a hard deadline. The system's response time and throughput distributions are used to denote the reward rates.

For these state-based approaches, tasks and task scheduling are implicitly accounted for within a system state. The number of states would explode when considering all possible subsets of tasks, their redundancy levels, and all possible feasible schedules. This may not be a major problem for small static systems. But, in a dynamic system, we cannot afford to use any time-consuming algorithm such as dynamic programming to exhaustively search for a solution and we cannot generate task schedules off-line because we do not have enough task information to make these scheduling decisions. Hence, to reduce computational complexity, we must look for alternatives.

Our main idea is to focus on one key factor which affects both system reliability and performance: the construction and use of task schedules that take into account prespecified requirements, dynamic demands, and current system status. This is the reason we call our analysis *task schedule based analysis*. The ability to construct feasible task schedules depends on tasks' redundancy levels which mainly affects system reliability and on the tasks themselves which mainly affects system performance.

Other work has developed specific algorithms or approaches to combining fault tolerance and scheduling. In [7], Liestman and Campbell propose a deadline mechanism that can guarantee that a primary task will make its deadline if there is no failure, and that an alternative task (of less precision) will run by the deadline if there is a failure. If the primary task executes then it is not necessary to run the alternative task and the time set aside for the alternative is reused. Krishna and Shin continue with this theme in [5]. Specifically, they want to be able to quickly switch to a new task schedule upon failure, where that new schedule has been precomputed.

Off-line they use a dynamic programming algorithm to compute contingency schedules which are embedded within the primary schedule. In this approach they are able to ensure that hard deadlines are met in the face of some maximum number of failures. The embedded contingency schedules are not used unless there is a failure. Approaches for fault tolerance, such as these last two papers represent, are valuable for static, embedded computer systems. However, these static approaches are not suitable for many next generation real-time systems which must provide for predictability while reacting to the dynamics of the environment.

The remainder of the paper is organized as follows. Section 2 presents notations, assumptions, and the system model. We derive the optimal task configuration strategy for a continuous model in Section 3. We discuss how to deal with a discrete model with tasks having different computation times in Section 4. In Section 5 we discuss the effects of using integer functions to approximate the optimal task configuration function which is a real valued function. In Section 6, we consider the task configuration strategy with tasks having different reward/penalty parameters. In Section 7, we discuss how to apply the task configuration theory in practice. We conclude the paper in Section 8 by discussing avenues for extending the results presented here.

2 System Model and Assumptions

In this section, we present the processor-task model first, followed by a discussion of task configuration and task scheduling.

Formally, the problem is characterized by a processor-task model given by $\{P_1, P_2, ..., P_m\}$ and $\{T_1, T_2, ..., T_n\}$.

$\{P_1, P_2, ..., P_m\}$ is a set of m identical processors in a homogeneous multiprocessor system[1]. Each processor is capable of executing any task. Processors may fail during a mission and the failed processors are assumed to be fail-stop with failures being independent. Processors are associated with the reliability function, $R(t)$, and the failure function $F(t)$, where t is the time variable and

$$R(t) = 1 - F(t). \tag{1}$$

[1]The method developed in this paper could be extended to a distributed system with m identical processor nodes. The main difficulty dealing with the distributed system is that both communication bandwidth and communication reliability should be considered as we compute task's reliability. The results of this paper are based on a simpler system model which may provide a base to deal with the issues related to systems involving communication among nodes.

There are no restrictions on the reliability function. A simple example of the reliability function is an exponential function which is widely used to model many fault-tolerance systems:

$$R(t) = 1 - e^{-\lambda t},$$

where λ is a constant representing the failure rate.

$\{T_1, T_2, ..., T_n\}$ is a set of n aperiodic tasks to be configured and scheduled on m processors in a time interval [0, L], where L is the largest deadline of the tasks. Task T_i is characterized by the following:

- e_i — its ready time, which is the earliest time the task can start,
- c_i — its computation time,
- d_i — its deadline,
- V_i — its reward, if it is serviced successfully,
- v_i — its reward rate, derived as V_i/c_i,
- P_i — its failure penalty, if it is scheduled and fails because of processor failures,
- p_i — its failure penalty rate, derived as P_i/c_i,
- Q_i — its rejection penalty, if it is rejected,
- q_i — its rejection penalty rate, derived as Q_i/c_i.

If task T_i is accepted, it gets a reward V_i if it succeeds, and gets a failure penalty P_i if it fails. If task T_i is rejected, it gets a rejection penalty Q_i. The scheduling window for task T_i is the time interval from its ready time e_i to its deadline d_i. To simplify the analysis, we assume that tasks' scheduling windows are relatively small compared to L. Further, tasks are assumed to be independent.

With the above processor-task model, our task configuration strategy assigns a redundancy level, u_i, to task T_i, for $1 \leq i \leq n$. Redundant copies of the same task are assumed to be scheduled on different processors. So u_i is bounded from above by the number of processors, m, where $1 \leq i \leq n$. A task is considered to have failed only if all its redundant copies fail.

After the task set is configured, a task scheduling algorithm attempts to generate a feasible schedule. Let u_i be the number of redundant copies of task T_i and $\vec{f}_i$ be its scheduled finish time vector made up of finish times of each copy of T_i:

$$\vec{f}_i = (f_1, f_2, ..., f_{u_i}), \text{ where } f_j \leq d_i,\ 1 \leq j \leq u_i.$$

Then its reliability and failure probability are

$$R_i = 1 - F_i \tag{2}$$

and

$$F_i = F(f_1)F(f_2)\cdots F(f_{u_i}). \tag{3}$$

Now we can define the *performance index* PI_i for task T_i. If the task is feasibly scheduled, i.e., guaranteed, it contributes a reward V_i with a probability R_i and a failure penalty P_i with a probability F_i. Thus, we have

$$\begin{aligned} PI_i &= V_i R_i - P_i F_i \\ &= c_i v_i - c_i(v_i + p_i)F_i. \end{aligned} \tag{4}$$

On the other hand, if T_i is rejected, the task contributes a rejection penalty Q_i. In this case, we have

$$PI_i = -Q_i = -c_i q_i. \tag{5}$$

Our goal is to maximize the total performance index PI,

$$PI = \sum_{i=1}^{n} PI_i. \tag{6}$$

PI is mainly determined by tasks' reliabilities and reward/penalty parameters. Tasks' reliabilities are determined by their redundancy levels which can be controlled within the task configuration phase, but we cannot change tasks' reward/penalty parameters. We present a simple example to demonstrate the relationship of PI and task redundancy level.

Example 1: Assume there is a multiprocessor with ten processors and there are ten tasks with their parameters listed in Table 1. All scheduled tasks will finish at time 10. If we assume each processor has a reliability of 0.9, then the reliability of a task is 0.9 when its

redundancy is one and it is 0.99 when its redundancy is two, etc. Table 2 shows the values of PI for different redundancy levels (u). The maximum PI is reached when all scheduled tasks have their redundancy levels at two ($u = 2$). According to PI, the redundancy level is not enough if $u < 2$ and it is too much if $u > 2$. So the idea we are following in the remainder of the paper is to derive the optimal redundancy level required at each time instance within L. Then, knowing this time-dependent optimal redundancy level, we can determine how much load to shed and based on this we can configure the task set accordingly (see Section 7).

Table 1: Task parameters for Example 1

Task	e_i	c_i	d_i	V_i	P_i	Q_i
$T_1, T_2, \cdots T_{10}$	0	10	10	10	100	1

Table 2: Relations between u and PI for Example 1

u	$PI = \sum_{i=1}^{10} PI_i$
1	$10(10 * 0.9 - 100 * 0.1) = -10$
2	$5(10 * 0.99 - 100 * 0.01) - 5 \approx 40$
3	$3(10 * 0.999 - 100 * 0.001) - 7 \approx 23$
4	$2(10 * 0.9999 - 100 * 0.0001) - 8 \approx 12$
10	$1(10 * (1 - 10^{-10}) - 100 * 10^{-10})) - 9 \approx 1$

In the next section, we discuss how tasks' redundancy levels can be derived as a closed form formula. To achieve this, we use a continuous model to represent discrete tasks. Here we briefly present the basic idea. Consider a task T_i with only one copy scheduled to start at t_1 and to finish at t_2. Its failure probability is $F(t_2)$, because task T_i can be executed successfully only if the processor does not fail up to t_2. Its performance index PI_i is

$$c_i v_i - c_i(v_i + p_i)F(t_2), \tag{7}$$

where $c_i = t_2 - t_1$. In a continuous model, the performance index of the same task is represented as an integral from $t1$ to $t2$:

$$\int_{t_1}^{t_2} (v_i - (v_i + p_i)F(t))dt, \tag{8}$$

which is slightly larger than the one computed by (7) if $F(t)$ is a monotonically increasing function (which is true in general because of hardware aging process) and if it changes very

slowly. Typically, the reliability function $R(t)$ or, equivalently, the failure probability function $F(t)$ changes very slowly. Hence,

$$c_i F(t_2) \approx \int_{t_1}^{t_2} F(t)dt, \tag{9}$$

and the value computed by (7) is about the same as the one computed by (8). A similar argument applies for tasks with multiple copies. Once we have such a continuous model, instead of considering the redundancy level for each task, we can consider the redundancy level required at a particular time t. Later, we show that while this simplifies analysis, it does not result in any loss of accuracy in determining task redundancy levels.

3 Basic Task Configuration Strategy

In this section, we assume that all tasks have the same computation time c and the same v, p, and q. We discuss a way to derive task configuration function $u(t)$, where $u(t)$ represents the required redundancy level at time t, so as to optimize the performance index. It is important to note that all these assumptions are relaxed in the sections that follow. Specifically,

- In Section 4, a discrete task model is considered with tasks having different computation times and it is shown that the continuous model is a good approximation for the discrete model.
- In Section 5, we study the effects of converting $u^*(t)$, a real valued function, into integer values of $u(t)$ since, in practice, redundancy levels are integers.
- In Section 6, we present an approach to handle tasks having different reward rates and penalty rates, i.e., different values of v, p, and q for different tasks.

Because tasks' scheduling windows are assumed to be small, all redundant copies of task T_i will be scheduled to finish around the same time. Let t be the task finish time. Its performance index PI_i given in (4) becomes

$$PI_i = c(v - (v+p)F(t)^{u(t)}), \tag{10}$$

where $u_i = u(t)$, which is the redundancy level for T_i.

When c becomes very small, we can use a continuous model and use dt to represent c. Equation (10) then becomes

$$PI_i = (v - (v+p)F(t)^{u(t)})dt. \tag{11}$$

On average, the number of tasks that can be scheduled in the time interval $[t-dt, t]$ is $m/u(t)$. So the total performance index for the time interval $[t-dt, t]$ is

$$\frac{m}{u(t)}(v - (v+p)F(t)^{u(t)})dt. \tag{12}$$

Thus, performance index for the tasks that can be accommodated is

$$\int_0^L \frac{m}{u(t)}(v - (v+p)F(t)^{u(t)})dt, \tag{13}$$

and the penalty due to all rejected tasks is

$$q(C - \int_0^L \frac{m}{u(t)}dt), \tag{14}$$

where C is the total computation times of all tasks without counting their redundant copies,

$$C = \sum_{i=1}^{n} c_i. \tag{15}$$

Therefore, the total performance index is

$$PI = \int_0^L \frac{m}{u(t)}(v + q - (v+p)F(t)^{u(t)})dt - qC. \tag{16}$$

The task configuration problem is translated into a form of calculus of variations, and we want to find the best $u(t)$ which maximizes PI. Let us define

$$G(t, u(t)) = \frac{m}{u(t)}(v + q - (v+p)F(t)^{u(t)}). \tag{17}$$

Then, the maximum PI is determined by the following Euler equation according to the theory of the calculus of variations [2]:

$$\frac{\partial G}{\partial u} = 0, \tag{18}$$

with the boundary conditions of $1 \leq u(t) \leq m$. Equation (18) is the same as:

$$F(t)^{u(t)}(1 - lnF(t)^{u(t)}) = \frac{v+q}{v+p}, \tag{19}$$

where $0 < F(t) < 1$.

Define

$$\frac{v+q}{v+p} = \alpha. \tag{20}$$

Let us explain the physical meaning of α. Suppose we have a rejection penalty rate of $q = 0$. Assume that the failure penalty rate p is much larger than the reward rate v. The latter is a reasonable assumption for many fault-tolerant systems. Then α is roughly the ratio of the reward rate v and the failure penalty rate p. However, if $q > p$, then $\alpha > 1$ and it means that the penalty for rejecting tasks is too high, so we should accept more tasks and reduce the task redundancy. In this case, there exists no solution for Equation (19) and the best configuration strategy is $u^*(t) = 1$ by using one of the boundary conditions.

Table 3 shows the relations between v, p, q, and α. Case 1 corresponds to a relatively low failure penalty rate while Case 5 corresponds to a relatively high failure penalty rate.

Table 3: Relations between v, p, q, and α

Case	v	p	q	α
1	1	19	1	0.1
2	1	199	1	0.01
3	1	1999	1	0.001
4	1	19999	1	0.0001
5	1	199999	1	0.00001

If Equation (19) has a solution, it must satisfy the *iso-reliability principle*:

$$F(t)^{u(t)} = A_\alpha, \tag{21}$$

where A_α is a constant mainly dependent on α. It is easy to verify that this is indeed the solution if we substitute $F(t)^{u(t)}$ by a constant (A_α) in Equation (19) and observe that both sides of the equation become constant, although we must choose A_α properly. The iso-reliability principle is the most interesting feature of this task configuration problem. Its name was chosen to suggest the fact that A_α represents a level of tasks' failure probability which should be kept as a constant. Thus, the tasks' reliability is also a constant with respect to $(1 - A_\alpha)$.

Substituting (21) into (19) to determine A_α and we have

$$A_\alpha(1 - lnA_\alpha) = \alpha. \tag{22}$$

To search for the root, we may use a binary search algorithm such as the Bisection algorithm or a fast converging algorithm such as the Newton-Raphson algorithm [8].

We can then derive the optimal task configuration strategy $u^*(t)$ by rewriting (21),

$$u^*(t) = \frac{lnA_\alpha}{lnF(t)}, \tag{23}$$

where $0 < F(t) < 1$ and $1 \leq u^*(t) \leq m$.

In practice, we cannot control the failure function $F(t)$, but we can control the function $u(t)$ during the task configuration stage. We make two observations that are of significance from a practical viewpoint.

Observation 1: $u^*(t)$ changes slower than the failure function $F(t)$, because, in Equation (23), $u^*(t)$ is inversely proportional to $lnF(t)$, where $0 < F(t) < 1$.

In practice, $F(t)$ is likely to be a very slow function of t, so $u^*(t)$ is likely to be an even slower function of t.

Observation 2: If $F(t)$ is a monotonically increasing function, then $u^*(t)$ is a non-decreasing function, where $0 < F(t) < 1$ and $1 \leq u^*(t) \leq m$.

To see that the observation is right, we show that $(u^*(t))' > 0$. If $F(t)$ is a monotonically increasing function and $0 < F(t) < 1$, then $F'(t) > 0$, $lnF(t) < 0$ because $F(t) < 1$, and $lnA_\alpha < 0$ because

$$0 < F(t)^{u^*(t)} = A_\alpha < 1.$$

From Equation (23),

$$(u^*(t))' = (\frac{lnA_\alpha}{lnF(t)})' = lnA_\alpha \frac{(-1)}{(lnF(t))^2} \frac{F'(t)}{F(t)} > 0.$$

$F(t)$ is likely to be a monotonically increasing function because of the hardware aging process. Therefore, we can expect $u^*(t)$ to increase with time. This is demonstrated in the following example.

Example 2: We plot $u^*(t)$ in Figures 1, 2, and 3 for three different $L's$ by assuming that $m = 10$, $F(t) = 1 - e^{-\lambda t}$, $\lambda = 0.0001$. In these figures, each curve corresponds to a different α. Here are some conclusions we can derive from these figures:

- u^* increases with t,
- u^* increases slowly when α becomes larger,
- u^* is flat for large α when t is small, e.g., $\alpha = 0.1$, because u^* hits the lower bound 1. This means that, when the failure penalty rate is low, we do not need any redundancy for tasks.

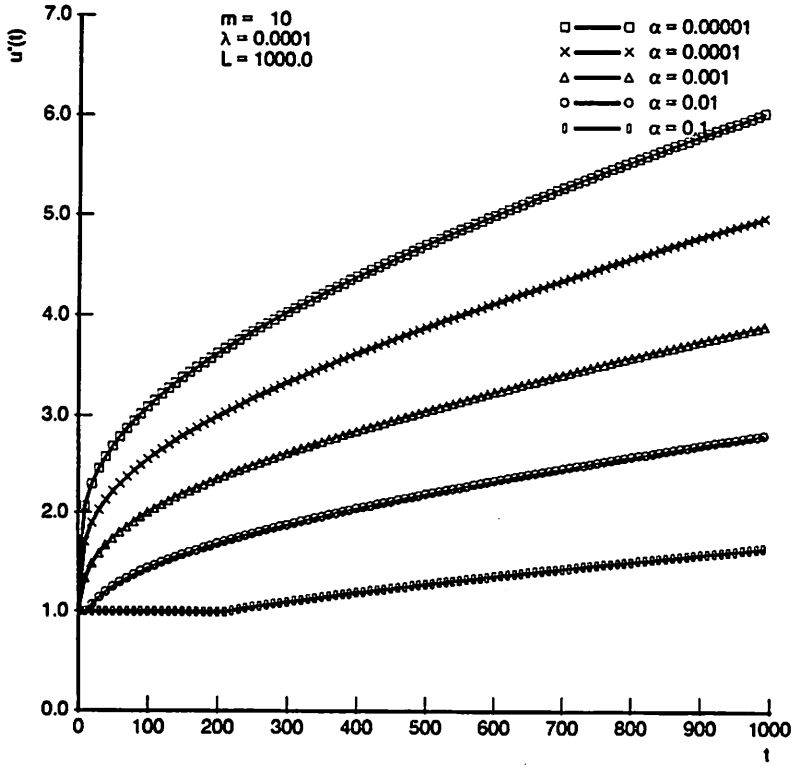

Figure 1: Optimal configuration strategy $u^*(t)$ with $L = 1000$

In this section, we have built the basic task configuration strategy, based on a continuous model assuming tasks have the same computation time and reward/penalty rates v, p, and q. Specifically, we derived the optimal task configuration strategy $u^*(t)$ in a simple closed form:

$$u^*(t) = \frac{B}{lnF(t)}$$

where B is a constant, $B = lnA_\alpha$, and $F(t)$ is the failure function. In the following sections we now relax these assumptions.

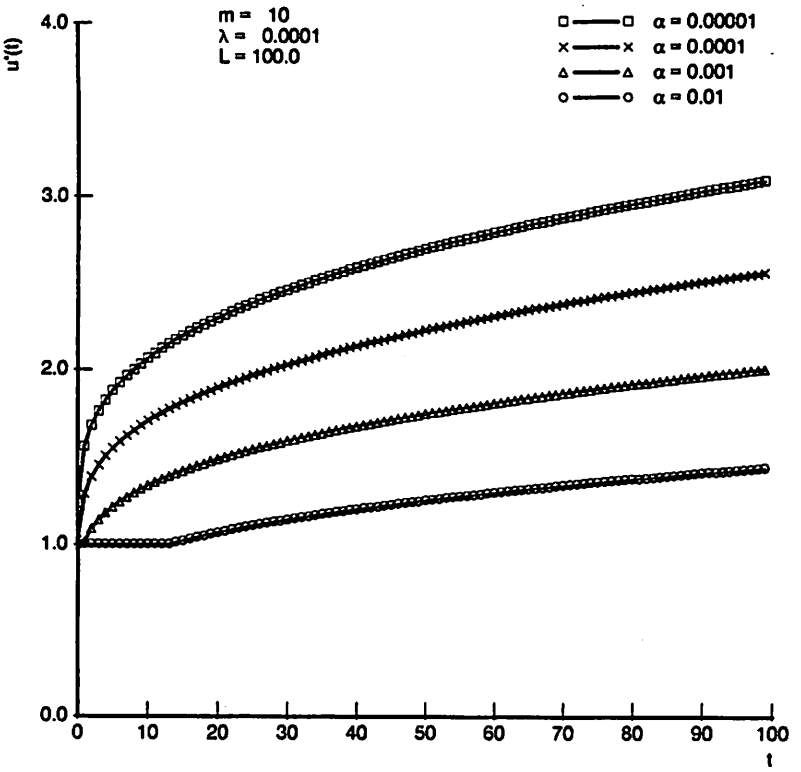

Figure 2: Optimal configuration strategy $u^*(t)$ with $L = 100$

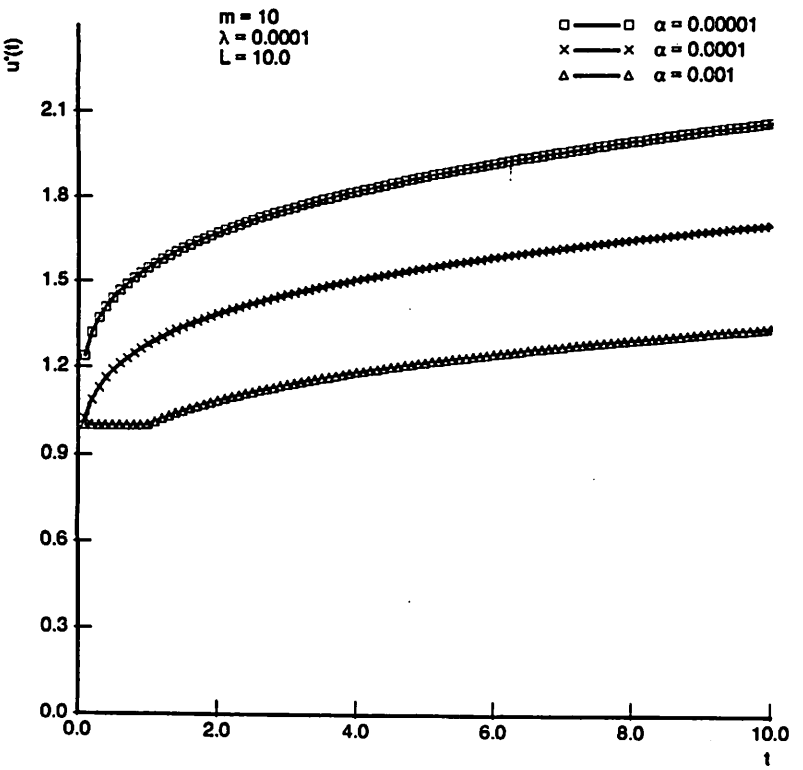

Figure 3: Optimal configuration strategy $u^*(t)$ with $L = 10$

4 Discrete Model

In this section, we extend the continuous model to a discrete model. This relaxes the assumption that task computations are infinitely small as assumed in the last section. We discuss two cases:

1. Tasks have the same computation time c.
2. Tasks have different computation times.

Tasks are assumed to have the same v, p, and q. This is relaxed in Section 6.

We consider case 1 first. Let L be divided equally into k equal sized intervals of size c:

$$[t_0, t_1], [t_1, t_2], \cdots\cdots, [t_{k-1}, t_k],$$

with $t_0 = 0$ and $t_k = L$. Let $u(t_i)$ be the average redundancy in the interval $[t_{i-1}, t_i]$. Then, Equation (16) becomes

$$PI = -qC + \sum_{i=1}^{k} \frac{m}{u(t_i)}(v + q - (v + p)F(t_i)^{u(t_i)})c. \tag{24}$$

Using the same analysis method, we can derive the optimal configuration strategy as

$$u^*(t_i) = \frac{lnA_\alpha}{lnF(t_i)} \tag{25}$$

where $0 < F(t_i) < 1$, $i = 1, 2, \cdots, k$, and A_α is the same as defined in (20). Table 4 shows the relations between c versus PI under three different $L's$, where PI is computed with the optimal configuration strategy, $u^*(t_i)$, for $1 \leq i \leq k$. We assume that $m = 10$, $\alpha = (r+q)/(r+p) = 0.0001$, and $\lambda = 0.0001$. The table shows that, for different values of c, the performance index PI is about the same for a particular L, especially when L is large compared to c. This implies that the continuous model which assumed very small values for c accurately represents the discrete model with respect to the performance index. So

$$\int_0^L \frac{m}{u(t)}(v + q - (v + p)F(t)^{u(t)})dt \approx \sum_{i=1}^{k} \frac{m}{u(t_i)}(v + q - (v + p)F(t_i)^{u(t_i)})c. \tag{26}$$

For case 2, where tasks have different computation times, it is difficult to to extend Equation (16) directly by using the similar method as in the case 1, because tasks may have different finish times in any subinterval. But, notice that in Table 4, for a given L, PI is almost the same for tasks with different computation times. So, given the approximation in (26), we can use the continuous model to approximate this case also to obtain the performance index.

Table 4: Relations between c and PI

c	$PI\|_{L=1000}$	$PI\|_{L=100}$	$PI\|_{L=10}$
10.0	2583.36988	423.31917	54.15487
1.0	2605.02754	437.06966	60.36285
0.1	2607.82060	439.25115	61.73948
0.01	2608.36467	439.49504	61.93850
0.001	2608.40129	439.55622	61.95905

5 Converting $u^*(t)$ into integer values of $u(t)$

The optimal configuration strategy $u^*(t)$ in (23) is a real valued function. In practice, tasks' redundancies are integers. In this section, we show that the optimal task configuration strategy $u^*(t)$ can be approximated by an integer function with a very small loss with respect to the performance index PI.

We compare the performance index using $u^*(t)$ with the performance index using the following integer functions which approximate $u^*(t)$:

- $u\_ceil(t)$ — the integer equal to or greater than $u^*(t)$;
- $u\_rint(t)$ — rounding $u^*(t)$ to an integer;
- $u\_int(t)$ — choosing one of the two neighboring integers of $u^*(t)$ which gives the better performance.

Let $PI(u(t))$ be the performance index using strategy $u(t)$. Comparing the performance index PI using $u^*(t)$ to the performance index PI using $u\_ceil(t)$, $u\_rint(t)$, and $u\_int(t)$ respectively, it is not difficult to see that

$$PI(u^*(t)) \geq PI(u\_int(t)) \geq \{PI(u\_ceil), PI(u\_rint)\}.$$

In Figures 4, 5, and 6, we plot $u^*(t)$ and these three functions with $L = 100$. Table 5 lists the ratios of the performance indices based on integer functions and the performance index using the optimal configuration strategy $u^*(t)$, for three different $L's$. Here, we assume that $m = 10$, $\alpha = (v+q)/(v+p) = 0.0001$, $\lambda = 0.0001$, and $c = 1$.

From Table 5, we conclude that $u\_int(t)$ is the best candidate to represent $u^*(t)$ with respect to the performance index PI. Also Figure 6 shows that $u\_int(t)$ has the redundancy values 2 and 3 in relatively large time windows. This validates the assumption we made earlier that the optimal task redundancy is highly likely to be a constant within tasks' scheduling windows.

Table 5: Ratios of the performance indices using integer functions and $PI(u^*)$

L	$PI(u\_ceil)/PI(u^*)$	$PI(u\_rint)/PI(u^*)$	$PI(u\_int)/PI(u^*)$
1000	0.929	0.921	0.960
100	0.859	0.716	0.906
10	0.825	0.080	0.825

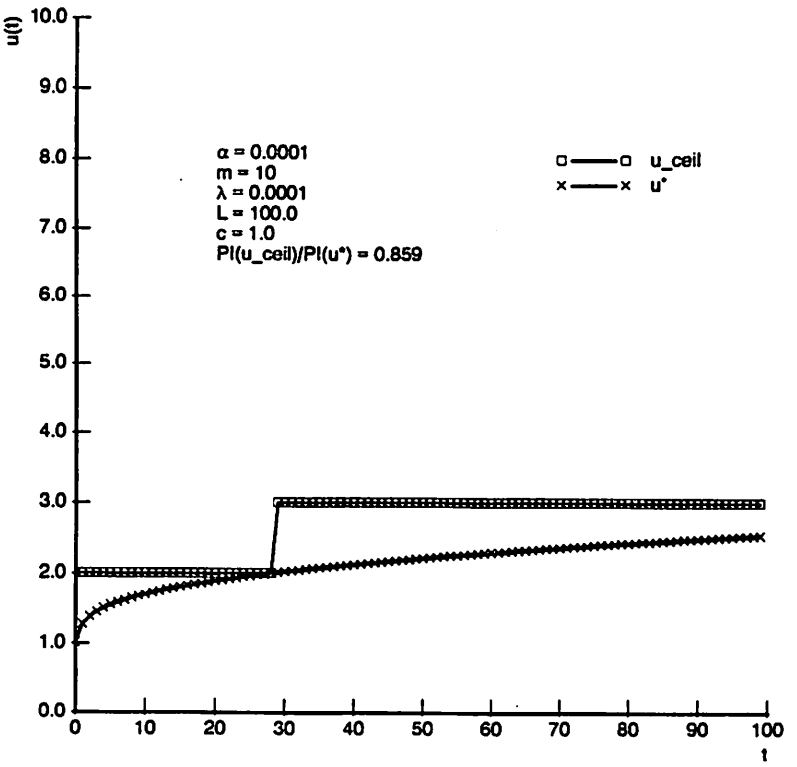

Figure 4: u^* versus u_ceil with $L = 100$

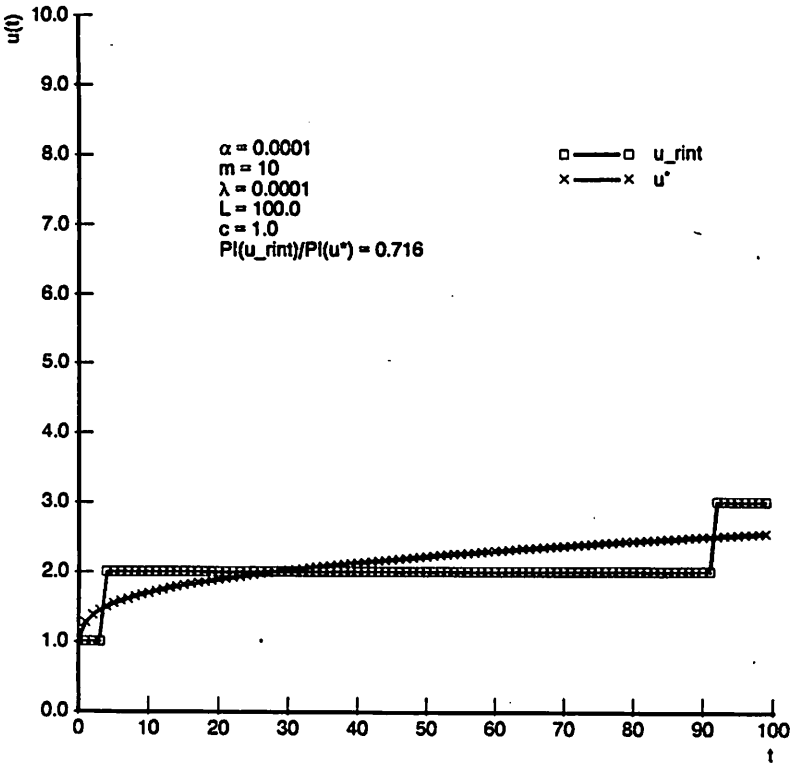

Figure 5: u^* versus u_rint with $L = 100$

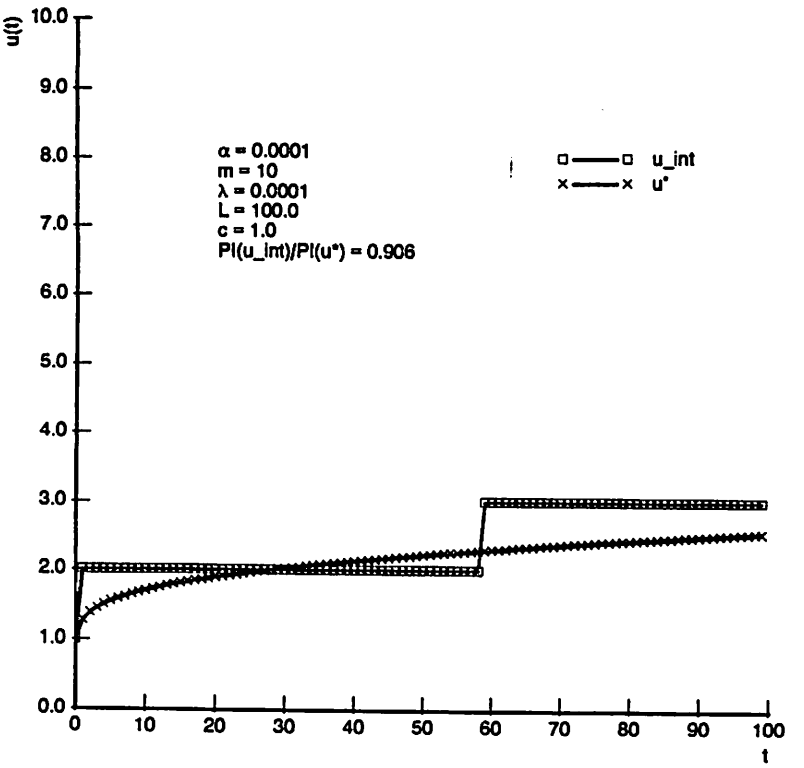

Figure 6: u^* versus u_int with $L = 100$

6 Configuring Tasks with Different Reward/Penalty Parameters

In this section, we extend the basic task configuration model to allow tasks with different v, p, and q. Let $v(t)$, $p(t)$, and $q(t)$ be the average reward rate and the average penalty rates at t, computed from tasks whose scheduling windows include t. The total performance index becomes

$$PI = \int_0^L \{\frac{m}{u(t)}[v(t) + q(t) - (v(t) + p(t))F(t)^{u(t)}] - \frac{q(t)C}{L}\}dt. \tag{27}$$

Define

$$G(t, u(t)) = \frac{m}{u(t)}[v(t) + q(t) - (v(t) + p(t))F(t)^{u(t)}] - \frac{q(t)C}{L}. \tag{28}$$

Then, the maximum of PI is again determined by Euler equation:

$$\frac{\partial G}{\partial u} = 0, \tag{29}$$

with the boundary conditions of $1 \leq u(t) \leq m$. Equation (29) is the same as:

$$F(t)^{u(t)}(1 - lnF(t)^{u(t)}) = \alpha(t), \tag{30}$$

where $0 < F(t) < 1$, $1 \leq u(t) \leq m$, and

$$\alpha(t) = \frac{v(t) + q(t)}{v(t) + p(t)}. \tag{31}$$

The solution for (30) is the optimal configuration strategy $u^*(t)$, which must satisfy

$$F(t)^{u^*(t)} = A_\alpha(t), \tag{32}$$

where $0 < F(t) < 1$, $1 \leq u^*(t) \leq m$, and $A_\alpha(t)$ depends on $\alpha(t)$. Rewriting the above equation, we have

$$u^*(t) = \frac{lnA_\alpha(t)}{lnF(t)}. \tag{33}$$

To compute function $A_\alpha(t)$, we substitute (32) into (30):

$$A_\alpha(t)(1 - lnA_\alpha(t)) = \alpha(t). \tag{34}$$

Thus, given t, $0 \leq t \leq L$, we can compute $\alpha(t)$. From $\alpha(t)$, we can determine the corresponding $A_\alpha(t)$, and then we can determine $u^*(t)$.

7 Applying the Results

In this section, we discuss how to apply this task configuration theory in a real-time system. The basic idea is to derive the optimal task redundancy function $u^*(t)$ first. Note that $u^*(t)$ is determined only by the processor reliability function and task reward/penalty rates. $u^*(t)$ is then approximated by a corresponding integer function $u\_int(t)$. Finally, by using $u\_int(t)$, we can determine how many copies of each task can be need to be scheduled.

If tasks to be configured have the same reward and penalty rates, e.g., the same v, p, and q, the task configuration procedure becomes relatively easy, even if their computation times are different. A_α is computed by solving Equation (22) and $F(t)$ is determined from the failure properties of the hardware. We can then compute $u^*(t)$ using Equation (23). Next, we use $u\_int(t)$ to approximate $u^*(t)$ as presented in Section 5. In general, $u\_int(t)$ has a shape similar to the one plotted in Figure 6, which is a step function. Thus, the computation for $u\_int(t)$ can be easily speeded up, by just computing each turning point of the step function $u\_int(t)$. For example, in Figure 6, the redundancy levels are 1 for [0, 1), 2 for [1, 59), and 3 for [59, 100], and the turning points occur at 1 and 59. Also, we may compute $u\_int(t)$ off-line to form a table for on-line use. After $u\_int(t)$ is derived, tasks are assigned the redundancy levels given by $u\_int(t)$ in the following way. If a task's scheduling window covers two different values of $u\_int(t)$, we assign the task the higher redundancy level. Otherwise, we assign the task with a redundancy level determined by $u\_int(t)$. Note that this is only an approximation because the optimal redundancy level required by a task is determined by its scheduled finished time in the final schedule.

We can then schedule the tasks. First, consider a scheduling algorithm that contains logic to shed tasks. In this case, the task set determined by the configuration phase is directly handed to the scheduling algorithm. We can apply a heuristic-based [11] or a bin-packing-based scheduling algorithm [4]. In either case, any remaining tasks are rejected after all available system resources are consumed. Second, if a scheduling algorithm does not contain logic for shedding tasks during scheduling, then we may shed some tasks before the task set is handed to the scheduling algorithm. We do this to avoid repeated failures of the scheduling algorithm in finding a feasible schedule. Failures are mainly caused by the following factors:

- The task set for the scheduling algorithm has an overload in a time interval $[t_x, t_y]$ (for $t_y \leq L$), such that the sum of the computation times of all tasks having deadlines within this interval is greater than $m \cdot (t_y - t_x)$, where m is the number of processors available.

- The heuristic scheduling algorithm may fail to find a feasible schedule even if the task set is feasible, because the heuristic scheduling algorithm is only an approximation to the optimal algorithm which always find the feasible schedule if it exists.

- Tasks may have more complex constraints, e.g., additional resource requirements, which may reduce the system utilization because of the resource contentions among tasks.

Possible solutions to avoid these failures are (1) to avoid overloads in all sub-intervals and (2) to reduce task workload further because a portion of the system utilization will be wasted because of the resource contentions among tasks.

If tasks to be configured have different reward and penalty rates, the task configuration procedure becomes a bit more complicated. Here is a high level description of one way to solve the problem. We divide L into K equal intervals of size Δ. The value of Δ will depend on how closely we would like the redundancy levels to reflect optimal values. For example, assuming that $\bar{c}$ is the average computation time for the tasks, we set Δ to be $\bar{c}/2$. For each t, where $t = i\Delta$, $i = 1 \cdots K$, we do the following: We compute $\alpha(t)$ based on Equation (31), with $v(t)$, $p(t)$, and $q(t)$ being the average reward rate and the average penalty rate at t, computed from tasks whose scheduling windows cover t. From $\alpha(t)$, we can derive $A_\alpha(t)$ either by solving Equation (34) or creating a table offline and just doing a table lookup. After knowing $A_\alpha(t)$ and $F(t)$, we compute $u^*(t)$ based on Equation (33). Next, we use $u\_int(t)$ to approximate $u^*(t)$ as presented in Section 5. Knowing $u\_int(t)$, tasks are then assigned the redundancy with the values specified by $u\_int(t)$. Finally, we can start to schedule tasks. Many issues related to task scheduling discussed above also apply here.

Note that, in both cases, the values of $u\_int(t)$ represent the lower bound on task redundancy to maximize performance without being jeopardized by too high redundancy levels. Therefore, some tasks could be assigned higher redundancy levels than the ones specified by $u\_int(t)$, if there are not enough tasks to fill the processor resources in a given time interval. This will improve reliability without affecting the schedulability of tasks. Also, how well all this works in practice must still be determined by simulations or actual system implementations.

8 Conclusions

In many situations, there are tradeoffs between task performance and task reliability. Increasing the redundancy level for a task decreases the probability that the task will fail after being

accepted while also decreasing the number of tasks that get scheduled in the first place. This implies tradeoffs involving the fault tolerance of the system, the rewards provided by guaranteed tasks that complete successfully, and the penalties due to tasks that fail after being guaranteed or that fail to be guaranteed. In this context, we presented an approach to mathematically determine the replication factor for a given set of tasks with the goal of maximizing the total performance index, which is a performance-related reliability measurement. Our analysis shows that the basic continuous task model very closely approximates the discrete models, and the optimal task configuration function $u^*(t)$ can be substituted by an integer function with very minor effects on the total performance index. Also, we showed how our basic model can be extended to handle tasks with different reward/penalty parameters and computation times.

If the set of tasks is static, then the resulting analysis shows what the redundancy of each task should be for such a static system. However, given our interest in dynamic real-time systems, when applied to a set of tasks that exists at that point in time, it tells us what the redundancy level should be to maximize the performance index at that point in time. This may or may not produce the best performance from the viewpoint of the whole system and the complete mission.

In the future we propose to continue from the results developed here in the following ways:

- we will expand the analysis and the performance index to apply to the whole system and to a complete mission rather than at a single point in time,
- so far we have focused on hardware faults and considered task replication as the only fault-tolerance approach; given that a number of fault-tolerance approaches can coexist even within a single system, we will look at supporting multiple approaches simulataneously,
- interactions between the configuration phase, during which task redundancy levels are determined, and the scheduling phase, when these redundant copies are scheduled, need to be further explored,
- the task characteristics handled by the approach need to be expanded to include other constraints such as resource constraints and precedence constraints as well as additional types of timing constraints such as periodicity.

References

[1] Beaudry, M. D. Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, pages 540–547, June 1978.

[2] Bolza, O. *Lectures on the Calculus of Variations.* Chelsea Publishing Company, 1960.

[3] Calabro, S. R. *Reliability Principles and Practices.* McGraw-Hill, New York, 1962.

[4] Garey, M. R., Graham, R. L., Johnson, D. S., and Yao, A. C.-C. Resource constrained scheduling as generalized bin packing. *J. Combinatorial Theory Ser. A*, 21:257–298, 1976.

[5] Krishna, C.M. and Shin, K.G. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Transactions on Computers*, C-35(5):448-55, May 1986.

[6] Lee, Y.-H. and Shin, K. G. Optimal reconfiguration strategy for a degradable multimodule computing system. *Journal of the ACM*, pages 326–348, April 1987.

[7] Liestman, A.L. and Campbell, R.H. A Fault Tolerant Scheduling Problem. *IEEE Transactions on Software Engineering*, SE-12(11):1089-95, November 1986.

[8] Mathews, J. *Numerical Methods for Mathematics, Science, and Engineering.* Prentice Hall, 1992.

[9] Meyer, J. F. On evaluating performability of degradable computing systems. *IEEE Trans. on Computers*, C-29:720–31, 1980.

[10] Muppala, J., Woolet, S. and Trivedi, K. Real-Time Systems Performance in the Presence of Failures. *IEEE Computer* 24(5), May 1991.

[11] Zhao, W. and Ramamritham, K. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *The Journal of System and Software*, 7:195–207, 1987.