On Scheduling Tasks with a Quick Recovery from Failure

C. M. KRISHNA, MEMBER, IEEE, AND KANG G. SHIN, SENIOR MEMBER, IEEE

Abstract — **Multiprocessors used in life-critical real-time systems must recover quickly from failure. Part of this recovery consists of switching to a new task schedule that ensures that hard deadlines for critical tasks continue to be met. We present a dynamic programming algorithm that ensures that backup, or contingency, schedules can be efficiently embedded within the original, "primary" schedule to ensure that hard deadlines continue to be met in the face of up to a given maximum number of processor failures. Several illustrative examples are included.**

Index Terms — **Fault-tolerant and real-time multiprocessors, hard deadlines, primary and contingency schedules, primary and ghost clones, notification times.**

I. INTRODUCTION

DIGITAL computers have become an essential part of real-time control systems. Such computers are called *real-time (control) computers* (SIFT [1] is a good example). Real-time computers are required to be fast and reliable and usually execute (control) tasks periodically. There are usually several concurrently executing tasks on a real-time computer, which communicate with each other at the beginning (for input) and the end (for output) of execution, but not during execution. This, along with the speed and reliability requirement, naturally leads to a popular structure of real-time computers: a multiprocessor with each processor having its own private memory.

It is important to allocate and schedule real-time tasks on the processors of such a multiprocessor so that all real-time requirements (hard deadlines[1]) may be met even in the presence of processor failures. Assume that we have an algorithm P which allocates and schedules the tasks on the processors quasioptimally so as to meet all task deadlines. Our goal in this paper is to devise a technique whereby algorithm P can be modified to generate fault-tolerant schedules which guarantee that the real-time computer can tolerate a given number of processor failures, and still deliver acceptable (albeit degraded) performance. We call this modified procedure algorithm Q. We will show that if algorithm P is optimal according to criterion given below, so is algorithm Q in the sense that if no processor failures occur, the schedules developed by Q are the best possible under the constraints described below.

The optimization criterion uses *cost functions* developed in [2], [3]. Real-time control computers are in the feedback loop of the *controlled process,* and so their task response time translates into feedback delay, and tends to degrade the quality of control provided. This can be quantified for each task by $f_i(C_i)$ where f_i is the cost function associated with task i and C_i is the associated response time. The cost function is a monotonically nondecreasing[2] function of the task response time and can be derived from the performance index used in control theory to describe the system's objectives. See [3] for a detailed derivation.

A schedule defines the task response times, and by adding up the costs accrued by the various tasks, a cost can be associated with the schedule. Also, hard deadlines exist for the tasks since the system must operate in real time. A schedule is said to be optimal if it has minimal cost under the constraint of meeting all deadlines.

Tasks are run periodically, and each release of a task is called a *version*. For fault tolerance, multiple copies of versions are executed in parallel, with voting or some other mechanism for deriving the output. When a processor fails, the versions scheduled on it and not already started must be replaced to maintain adequate fault tolerance. Such a contingency cannot entirely be handled in real time because the allocation/scheduling problem is time consuming. A predetermined course of actions has to be designed which can be executed sufficiently quickly. We solve this problem by having contingency schedules which can be quickly manipulated upon processor failure. A constraint we impose, which conserves memory, is that *only one* contingency schedule is needed per processor.

We now introduce some terminology. The multiple copies of versions are called *clones*. There are two types of clones: *primary* and *ghost*. A *primary clone* is executed in the normal course of things. A *ghost clone* is a backup copy which lies dormant until it is activated to take the place of a corresponding primary or previously activated ghost whose processor has failed. Each ghost clone of version j has a *notification time*, ν_j, associated with it. After this time, the

Manuscript received January 9, 1985; revised July 30, 1985 and November 5, 1985. This work was supported in part by NASA Grant NAG-1-296 and by the Office of Naval Research under Contract N00014-85-K-0122. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

C. M. Krishna was with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. He is now with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003.

K. G. Shin is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8608215.

[1]Deadlines and hard deadlines will be used interchangeably throughout the paper.

[2]In this paper, we will assume later that the cost function is monotonically increasing in the response time.

system is blind to the need to activate it. The following example might help establish this concept.

Suppose there are three primary clones of version α. Call them P_{α_1}, P_{α_2}, and P_{α_3}. Let the processors on which they are supposed to run be β_1, β_2, and β_3, respectively. Let there be only one ghost clone of that version: call it g_α. If the deadline of g_α is $t_{d\alpha}$, and its run time r_α, g_α must begin executing by $\tau_\alpha = t_{d\alpha} - r_\alpha$ at the latest if it is to meet its deadline. Let the system monitor the status of $P_{\alpha 1}$, $P_{\alpha 2}$, and $P_{\alpha 3}$ at instants $T_1 = \{t^1_{\alpha 1}, t^2_{\alpha 1}, \cdots,\}$, $T_2 = \{t^1_{\alpha 2}, t^2_{\alpha 2}, \cdots,\}$, and $T_3 = \{t^1_{\alpha 3}, t^2_{\alpha 3}, \cdots,\}$ respectively. Such monitoring may be done either implicitly (by voting at version completion instants) or explicitly (by executing test routines at given moments). The means of monitoring are irrelevant to this paper; it matters only that because the loading is assumed to be deterministic, the sets T_1, T_2, and T_3 are available. Then, the notification time of ghost g_α is $\nu_\alpha = \max\{\xi \mid \xi \leq \tau_\alpha, \xi \in T_1 \cup T_2 \cup T_3\}$.

This paper is organized as follows. The next section contains the problem statement and some background material. In Section III we present the main result with its related proofs. Section IV provides an illustrative example, and the paper concludes with a discussion in Section V. A list of the more commonly used notation forms the Appendix.

II. Problem Statement and Assumptions

Assume that the given algorithm $\boldsymbol{P}$ can be split into two subalgorithms: $\boldsymbol{P}_1$ which finds the optimal allocation of tasks to processors and also the optimal schedule, by calling $\boldsymbol{P}_2$ which is an optimum scheduler for uniprocessor systems.

For reliability reasons the real-time computer runs a certain number n_i, of clones of version i in parallel, i.e., $n_i \geq 1$. It is said to *sustain* up to N_{sust} failures if, despite the failure of up to N_{sust} processors in any sequence, the system is able — after the failures have been identified and the system has been reconfigured — to schedule tasks so that n_i clones of version i for all i (including some activated ghosts) can be executed in parallel without deadlines being missed.

Our problem is to develop an algorithm $\boldsymbol{Q}$ that can be used in conjunction with $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ to obtain an optimal schedule when enough ghosts are incorporated into the schedule to sustain up to N_{sust} processor failures, and also obtain contingency schedules that can be invoked when the system is notified that a given ghost is to be activated. This must be done subject to the constraints a) that only two (i.e., primary and contingency) schedules per processor need to be stored, and b) that no major computation is required to activate a ghost.

Both these constraints stem from practical considerations: the need to conserve memory and (on-line) time. If unlimited memory or unlimited time were available, the problem would become trivial: one would obtain a separate schedule for each possible sequence of processor failures, or one would recalculate an optimal schedule each time a processor failed. Neither course of action is open to us for obvious practical reasons.

The schedule to be derived is *locally-preemptive,* i.e., tasks resident on the same processor are allowed to preempt one another, but tasks resident on separate processors are not. The locally-preemptive regime has been assumed in order to avoid the possibility of causing excessive congestion on the interconnection structure within the multiprocessor. It also removes the need to allow for queueing delays as part of the schedule.

Some assumptions follow.

A1) The interrepair interval, L, is much smaller than the mean time between successive processor failures.

A2) The release time, deadline, and computational demand of each version are deterministic. So is L.

A3) The number of processors available is sufficient to satisfy all deadlines and reliability requirements.

A1) is reasonable, considering that mean times between processor failures are nowadays in the 1000–10 000-hour range. A2 follows in part from the tasks being released periodically. The run time will be a random variable due to data-dependent conditional branches in the code, but worst-case numbers can be used to safely represent a deterministic run time. Without sufficient processors, we clearly cannot proceed with a scheduling operation, which justifies A3).

As noted before, in order to sustain the allowed-for number of processor failures, ghost clones of the various *critical* task[3] are scheduled. Now, ghost clones are not activated if there is no processor failure, and we have assumed in A1) that failures are not common. So, the only constraint upon the scheduling of the ghosts is that their hard deadlines must be met: we are not unduly concerned with the efficiency of a schedule that has one or more ghosts activated. Although the ghosts are initially passive, and do not in that state require any processor time, they represent a latent demand for processor time that must be allowed for. For this reason, they affect the scheduling of the primary clones (since they share processors with them), and tend to lower the efficiency of the primary schedules. This can best be illustrated through the following example. Consider a processor to which are allocated primary clones 1 and 2, and ghost clone 3. Let their release times be $\text{rel}_1 = 1$, $\text{rel}_2 = 2$, and $\text{rel}_3 = 3$, their deadlines be $t_{d1} = 11$, $t_{d2} = 12$, and $t_{d3} = 7$, and their cost functions be $f_1(t) = t - 1\ t \geq 1$, $f_2(t) = (t-2)^2\ t \geq 2$, and $f_3(t) = 0^4$ for arguments t less than the corresponding hard deadlines, and infinity above them. Let their run times be $r_1 = 6$, $r_2 = 0.5$, and $r_3 = 4$, respectively. Without the ghost, the optimal primary schedule is as shown in Fig. 1(a). However, although the ghost is not initially activated, does not even form a part of the primary schedule, and if not activated needs no processor time, we must always allow for the possibility of its being activated, and then occupying the processor from $t = 3$ to $t = 7$. To do this, the only acceptable primary schedule is that shown in Fig. 1(b) (cost = 25.25), which is much more expensive than the one in Fig. 1(a) (cost = 9.75).

Since, in the overwhelming majority of cases, the primary schedules are the only ones that will be executed, their effi-

[3]A task is said to be critical if it has a hard deadline, which if violated, can lead to catastrophe.

[4]The ghost cost function f_3 is consistent with the definition of the ghost cost functions in general, as in the next paragraph.

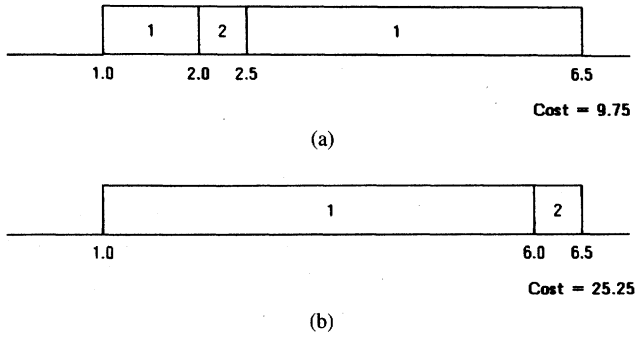


Fig. 1. Effect of ghosts on primary schedules: an example. (a) Optimal primary schedule without influence of ghost 3. (b) Optimal primary schedule showing influence of ghost 3.

ciency is of prime concern. Any fault-tolerant scheduling with ghosts must, therefore, strive to minimize the increased cost of the primary schedule that can be attributed to the existence of the ghosts, while ensuring, if the ghosts ever have to be activated, that their hard deadlines are met. For this reason, we define the ghost cost functions as identically zero for values of the response time less than their associated hard deadlines. Since the penalty to be paid for an activated ghost's missing its deadline is as great as that for a primary clone, the ghost cost function for response times greater than the deadline will continue to be infinity. When this ghost cost function is used as input to an algorithm that performs optimal scheduling, the result will be to degrade the efficiency of the primary schedules by as little as is consistent with the need to allow the ghost clones to meet their deadlines. We also assume that the cost functions of the primary clones for response times less than their hard deadlines are monotonically increasing, not just monotonically nondecreasing as defined in [2], [3]. Given that the set of real numbers is everywhere dense, this inflicts no practical inaccuracy on our calculations.

The number of ghosts per critical task, and the scheduling rules that they follow, are developed in the next section.

III. Main Result

We begin by stating certain conditions that the ghosts should satisfy.

C1) Each version must have ghost clones scheduled on N_{sust} processors, and a ghost and a primary of the same version may not be scheduled on the same processor.

C2) Ghosts are *conditionally transparent*. That is to say, a) two ghost clones may overlap in the schedule if and only if none of their corresponding primary clones are scheduled on the same processor, and b) primary clones may overlap ghosts on the same processor, and may indeed be scheduled without regard for the ghosts, so long as the following condition is met: for every instant t in the schedule, if all ghosts whose notification time is greater than t are activated, the schedule for beyond t can be reordered to ensure that all ghosts and all primaries allocated to that processor meet their hard deadlines, provided only that when some ghosts mutually overlap, at most one of them is activated.

An illustration of C2 is provided in Fig. 2. In this example, the notification time of a ghost is its release time. (Note that the notification time should be less than or equal to its corresponding release time.) In Fig. 2(a), we see an overlap of primary and ghostly clones that show condition C2: when the system is notified that the ghost is to be activated, the schedule for beyond that time can be reordered to ensure that all hard deadlines are met. Fig. 2(b) shows an overlap that does not meet condition C2: in the event that the ghost is activated, either ghost i or primary j will miss its deadline.

Lemma 1: C1 and C2 are necessary and sufficient conditions for up to N_{sust} processor failures to be sustained.

Proof: The necessity of C1 is obvious. Consider ghost clones g_1 and g_2 of two versions V_1 and V_2 which overlap in the schedule of some processor p in such a way that if both of them are activated, at least one of them will miss its deadline. Let $\rho(V_1)$ and $\rho(V_2)$ be the sets of processors allocated to run the primary clones of V_1 and V_2, respectively. If $\rho(V_1) \cap \rho(V_2) = A \neq \emptyset$, there exists a processor $q \in A$. If $|\rho(V_1) \cup \rho(V_2)| \geq N_{sust}$ and there are exactly N_{sust} ghost clones for each version, the failure of processor q together with $(N_{sust} - 1)$ processors in the set $\rho(V_1) \cup \rho(V_2) - \{q\}$ results in a situation in which it is impossible to replace all the failed clones by ghosts. If $|\rho(V_1) \cup \rho(V_2)| < N_{sust}$ and there are exactly N_{sust} ghost clones for each version, let $\gamma(V_1)$ and $\gamma(V_2)$ be the processors carrying the ghost clones of V_1 and V_2, respectively. By definition, $p \in \gamma(V_1) \cup \gamma(V_2)$. If all the processors in $\rho(V_1) \cup \rho(V_2)$ and $N_{sust} - |\rho(V_1) \cup \rho(V_2)|$ of the processors in $\gamma(V_1) \cup \gamma(V_2) - \{p\}$ fail, then again it will be impossible to activate the requisite number of ghosts. This proves the necessity of C2(a); that of C2(b) may be similarly shown.

The proof of sufficiency is similar, and is left to the reader. □

We can now turn to obtaining algorithm $\boldsymbol{Q}$. We begin by stating some additional definitions and notation pertinent to algorithm $\boldsymbol{Q}$. A *hole* arising out of an allocation of tasks to a particular processor is defined as an interval in which it is impossible to schedule any of the allocated primary clones due to their release times and deadlines. A point in the schedule representing time t is in a hole if and only if i) the deadlines of all primary clones released prior to t are less than t, and ii) no primary clone is released at t. Clearly, a hole depends *only* on the allocation of primary clones and not on their specific positioning in a schedule. In other words, all feasible schedules for a given allocation of primaries have the same holes. In order to avoid confusion, we emphasize that not all blank spaces in a schedule need be holes: under our definition, only zones in which it is impossible to schedule any allocated primary clone are holes.

Algorithm $\boldsymbol{P}_2$ has, as input, the deadlines of the clones, and the remaining run times. *Running* $\boldsymbol{P}_2(A)$ means that $\boldsymbol{P}_2$ is invoked with the set of clones A allocated to the processor. By *initializing* $\boldsymbol{P}_2$ at some time τ we mean that $\boldsymbol{P}_2$ generates a schedule starting at τ, using the inputs mentioned above. We denote the set of ghosts that have been allocated by $\boldsymbol{P}_1$ to processor i by θ_i. By $ghost_i(j)$ we mean the jth ghost (in order of notification time) allocated to processor i. $Ghost_i(0)$ is a

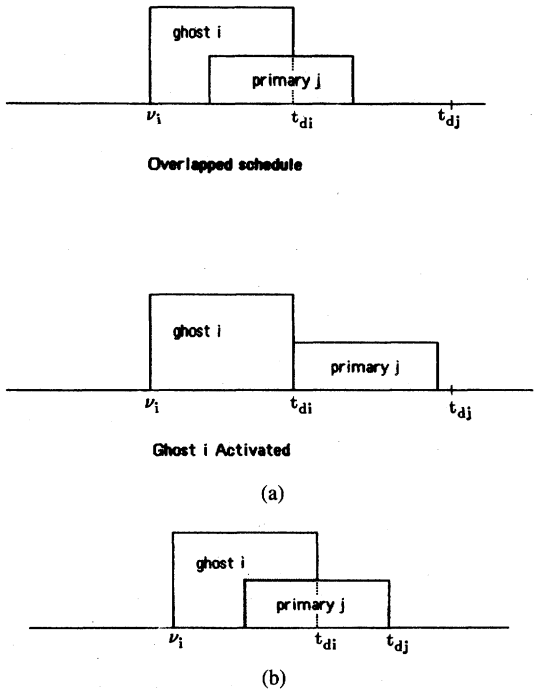


Fig. 2. Illustration of condition C2. (a) permissible overlap. (b) Impermissible overlap.

dummy ghost, requiring 0 time, which is introduced for notational convenience.

Algorithm $\boldsymbol{Q}$ adjusts or *transforms* the hard deadlines to accommodate the ghost schedule as shown below at Step 3b). A deadline transformation is said to be *feasible* if, when it is possible to generate a feasible schedule of clones with pretransformed deadlines, it is also possible to do so after the clones' deadlines have been transformed. Finally, we define $\boldsymbol{P}_2^*$ to be a modification of $\boldsymbol{P}_2$ that fills the holes to the extent possible with ghosts. In the event of more than one ghost competing for the same hole space, $\boldsymbol{P}_2^*$ will try to overlap them. If it is not possible to do this and still satisfy condition C2 (recall that if ghosts overlap in the schedule, only one of the overlapping ghosts can be activated), priority will be given to the ghost with the earliest deadline. The remaining ghosts will be scheduled as primary clones, with the cost functions indicated earlier, with the additional condition that the time-slices given to the ghosts under the schedule are moved as far right as possible, consistent with the need to meet all hard deadlines. This last condition is easily met. For, with all primaries having monotonically increasing cost functions, the only case in which this is not automatically done by $\boldsymbol{P}_2$ is when it would not alter the finishing time of a primary clone. In that event, all that is required to do is to "nest" the ghost within the primary as far right as possible. This means that, for any given time t, the nonprimary-hole space occupied by the ghosts prior to t is the minimum possible.

Algorithm $\boldsymbol{Q}$ is as follows. Let the set of primaries and ghosts allocated to i be π_i and θ_i, respectively. For every allocation of primaries and ghosts to processors by $\boldsymbol{P}_1$ do Steps 1–6 for every processor i:

Step 1) Record the holes for processor i, defined by the allocation.

Step 2) Run $\boldsymbol{P}_2^*(\pi_i \cup \theta_i)$. Return control to $\boldsymbol{P}_1$ if the schedule is found to be infeasible. Otherwise, record the positions of the ghosts in G.

Step 3) Set $j := 0$, and, while θ_i is nonempty, do a)–f).

a) Initialize $\boldsymbol{P}_2$ at $\nu_{\text{ghost}_i(j)}$. (* See definition of initialization above *.)

b) Set $\theta_i := \theta_i - \{\text{ghost}_i(j)\}$.

c) Transform the hard deadlines of the primary clones $k \in \pi_i$ to be $t_{dk}^{(0)} - \text{util}_{\theta_i}^{G}(0, t_{dk}^{(0)})$, where $t_{dk}^{(0)}$ is the original hard deadline of clone k and $\text{util}_{\theta_i}^{G}(a, b)$ is the nonprimary-hole space occupied by the ghosts in θ_i, according to the ghost schedule G. (* Guarantees there is enough time redundancy for all deadlines to be met if one or more ghosts in G are activated. *)

d) Run $\boldsymbol{P}_2(\pi_i)$ with these new deadlines, and let S_{ij} be the schedule obtained. (* S_{ij} is the contingency schedule, a portion of which will be used to obtain the optimal schedule. *)

e) Calculate the remaining run time of the clones scheduled in d) by subtracting from the current remaining run time the duration for which they were scheduled in the interval $[\nu_{\text{ghost}_i(j)}, \nu_{\text{ghost}_i(j+1)}]$.

f) Set $j := j + 1$.

Step 4) If all primary clones $k \in \pi_i$ have not been completed by $\nu_{\text{ghost}_i(j)}$, set $t_{dk} := t_{dk}^{(0)}$, initialize $\boldsymbol{P}_2$ at $\nu_{\text{ghost}_i(j)}$, and run $\boldsymbol{P}_2(\pi_i)$, recording the output in S_{ij}.

Step 5) Form schedule S_i by piecing together the schedules represented by S_{ij} in the interval $[\nu_{\text{ghost}_i(j)}, \nu_{\text{ghost}_i(j+1)}]$. Letting $\{C_k\}$, $k \in \pi_i$ be the completion times of the primary clones k according to schedule S_i, compute $\Sigma_{k \in \pi_i} f_k(C_k)$, and return this value as the cost associated with S_i. (* See below for explanation. *)

Step 6) Restore all hard deadlines of the primary clones $k \in \pi_i$ to their original values $t_{dk}^{(0)}$.

Notice that only in Step 2 are the ghosts scheduled: this ghost schedule determines the position of the ghosts for the given allocation. $\boldsymbol{Q}$ puts out a matrix of contingency schedules S_{ij}, with the optimal primary schedules (optimality is proved below in Theorem 2) being given by S_i^*, where i is the processor index and j the notification sequence index. Contingency schedule S_{ij} is invoked at time $\nu_{\text{ghost}_i(j)}$ if a) $\text{ghost}_i(j)$ is to be activated, and b) all ghosts l with $\nu_l < \nu_j$ and allocated to processor i will not be activated.

$\boldsymbol{Q}$ is a dynamic programming algorithm. For every processor i in the system, it begins by obtaining the ghost positions. Then, it obtains the optimal schedule that also satisfies conditions C1 and C2 for the ghosts in θ_i. This is schedule S_{i0}. The portion of S_{i0} in the interval $[\nu_0, \nu_{\text{ghost}_i(1)}]$ is also the corresponding portion of S_i^*. Run times of each primary clone are reduced by the amount of time they have been scheduled for under S_{i0} in the interval $[\nu_0, \nu_{\text{ghost}_i(1)}]$. Then, at $\nu_{\text{ghost}_i(1)}$, $\text{ghost}_i(1)$ is discarded from θ_i, and the above procedure is repeated to obtain S_{i1}. The portion of S_{i1} in the interval $[\nu_{\text{ghost}_i(1)}, \nu_{\text{ghost}_i(2)}]$ is the corresponding portion of S_i^*. Proceeding in this manner, $\boldsymbol{Q}$ pieces together the optimum primary schedule, which is the one that will be run if there is no ghost invocation. If $\text{ghost}_i(n)$ is activated and no ghost allocated to that processor with notification time prior to that of n is activated, the contingency schedule to be followed is S_{in}.

This schedule will ensure that even if any or all of succeeding ghosts (i.e., ghosts in θ_i with notification times greater than $\nu_{\text{ghost}_i(n)}$) are activated, all hard deadlines of tasks assigned by P_1 to processor i will continue to be met. The actual schedule to be used in the case of failure is then trivially obtained by amalgamating the ghost schedule onto the contingency schedule. When a ghost is activated, it retains its position as defined in the ghost schedule, interrupting any primary clone that may be occupying the same time-stretch in the contingency schedule. This effect can propagate, and all primary clones that are affected by this are moved right by an appropriate amount.

Fig. 3 provides an illustration of this. In this example, it is assumed that there is no hole for inserting the ghosts in the schedule prior to the deadline $t = 40$ of all the primary clones α, β, γ, and δ. If only ghost i is activated, then it retains in the actual schedule the position it had in the ghost schedule and interrupts primary β. If ghosts i, j, and k are all activated, each ghost retains its position as prescribed by the ghost schedule, appropriately interrupting and translating the primary clones.

Theorem 1: If step 2 of Q yields a schedule for a processor i which meets all deadlines, then schedule S_i^* which also satisfies all deadlines exists.

Proof: This follows immediately from the fact that ghosts are only scheduled by P_2^* either in the holes left by step 2 of Q, or at the last possible moment (consistent with the need to satisfy all hard deadlines). Indeed, owing to the way we have defined P_2^* the schedule of ghosts G obtained from P_2^* is the one for which $\text{util}_{\theta_i}^{G'}(0, t)$ is a minimum for any ghost schedule G' and time t if all deadlines are to be satisfied. All this means that if one or more primaries cannot be scheduled feasibly under the deadline transformation in Step 2, neither can a feasible schedule be found in Step 2, leading to a contradiction. □

Theorem 2: If P is optimal, S_i^* is optimal, under the constraints imposed above.

Proof: Let schedule S_i^* found by Q not be optimal for some i. Then there exists a schedule S', with an objective function value less than that of S_i^*, and in which at least one primary clone is scheduled to finish after the transformed hard deadline as defined in Step 3b). Consider one such clone j, assigned to processor i. Let clone j finish at time δ in schedule S_i^* and ϵ in schedule S'. By assumption, $\epsilon > \delta$. Let δ be in the notification interval h, i.e., $\delta \in [\nu_{\text{ghost}_i(h)}, \nu_{\text{ghost}_i(h+1)}]$. Denote by $\text{util}_i(a, b)$ the amount of nonhole space occupied on processor i by ghosts in the set $B = \{c \mid \nu_c > \nu_{\text{ghost}_i(h)}$ and c is allocated to $i\}$ in the interval [a, b]. Clearly, $\epsilon \in (t_{dj}^{(0)} - \text{util}_i(0, t_{dj}^{(0)}), t_{dj}^{(0)})$, where $t_{dj}^{(0)}$ denotes the original (i.e., pretransformed) deadline of clone j. If $\text{util}_i(0, \epsilon) = \text{util}_i(0, t_{dj}^{(0)})$, then, by the minimality of util_i, we have a contradiction: clone j will miss its deadline if all the ghosts in B are activated.

So, $\text{util}_i(0, \epsilon) < \text{util}_i(0, t_{dj}^{(0)})$, i.e., $\text{util}_i(\epsilon, t_{dj}^{(0)}) > 0$. If all ghosts in B scheduled prior to ϵ are activated, then clone j's finishing time will be increased
by at least $\text{util}_i(0, \epsilon) + \text{util}_i(\epsilon, \text{util}_i(0, \epsilon) + \epsilon)$.
Now, $\text{util}_i(\epsilon, \text{util}_i(0, \epsilon) + \epsilon) = \sigma_1 > 0$, since otherwise
$\text{util}_i(\text{util}_i(0, \epsilon) + \epsilon, t_{dj}^{(0)}) > t_{dj}^{(0)} - \epsilon - \text{util}_i(0, \epsilon)$, a contradiction. So, the finishing time for clone j will be incremented again, this time by σ_1. Similarly, we can show that $\text{util}_i(\text{util}_i(0, \epsilon) + \epsilon, \text{util}_i(0, \epsilon) + \epsilon + \sigma_1) = \sigma_2 > 0$, and so on. An infinite sequence $\{\sigma_i\}$ will be generated.

Now, if we can show that $\sum_{k=1}^{\infty} \sigma_k = \text{util}_i(0, t_{dj}^{(0)}) - \text{util}_i(0, \epsilon)$, we have proved that S' is infeasible.
Suppose, to the contrary, that $\sum_{k=1}^{\infty} \sigma_k = m < \text{util}_i(0, t_{dj}^{(0)}) - \text{util}_j(0, \epsilon)$.
Then, $\text{util}_i(\epsilon + m + \text{util}_i(0,\epsilon), t_{dj}^{(0)}) = \text{util}_i(0, t_{dj}^{(0)}) - \text{util}_i(0, \epsilon) - m$. Then, clearly, we have $\text{util}_i(\epsilon + m + \text{util}_i(0, \epsilon), t_{dj}^{(0)}) \le t_{dj}^{(0)} - \epsilon - m - \text{util}_i(0, \epsilon)$, i.e., $\epsilon \le t_{dj}^{(0)} - \text{util}_i(0, t_{dj}^{(0)})$, a contradiction. So S' does not exist. □

It is important to realize that it is only S_i^* that is optimal, i.e., that a contingency schedule S_{in} is, taken as a whole, not guaranteed to be optimal in that one or more of the schedules that arose from superimposing the ghost schedule on the contingency schedule need not be optimal.

We turn next to the question of relaxing assumption A1. If A1 is relaxed because the mean time between successive processor failures is small, our method breaks down since S_i^* can no longer be regarded as the optimal cost associated with a given task allocation. Fortunately, as remarked earlier, the mean time between failures is usually not small.

It is more likely, however, that A1 must be relaxed because the interrepair interval of the computer system is very long. This is true, for example, of computers aboard spacecraft. In such a case, it is useful to use the contingency schedules not for the entire duration of the mission, but to cover the interval during which a new optimum schedule (complete with a new primary and corresponding contingency schedules) is computed. This is profitable as long as the mean time between successive processor failures is much greater than the time taken to compute a new optimum schedule.

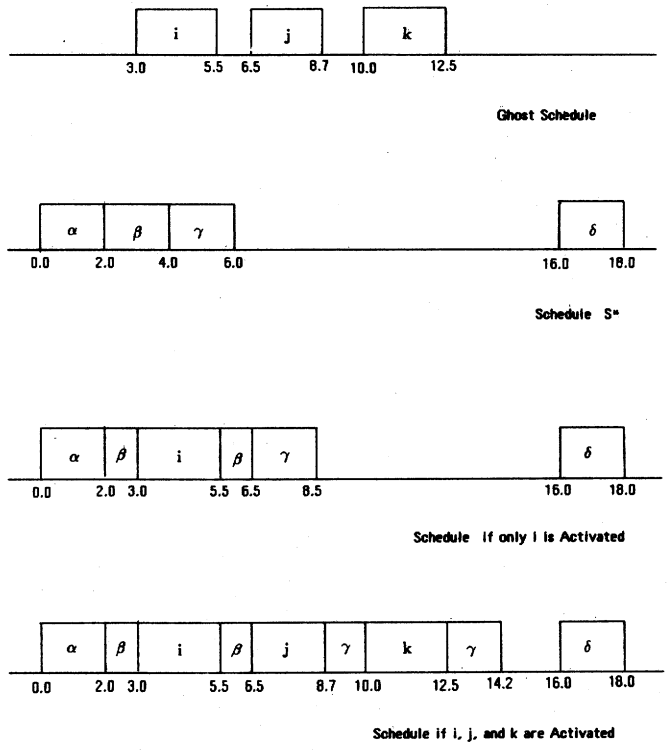


Fig. 3. Amalgamating ghost and contingency schedules.

Finally, we look at the burden this algorithm places on the computer during operation. This consists solely of amalgamating the ghost schedule with the contingency schedule, which amounts only to preempting the contingency schedule with the ghosts wherever necessary. The maximum number of preemptions is equal to the number of ghosts, and so the overhead for any given processor is directly proportional to the number of ghosts carried on that processor. Specifically, it is the product of the number of ghosts and the time consumed in handling preemptions. Since the latter time is usually very small, this is indeed an *on-line* algorithm.

By contrast, nothing can be said *a priori* about the complexity of the algorithm $\boldsymbol{Q}$, since it depends on the given algorithm $\boldsymbol{P}$.

IV. Example

We illustrate here the formation of an optimal schedule for a given allocation (specified by algorithm $\boldsymbol{P}_1$). We provide schedules for one of the processors only: the others are similarly derived.

Assume that $\boldsymbol{P}_1$ has allocated primary clones of task versions 1, 2, 3, and 4, and ghost clones of versions 5, 6, 7, and 8 to this processor. The cost function for primary clones of version i is given by:

$$f_i(t) = \begin{cases} t^{(5-i)} & \text{if } t < t_{di} \\ \infty & \text{otherwise.} \end{cases}$$

The hard deadlines for the task versions are $t_{d1} = 130$, $t_{d2} = 110$, $t_{d3} = 90$, $t_{d4} = 60$, $t_{d5} = 22$, $t_{d6} = 39$, $t_{d7} = 55$, and $t_{d8} = 130$. The run time, rt, of all these versions is 15 time units. The release times are: $\text{rel}_i = i$ for $i = 1, 2, 3, 4$, $\text{rel}_5 = 0$, $\text{rel}_6 = 24$, $\text{rel}_7 = 40$, and $\text{rel}_8 = 90$. Ghost notification times are: $\nu_5 = 0$, $\nu_6 = 20$, $\nu_7 = 40$, and $\nu_8 = 90$. These values were chosen arbitrarily, and have no physical significance by themselves.

The algorithm $\boldsymbol{P}_2$ employed by us is a heuristic that takes scheduling decisions at the release and at the finishing of clones. At each of these moments t, it computes, given the set of available clones R (i.e., clones released, but not yet finished), an array of values $V(i)$, where $V(i) = \sum_{j \neq i, j \in R} f_j(t + rrt_i + rrt_j)$, and rrt_k is the remaining (at time t) run time of clone k. The clone scheduled is l such that $V(l) = \min_{j \in R} V(j)$.

The holes are the intervals $[0, 1)$ and $(130, \infty)$. The contingency schedules are shown in Fig. 4. The optimal schedule S^* for that processor is pieced together from these as shown. This schedule, together with the cost it represents, is returned to algorithm $\boldsymbol{P}_1$.

In Fig. 5, we show the actual schedules followed upon ghost activation. The case treated is one in which contingency schedule S_{i0} has had to be invoked, i.e., the system has been notified that ghost 5 is to be activated. If only ghost 5 is activated, then it occupies the schedule during the intervals allocated to it by the ghost schedule, G. Primary clone 4 is thus preempted, and only resumes execution after ghost 5 has stopped running at time 22. The rest of the schedule gets shifted right by 14 time units to absorb the nonhole space occupied by ghost 5. If ghosts 5 and 6 are activated,

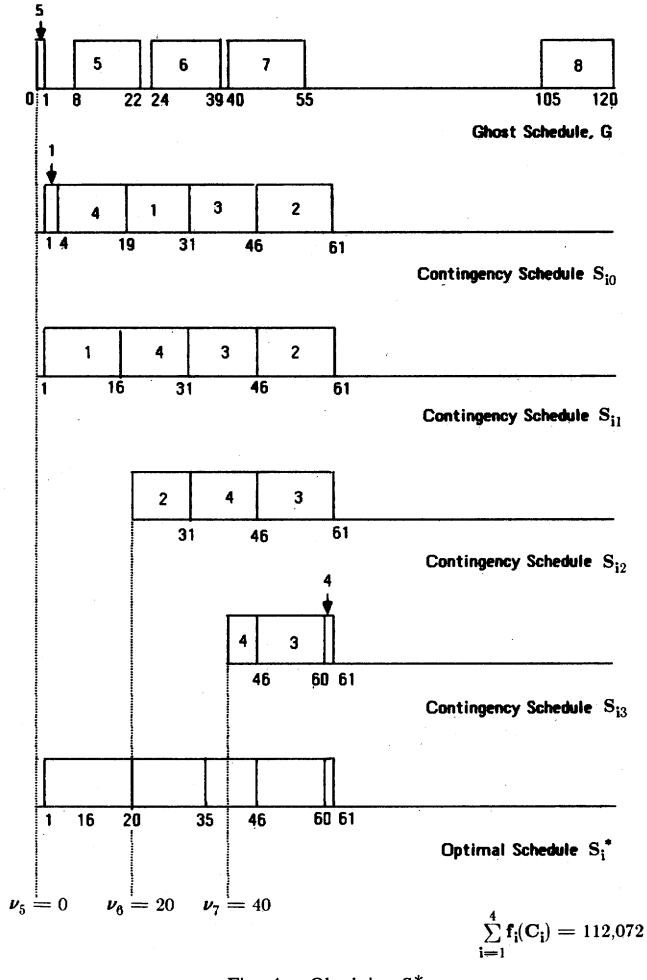


Fig. 4. Obtaining S_i^*.

then primary 4 is interrupted twice: once by 5 and then by 6. In each case, it must wait for the ghosts to complete before resuming execution. The rest of the schedule now gets shifted right by 29 time units. If ghosts 5 and 7 are activated, primary 4 is interrupted only by ghost 5, but primary 1 is also interrupted, by ghost 7. If ghosts 5, 6, and 7 are activated, then primary 4 is interrupted thrice, once by each of the ghosts, as shown. Notice that primary 4 just manages to meet its deadline: if it, in its turn, had not interrupted primary 1 upon release, this would have been impossible.

If the system is notified that ghost 5 is *not* to be activated, then schedule S_{i0} is discarded. Since $\nu_5 = 0$, this can be done at time 0, which is why S_i^* does not contain any portion of S_{i0}.

V. Discussion

This paper has introduced a dynamic programming algorithm to generate fault-tolerant schedules out of a given nonfault-tolerant schedule. This algorithm is expected to be used in embedded computers where extremely high reliabilities are required. Examples are aircraft-, spacecraft-, and nuclear reactor-controllers, life-support systems, generation and distribution of electric power, etc.

The usefulness of this approach increases as increased demands are made on reliability: in the "obvious" approach which has all clones primary, an extremely stringent reliability requirement would swamp the system with a large

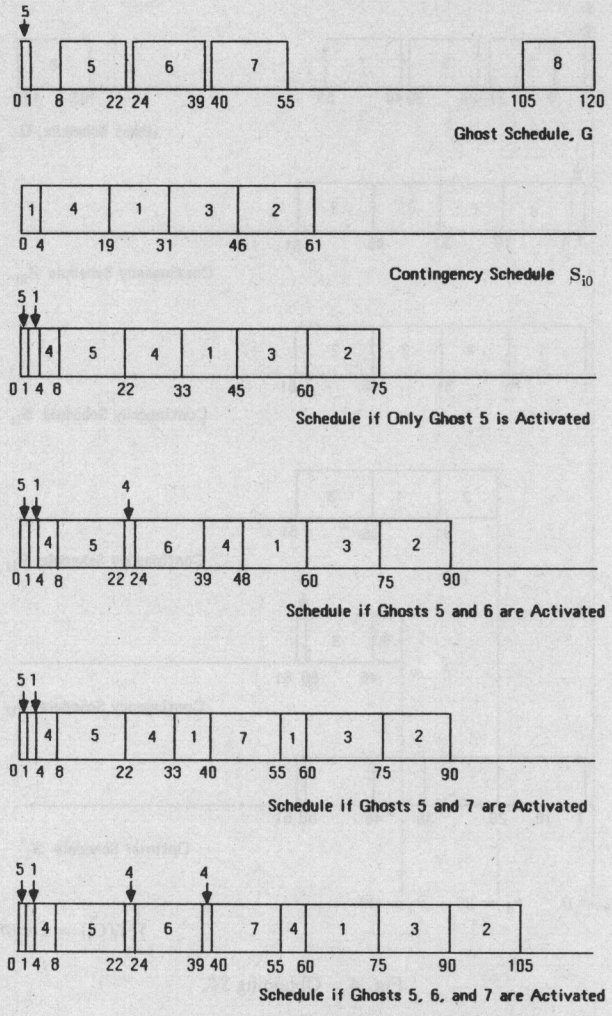


Fig. 5. Schedules when ghosts are activated.

number of primary clones, with all the implications that this would have for the computer response times and the consequent quality of (control) service provided by the computer. An example of this is provided in Fig. 6, where the obvious approach is used, all the clones in the earlier example being declared primary. The degradation in response times is clear on comparison with Fig. 5.

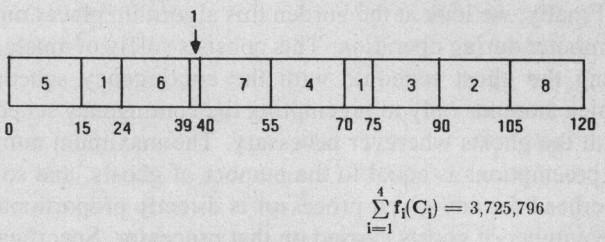
There are many obvious extensions to this work. One is to investigate the sensitivity of the optimal solution to changing the number of schedules each processor may carry: in this paper, we restricted it to two.

Appendix
List of Important Symbols

P: A nonfault-tolerant scheduling algorithm which can be decomposed into two subalgorithms P_1 and P_2. P_1 finds the optimal allocation of tasks to processors and also the optimal schedule by calling P_2, which is an optimum scheduler for uniprocessor systems.

Q: A fault-tolerant scheduling algorithm derived from P.

$f_i(C_i)$: Cost accrued due to a response time of C_i for version i.



$$\sum_{i=1}^{4} f_i(C_i) = 3{,}725{,}796$$

Fig. 6. All clones treated as primary.

L: Interrepair interval of the computer system.

ν_j: Notification time of ghost j.

t_{di}: Hard deadline of version i. $t_{dk}^{(0)}$ is the original hard deadline of clone k.

N_{sust}: The maximum number of processor failures that the system can tolerate.

$\rho(V)$: The set of processors allocated to run the primary clones of version V.

$\gamma(V)$: The set of processors carrying the ghost clones of V.

π_i (θ_i): The set of primaries (ghosts) allocated to processor i.

$\text{ghost}_i(j)$: The jth ghost (in order of notification time) allocated to processor i.

$\text{util}_{\theta_i}^{G}(a, b)$: The amount of nonhole space in the interval $[a, b]$ occupied by the ghosts in θ_i, according to the ghost schedule G.

S_{ij}: A contingency schedule on processor i, a portion of which is used to obtain the optimal schedule S_i^*.

Acknowledgment

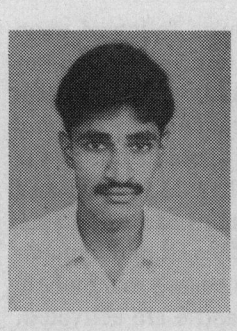
The authors wish to thank R. W. Butler and M. Holt of the NASA Langley Research Center and D. W. Mizell of Office of Naval Research for their technical and financial support, and the anonymous referees for their comments. Also, the authors are grateful to M. A. Pruder of the University of Michigan for the artwork in this paper.

References

[1] J. H. Wensley *et al.*, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE*, vol. 66, no. 10, pp. 1240–1255, Oct. 1978.

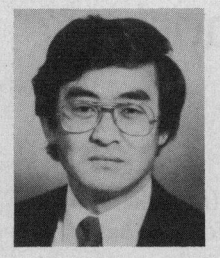
[2] C. M. Krishna and K. G. Shin, "Performance measures for multiprocessor controllers," in *Performance'83*, A. K. Agrawala and S. K. Tripathi, Eds. Amsterdam, The Netherlands: North-Holland, pp. 229–250, 1983.

[3] K. G. Shin, C. M. Krishna, and Y. -H. Lee, "A unified method for characterizing real-time computer controllers and its application," *IEEE Trans. Automat. Contr.*, vol. AC-30, no. 4, pp. 357–366, Apr. 1985.



C. M. Krishna (S'79–M'84) received the B.Tech. degree from the Indian Institute of Technology, Delhi, in 1979, the M.S. degree from Rensselaer Polytechnic Institute, Troy, NY, in 1980, and the Ph.D. degree from the University of Michigan, Ann Arbor, MI, in 1984, all in electrical engineering.

Since September 1984, he has been on the faculty of the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA. His research interests include reliability modeling, queueing and scheduling theory, and distributed architectures and operating systems.



Kang G. Shin (S'75–M'78–SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1970 to 1972 he served in the Korean Army as an ROTC officer and from 1972 to 1974 he was on the research staff of the Korea Institute of Science and Technology, Seoul, working on the design of VHF/UHF communication systems. From 1974 to 1978 he was a Teaching/Research Assistant and then an Instructor in the School of Electrical Engineering, Cornell University. From 1978 to 1982 he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a Visiting Scientist at the U.S. Air Force Flight Dynamics Laboratory in summer 1979 and at Bell Laboratories, Holmdel, NJ, in summer 1980 where his work was concerned with distributed airborne computing and cache memory architecture, respectively. He also taught short courses for the IBM Computer Science Series in the area of computer architecture. Since September 1982, he has been with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, MI, where he is currently an Associate Professor. His current teaching and research interests are in the areas of distributed and fault-tolerant computing, computer architecture, and robotics and automation.

Dr. Shin is a member of ACM, Sigma Xi, and Phi Kappa Phi.