Acta Informatica 20, 261–281 (1983)

Task Allocation in Fault-Tolerant Distributed Systems*

Joseph A. Bannister[1] and Kishor S. Trivedi[2]

[1] University of California, Los Angeles, CA 90024, USA
[2] Duke University, Durham, NC 27706, USA

Summary. This paper examines task allocation in fault-tolerant distributed systems. The problem is formulated as a constrained sum of squares minimization problem. The computational complexity of this problem prompts us to consider an efficient approximation algorithm. We show that the ratio of the performance of the approximation algorithm to that of the optimal solution is bounded by $9m/(8(m-r+1))$, where m is the number of processors to be allocated and r is the number of times each task is to be replicated. Experience with the algorithm suggests that even better performance ratios can be expected.

List of Important Symbols

n	number of tasks to be assigned
m	number of processors to be allocated
x_{ij}	1 if task i is assigned to processor j
r_i	number of clones of task i
M_{ij}	units of memory space required by task i on processor j
B_j	units of memory space available on processor j
I_{ij}	number of instructions executed by task i per iteration on processor j
T_i	task i's period
R_j	speed of a processor j in instructions per second
u_{ij}	task i's utilization of processor j
f_j	scheduling constant (e.g. 1 or ln 2) for processor j
r	number of clones of a task assuming a fixed level of replication
q_j	utilization of processor j
q_j^*	utilization of processor j under an optimal assignment
min-r	the r^{th} minimum of a sorted multiset
K_r^m	least upper bound on a family of series ratios
$q_*(t)$	used in computing K_r^*

* This work was supported in part by the National Aeronautics and Space Administration under and by the National Science Foundation under Grant US NSF MCS-8302000

1. Introduction

One of the first problems encountered in the operation of a distributed system is the problem of allocating the tasks among the processing nodes. Allocation problems of various types have been widely studied [1, 2, 4, 5, 9–11, 14, 15, 18–20, 22]. The allocation problem has typically been formulated as a constrained optimization problem. The constraining equations may describe system attributes such as limited memory capacity or given processor speed, and the objective is usually the minimization of some kind of cost function that varies with the particular allocation.

Allocation problems are solved by providing a general, cost effective procedure for finding the optimal assignment for specific instances of the problem. As a rule, allocation problems tend to be computationally intensive [5, 15, 19, 22]. This has spawned a variety of approaches to solving them. The three widely used approaches to solving allocation problems are graph theoretic, integer programming, and approximation methods. The graph theoretic approach represents the problem by a graph and then uses common graph techniques such as the max-flow, min-cut algorithm to solve for the best allocation [15, 18, 19]. The integer programming approach is perhaps the most widely used method in solving allocation problems [5, 10, 14, 21]. The problem can be formulated as an integer program, and well-known techniques such as implicit enumeration or branch-and-bound can be employed to find the solution. The approximation approach is used when one wants a fast algorithm that produces reasonably good approximations to the optimal solution [2, 3, 7, 10]. Other approaches to solving such problems include goal programming [11] and Markov decision theory [4]. This paper will concentrate on approximation methods for solving allocation problems. The approximation algorithm considered here is different in spirit from those advocated in [20, 22] where one first removes the integer constraint, solves the continuous optimization problem, discretizes the continuous solution, and obtains bounds on the discretization error. This bound is with respect to the continuous optimum, whereas in this paper we use an approximation directly to solve the discrete problem and bound its performance with respect to the discrete optimum.

A contribution of this paper is to consider a specific allocation problem that arises in the context of fault-tolerant distributed systems. These systems typically achieve their tolerance to faults by the use of redundancy in hardware, programs, or data. The use of redundancy enables the masking of faults by voting on multiple copies of replicated (hardware or software) modules' outputs. The use of redundancy also allows the replacement of a detected faulty function by a fault-free function. The use of redundancy is modeled here for certain fault-tolerant distributed systems.

Section 2 gives the necessary background on the type of system being modeled and on real-time scheduling. Section 3 presents the mathematical formulation of the allocation problem. In Sect. 4 we present an efficient approximation algorithm to solve the problem defined in Sect. 3, and we show that the algorithm does not deviate significantly from the optimum. Section 5 discusses empirical results comparing the approximation algorithm to the

optimal algorithm. We summarize and discuss the significance of our results in Sect. 6.

2. Background

First we briefly describe a real-time fault-tolerant distributed system similar to the Software-Implemented Fault-Tolerance (SIFT) computer. The SIFT computer was originally designed by SRI International to be used on board large passenger aircraft [23–25]. The prime purpose behind the SIFT computer was to make it possible for flight crews to control the new generation of fuel efficient aircraft. The SIFT computer performs functions associated with flight control, engine control, navigation, guidance, and other avionic functions. Control functions such as these are essentially iterative in nature. A specific avionic control task is typically performed at a constant frequency of from one to 1,000 times per second. These tasks must execute fast enough so that they can be completed within their iteration periods.

As described originally [23, 24] the SIFT computer is a system in which multiple processors are interconnected by a triplicated bus. See Fig. 1 for a processor-memory-switch (PMS) description of the SIFT system. Each processor has a local memory and communicates with other processors by means of a message passing protocol. A task consists of several clones that are resident on and executed by a number of separate processors. Output values of all clones of a task are submitted to a vote, and the majority value is provided to consumers of the task's output. Thus if any single clone of a task for any reason (e.g. processor, memory or bus failure) produces incorrect output, that output will be suppressed by the majority vote. The voting is done under program control, rather than through hard-wired circuitry – hence the sobriquet "Software-Implemented Fault-Tolerance".

The SIFT system was designed to execute a set of tasks composed pre-

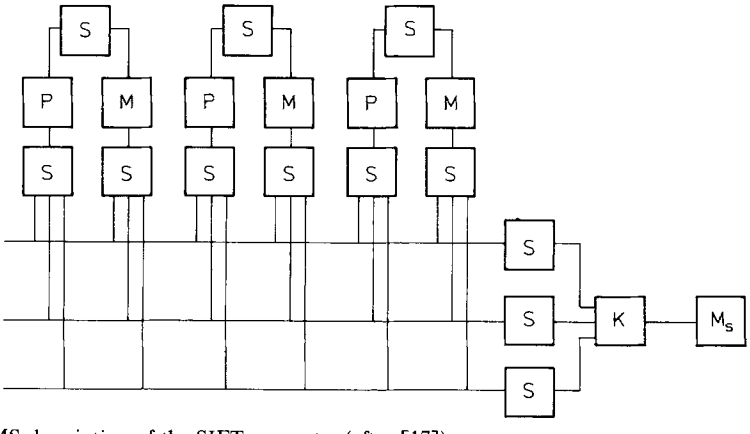

Fig. 1. PMS description of the SIFT computer (after [17])

dominantly of closed-loop control functions, or in the parlance of [8], periodic real-time tasks. By periodic it is meant that the task is requested with a fixed frequency, e.g. 50 times per second. By real-time it is meant that each task has some deadline that it must meet. A periodic real-time task can then be specified as a quadruple (s, e, d, p) with $0 < e \leqq d \leqq p$, where s represents the time that the task is first requested, e is the execution time of the task, d is the task's deadline, and p is the task's period. In this scheme the $(k+1)$-st request for the task occurs at time $s+kp$ and the deadline for this request comes at time $s+kp+d$. The utilization of the task is given by e/p. It is common to assume that $s=0$ and $d=p$.

The problem of scheduling periodic real-time tasks on a single processor has been studied in [13], where two algorithms were proposed. Both algorithms schedule tasks preemptively, which means that any task may be interrupted and later resumed at the point where it was interrupted - the requirement being that the task should complete before its deadline expires. The first algorithm, called the rate-monotonic priority assignment algorithm, assigns static priorities to tasks according to their iteration periods - tasks with smaller iteration periods have higher priorities. At any given time the rate-monotonic priority assignment algorithm starts the task with the highest priority provided the task has not already been completed for its current iteration period. In [13] it was proved that a set of n tasks can be scheduled on a processor so that each task completes within its iteration period as long as the processor's utilization will be no greater than $n(2^{1/n}-1)$ which tends to $\ln 2$ (approximately 0.69) as n grows. The quantity $\ln 2$ is a sufficient, though not a necessary, condition for scheduling with the rate-monotonic priority assignment algorithm. The second algorithm, called the deadline driven scheduling algorithm, assigns dynamic priorities to tasks on the basis of proximity of deadline. Thus the algorithm always starts the task whose iteration period will most imminently expire (assuming that the task has not yet been completed for the current iteration period). The deadline driven scheduling algorithm will schedule a set of tasks on a processor so that all tasks are completed within their iteration periods so long as the processor's utilization will not be greater than unity.

In [6, 12] the problem of scheduling periodic real-time tasks on a multiple processor system was studied. There are two basic approaches to scheduling periodic real-time tasks on a multiple processor system. The first approach is called the partitioning method and it seeks to partition the task set into groups and assign the groups to distinct processors. The group of tasks assigned to a processor can then be scheduled by techniques for single processor systems. The second approach is called the nonpartitioning method and it treats the entire collection of processors as one large virtual processor that uses its increased computing power to quickly execute the entire task set sequentially (as opposed to the parallel execution of tasks in the partitioning method). The entire task set is then scheduled on the single virtual processor as in the single processor case. In this paper we will be concerned exclusively with the partitioning method.

3. Mathematical Formulation of the Problem

Consider the following general allocation problem schema. Suppose that we are given fixed parameters r_i, M_{ij}, B_j, u_{ij}, and f_j where $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$. We wish to find an n by m matrix (x_{ij}) of zeros and ones satisfying the following problem:

minimize

$$\sum_{j=1}^{m} \left(\sum_{i=1}^{n} u_{ij} x_{ij} \right)^2 \tag{1}$$

subject to constraints

$$\sum_{j=1}^{m} x_{ij} = r_i \quad \text{for } 1 \leqq i \leqq n, \tag{2}$$

$$\sum_{i=1}^{n} M_{ij} x_{ij} \leqq B_j \quad \text{for } 1 \leqq j \leqq m, \tag{3}$$

$$\sum_{i=1}^{n} u_{ij} x_{ij} \leqq f_j \quad \text{for } 1 \leqq j \leqq m. \tag{4}$$

The problem specified in (1)–(4) is a constrained sum of squares minimization that may be used to model a task allocation problem (TAP) in a SIFT-like system. Assume that the system consists of m processors. There are n tasks which must be periodically executed. Each task will be loaded into a certain number of processors' local memories and executed by those processors. Let (x_{ij}) be the n by m matrix of zeros and ones with x_{ij} equal to one if task i is scheduled on processor j and x_{ij} equal to zero otherwise. The number of processors to which task i is assigned is the replication factor r_i of the task; that is, the task i is composed of r_i clones, and each clone is resident on and executed by a distinct processor. Each task i must then be assigned to exactly r_i distinct processors as indicated by Eq. (2).

If task i requires M_{ij} units (bytes, words) of memory to execute on processor j, and processor j has B_j units of memory available, then the task assignment must satisfy inequality (3).

For each task i we also know the number of instructions in the task, I_{ij}, when it is executed on processor j, and the iteration period of the task, T_i. The iteration period of the task is an amount of time during which the task must be executed at least once. The iteration period is dictated by physical conditions such as how frequently a device must be serviced or how often an aileron must be controlled. If the speed of processor j is R_j instructions per second, then task i requires I_{ij}/R_j seconds for its execution on processor j. The quantity $I_{ij}/(R_j T_i)$ represents the fraction of its iteration period that task i actually performs computations (if assigned to processor j). We will refer to this ratio as task i's utilization of processor j, designated u_{ij}. Recall that if a collection of tasks is assigned to a processor so that their utilization of the processor falls below a certain level, then those tasks can be scheduled on the

processor. Constraint (4) specifies that those tasks assigned to a processor must be schedulable on that processor by one of the scheduling disciplines discussed above. If the rate-monotonic priority assignment algorithm is used on processor j, then f_j has the value $\ln 2$. In the case that processor j uses the deadline driven algorithm to schedule its tasks, f_j assumes the value one.

In a SIFT-like system minimization of the objective function (1) is desired as a way to achieve load balancing among the system's processors. The quantity

$$q_j = \sum_{i=1}^{n} u_{ij} x_{ij}$$

represents the utilization of processor j under allocation (x_{ij}). There are several ways to measure the imbalance in the processor utilizations $q_1, \ldots, q_m$. The following two formulas immediately come to mind:

$$\frac{1}{m} \sum_{j=1}^{m} \left[q_j - \frac{1}{m} \sum_{j=1}^{m} q_j \right]^2 = \frac{1}{m} \sum_{j=1}^{m} (q_j)^2 - \left[\frac{1}{m} \sum_{j=1}^{m} q_j \right]^2, \tag{5}$$

$$\frac{\frac{1}{m} \sum_{j=1}^{m} (q_j)^2}{\left[\frac{1}{m} \sum_{j=1}^{m} q_j \right]^2}. \tag{6}$$

Formula (5) is known as the statistical variance of the q_j's and formula (6) is the normalized coefficient of variation of the q_j's (equal to one plus the squared coefficient of variation).

The SIFT system is a homogeneous system in which all processors are identical. Homogeneity greatly simplifies system development, validation, and maintenance. The assumption of homogeneity greatly simplifies the TAP as well: one may now assume that $u_{ij} = u_{ik}$, $M_{ij} = M_{ik}$, and $B_j = B_k$ for $1 \leqq n$, $1 \leqq j \leqq m$, and $1 \leqq k \leqq m$. Thus we need only consider problem parameters u_i, M_i, and B. It is also likely that all processors might be scheduled by a single scheduling algorithm and all tasks might undergo a common level of cloning. This further reduces the problem's complexity by requiring us to consider a single replication factor r and a single scheduling constant f. Given a homogeneous system the quantity $\sum_{j=1}^{m} q_j$ is independent of the particular allocation (x_{ij}) under consideration. This term will be constant in formulas (5) and (6) and so both (5) and (6) may be minimized by minimizing (1).

We have assumed that the level of interprocessor communication will not be significantly affected by the choice of allocation in a SIFT-like system. The (relatively) constant interprocessor communication overhead is assumed to be incorporated in the u_{ij}'s.

The objective of balancing computational load evenly among all processors of the system follows from the desire to maximize the reliability of the system. Suppose that the tasks have been assigned so that processor utilization is unbalanced, e.g. one processor is significantly more utilized than the others. An underlying postulate of the system is that any processor is vulnerable to failure - so suppose that the overutilized processor has experienced a failure. In a

certain sense, this overutilized processor represents a weak link in the system. Some of the dangers associated with a system of unevenly utilized processors are:

1. The failure of the overutilized processor makes it necessary to load, set up, and restart on other processors the tasks originally assigned to the failed processor. The time required for this reconfiguration varies roughly with the reassigned tasks' utilizations. Thus the failure of the overutilized processor jeopardizes the robustness of the system by increasing the likelihood that the system may not recover within a reasonable grace period.

2. The slack time (time when no task is running) of the overutilized processor is significantly less than the slack time of the other processors. The effect of reducing slack time is to reduce the amount of time which might have been allotted to running diagnostic programs. This in turn increases the probability that the overutilized processor is "underdiagnosed" and is therefore more prone to produce erroneous results. Such errors could seriously impair the intended operation of the system.

3. Although we have assumed a priori knowledge of the tasks' characteristics (notably running time), these characteristics are in fact nondeterministic by nature. For instance, a task's running time will be influenced by program branching, data dependencies, and hardware fluctuations and may therefore vary from activation to activation. The tasks' tendency to exhibit this random variation means that we risk missing task deadlines by overutilizing a processor. Missed deadlines will have a deleterious effect on the system.

4. In cases where diagnostic programs are not employed or are employed for only a fraction of the slack time, it is conceivable that the failure rate of a processor will increase with its duty cycle. This implies that the overutilized processor, by virtue of being computationally "overburdened," suffers a higher failure rate than its less utilized comrades.

All of these considerations acquire considerable importance in light of the fact that we are considering a life-critical application. The above discussion suggests that a balanced system is inherently more reliable than an unbalanced system. Note that the minimum value of (1) corresponds to the best possible approximation of a perfectly balanced system.

4. An Efficient Approximation Algorithm for the Problem

We will now consider an approximation algorithm for restricted versions of TAP. Depending on the parameters of TAP (e.g. u_i, n, m, r), there may or may not be a solution to the problem. For example, in TAP there may not be a sufficient number of processors to satisfy the separation, capacity, and scheduling constraints with any particular allocation. In practice, the system designers will have built the system with enough processing power to accommodate a wide range of task sets. Appealing to the good judgement and foresight of the system architect, we shall henceforth assume that the memory capacity and processing speed of the processors, as well as the number of processors are sufficient to accommodate all reasonable allocations (where a reasonable allocation might self servingly be defined as one produced by the soon to be presented approximation algorithm). The point here is that the system was

designed with a dedicated application in mind and should therefore be expected to handle all allocations associated with the application.

Our intent is to decide which of the many possible allocations is best, given that we are constrained to make this decision in real-time. This real-time requirement is necessary in the types of fault-tolerant systems that we are considering. Processors are at all times subject to failure with subsequent removal of chronically faulty processors from the system. The removal of processors may then necessitate shedding some of the tasks. Every time this occurs a new allocation must be created. One possibility is to compute optimal allocations for various combinations of tasks and processors off-line, perhaps using integer programming methods. These allocations could then be stored in tables for system use after reconfiguration. The major drawback with this is that the number of conceivable combinations of tasks and processors is very large for even moderately sized systems.

It is therefore desirable to have a procedure for creating allocations of arbitrary sets of processors to arbitrary sets of tasks on-line and in real-time. Since there is little hope of computing optimal allocations in real-time, what is needed is some sort of "quick and dirty" solution to the problem, though we prefer a solution that is long on "quick" and short on "dirty." To this end we propose an algorithm for allocating processors to tasks. Suppose that we have m processors to be allocated to n tasks, with each task to be assigned to r distinct processors. Also suppose that the tasks have been sorted so that their utilizations are in descending order: $u_1 \geqq \ldots \geqq u_n$. The algorithm (dubbed Allocate) works as follows:

(1) Initialize $q_j := 0$ for $1 \leqq j \leqq m$. Initialize $i := 1$.
(2) Assign the r clones of task i to the r least utilized processors. Set $q_j := q_j + u_i$ for each processor j that task i is assigned to. Increment $i := i + 1$.
(3) If $i > n$ then end else go to (2).

Algorithm Allocate is a best fit algorithm. The algorithm consists of n basic steps, one for each task being assigned. During the i-th step the algorithm assigns the r clones of the i-th task to the r least utilized processors and updates their utilizations to reflect the assignment. This results in a sort of "balance-as-you-go" effect. The execution time of Allocate is bounded above by $O(mnr)$ which compares favorably with the exhaustive search algorithm. [If the task utilizations are not already sorted, we must add $O(n \log n)$ to the above.]

Notice that our description of the algorithm ensures that the separation constraint (2) is met. However, we have said nothing about the capacity constraint (3) or the scheduling constraint (4). The algorithm can be modified to keep track of memory consumption and check for violations of capacity and scheduling violations. In the case of a SIFT-like system, if a violation is detected, then additional processors must be used or tasks must be shed. In our discussion we will assume that there are enough processors to accommodate the assignment without capacity or scheduling violations.

The main result of this section is that algorithm Allocate produces allocations that are relatively well balanced when compared to the optimum. More formally, when algorithm Allocate assigns the r clones of n tasks to m processors, the resulting sum of squares is guaranteed to be no more than

$\frac{9}{8}\frac{m}{m-r+1}$ times greater than the optimal sum of squares. This bodes well for real design problems, since m is usually considerably larger than r, and as m grows large in relation to r the ratio tends to 1.125. To use the TAP as an example, one might expect up to 200 triplicated tasks to be executed by a 20 processor system. In this example the approximate allocation produced by Allocate would never be more than 25% off optimum.

We will derive the performance guarantee via a technique that we dub the series ratio method. The series ratio method was first used in deriving a performance guarantee for the problem of minimizing sum of squares (without cloning) [3]. This method relies on the fact that the ratio of two appropriately defined series of numbers can be bounded from above. We then show that the ratio of Allocate's sum of squares to the optimal sum of squares can be made to conform to the constraints of the series, and thus bounded from above. Recall that q_j represents processor j's utilization, i.e. $q_j = \sum_{i=1}^{n} u_i x_{ij}$ given the allocation (x_{ij}). The problem then becomes one of minimizing $\sum_{j=1}^{m} q_j^2$ while providing that distinct clones of any task are assigned to distinct processors. The basic strategy for proving that the absolute performance ratio is bounded for the given parameters is as follows.

1. Establish technical and auxiliary lemmas for manipulating the problem (Lemmas 1, 2, 5).
2. Show that the approximate and optimal sums of squares can be made to fit the constraints of the series used in the series ratio method (Lemmas 3, 4, 6).
3. Compute the upper bound on the series ratio (Lemma 7).
4. The main result follows immediately (Theorem 1).

First we state two simple lemmas that will be used later.

Lemma 1. *Suppose that there is an assignment of n tasks to m processors with processor utilizations $q_1, \ldots, q_m$. If $q_j \leqq Q$ for $s \leqq j \leqq m$, then any other assignment with processor utilizations $q'_1, \ldots, q'_m$ satisfying $\sum_{j=s}^{m} q'_j \leqq \sum_{j=s}^{m} q_j$ must satisfy the inequality $\min_{j=s}^{m} \{q'_j\} \leqq Q$.*

Proof. Assume that $\min_{j=s}^{m} \{q'_j\} > Q$. Then $q'_j > Q$ for $s \leqq j \leqq m$. Hence

$$(m-s+1)Q \geqq \sum_{j=s}^{m} q_j \geqq \sum_{j=s}^{m} q'_j > (m-s+1)Q.$$

We have a contradiction; so $\min_{j=s}^{m} \{q'_j\} \leqq Q$. #

The next lemma, stated without proof, will be used to make comparisons between the sums of squares of two allocations. It states that if we transfer a task (represented by x in the lemma) from a processor of lesser utilization to a processor of greater utilization, then the new sum of squares will be greater than the old sum of squares.

Lemma 2. *If $a \geqq b \geqq 0$ and $x \geqq 0$, then $(a+x)^2 + b^2 \geqq a^2 + (b+x)^2$.*

The next result is a key lemma showing that the optimal assignment can be transformed into a new assignment without increasing the sum of squares. The new assignment is shown to agree with the approximate assignment on a certain subset of the processors.

Lemma 3. *Suppose that r clones of n tasks are to be assigned to m processors so that distinct clones of any task are assigned to distinct processors. Let* $q_1, \ldots, q_m$ *be the processor utilizations resulting from algorithm Allocate, ordered so that* $q_1 \geqq \ldots \geqq q_m$. *Let* $q_t = \max\{q_j$: *processor j is allocated to two or more tasks*$\}$, *and define* $s = \min\{j: q_j \leqq q_t$ *and processor j is allocated to two or more tasks*$\}$. *(If no such t exists, i.e. no processor is allocated to more than one task, then define* $s = m+1$.*) If* $q_1^*, \ldots, q_m^*$ *are processor utilizations in an optimal assignment, then there exists an assignment with processor utilizations* $q_1', \ldots, q_m'$ *such that* $q_j' = q_j$ *for* $1 \leqq j < s$ *and*

$$\sum_{j=1}^{m} (q_j')^2 \leqq \sum_{j=1}^{m} (q_j^*)^2.$$

The assignment producing processor utilizations $q_1', \ldots, q_m'$ *might not assign distinct clones of a task to distinct processors.*

Proof. For $1 \leqq i \leqq n$ assume that u_i is task i's utilization and that $u_1 \geqq \ldots \geqq u_n$. Consider the assignment produced by Allocate, reordered so that $q_1 \geqq \ldots \geqq q_m$. The allocation is depicted in Fig. 2. This figure indicates that if some processors are allocated to only single tasks, then these processors will be the most utilized in the system. Without loss of generality, assume that processor s is allocated to at least two tasks and processors $1, \ldots, s-1$ are each allocated to single tasks. Assume that only tasks $1, \ldots, k$ are assigned to processors $1, \ldots, s-1$ and only tasks $k+1, \ldots, n$ are assigned to processors $s, \ldots, m$.

Let $q_1^*, \ldots, q_m^*$ be the processor utilizations in an optimal assignment, ordered so that $q_1^* \geqq \ldots \geqq q_m^*$. We will transform this optimal assignment as follows:

for $j := 1$ to k do
  for $i := 1$ to r do
    find the processor allocated to clone i of task j and reassign all tasks but j to the least utilized processor allocated to tasks chosen only from $k+1, \ldots, n$, and update processor utilizations.

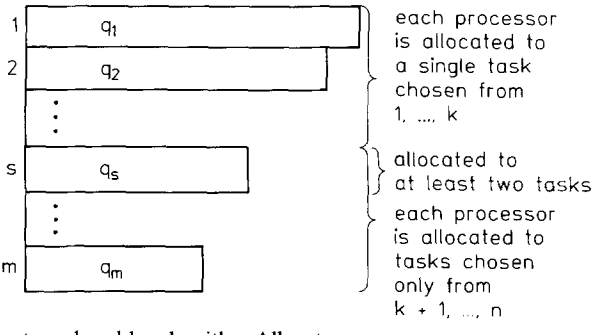

Fig. 2. An assignment produced by algorithm Allocate

This transformation will produce an allocation in which tasks $1, \ldots, k$ are assigned to $s-1$ processors, with exactly one task assigned to each of these processors. It will now be shown that this transformation is always possible. Any assignment must have no more than $s-1$ processors allocated to tasks $1, \ldots, k$. Therefore there will always be at least one processor allocated to tasks chosen only from $k+1, \ldots, n$ if $1 \leqq s \leqq m$, and hence the transformation can be performed. The only other case to consider is when $s=m+1$, in which case Allocate assigned at most one task per processor and it is easy to verify that the transformation is possible. [In fact there is no "transformation" in this case - the "transformed" assignment is identical to the optimal assignment.] Since the transformation is possible, assume that the new assignment has processor utilizations $q'_1, \ldots, q'_m$, reordered so that $q'_j = q_j$ for $1 \leqq j < s$.

It now remains to show that

$$\sum_{j=1}^{m} (q'_j)^2 \leqq \sum_{j=1}^{m} (q^*_j)^2.$$

We will demonstrate that each step of the transformation produces a new assignment whose sum of squares is no greater than the previous assignment's. Suppose that the previous assignment had processor utilizations $p_1, \ldots, p_m$ and reassignment of tasks in processor t_1 to processor t_2 produced a new assignment with processor utilizations $p'_1, \ldots, p'_m$. Assume that the reassigned tasks had total utilization x. Task k_1 was the task examined in the transformation and $k_1 \leqq k$. Thus we have $p_{t_1} = u_{k_1} + x$. The old and new assignments are shown in Figure 3. Since $u_{k_1} \geqq q_j$ for $s \leqq j \leqq m$, by Lemma 1 there is a processor t_2 allocated to tasks chosen only from $k+1, \ldots, n$ such that $p_{t_2} \leqq u_{k_1}$. Then by Lemma 2

$$\begin{aligned}
\sum_{j=1}^{m} (p_j)^2 &= \Big[\sum_{j \neq t_1, j \neq t_2} p_j^2\Big] + (u_{k_1} + x)^2 + (p_{t_2})^2 \\
&\geqq \Big[\sum_{j \neq t_1, j \neq t_2} p_j^2\Big] + (u_{k_1})^2 + (p_{t_2} + x)^2 \\
&= \sum_{j=1}^{m} (p'_j)^2.
\end{aligned}$$

Note that processor t_2 may have been allocated to a clone of a task also contained in the shaded region of Fig. 3. Thus the transformation might not produce an assignment in which distinct clones of a task are assigned to distinct processors.

Hence the transformed assignment with processor utilizations $q'_1, \ldots, q'_m$ has $q'_j = q_j$ for $1 \leqq j < s$ and

$$\sum_{j=1}^{m} (q'_j)^2 \leqq \sum_{j=1}^{m} (q^*_j)^2. \quad \#$$

Definition 1. If $S = \{s_1, \ldots, s_m\}$ is a multiset (i.e. a set in which multiple occurrences of an element are possible - also known as a bag) in which $s_1 \leqq \ldots \leqq s_m$, then for $1 \leqq r \leqq m$ define min-r $S = s_r$. This is the r^{th} minimum of S.

For example, if we take S to be the multiset {1, 2, 2, 8, 10}, then min-3 $S = 2$.

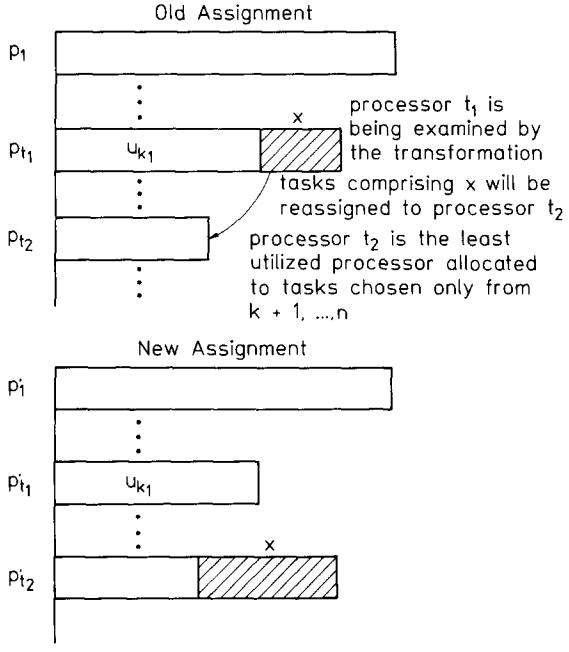

Fig. 3. Reassignment of tasks in the transformation of Lemma 3

Lemma 4. *Suppose that r clones of n tasks are to be assigned to m processors so that distinct clones of any task are assigned to distinct processors. Let* $q_1, \ldots, q_m$ *be the processor utilizations resulting from algorithm Allocate. If a maximally utilized processor is allocated to at least two tasks, then* $\max_{j=1}^{m} \{q_j\} \leqq 2 \min_{j=1}^{m}\text{-}r \{q_j\}$.

Proof. Let s be a maximally utilized processor allocated to at least two tasks. Suppose that task k was the last task assigned (by algorithm Allocate) to processor s and $p_1, \ldots, p_m$ were the processor utilizations just prior to the assignment of task k. It is clear that $p_j \leqq q_j$ for $1 \leqq j \leqq m$, and hence

$$\min_{j=1}^{m}\text{-}r\{p_j\} \leqq \min_{j=1}^{m}\text{-}r\{q_j\}. \tag{7}$$

The r clones of task k were assigned to the r least utilized processors; thus

$$p_s \leqq \min_{j=1}^{m}\text{-}r\{p_j\}. \tag{8}$$

Since at least two tasks were assigned to processor s, $p_s \geqq u_k$. Hence

$$p_s + u_k \leqq 2p_s. \tag{9}$$

But $p_s + u_k = \max_{j=1}^{m} \{q_j\}$; so by (7), (8) and (9),

$$\max_{j=1}^{m}\{q_j\} \leqq 2p_s$$
$$\leqq 2 \min_{j=1}^{m}\text{-}r\{p_j\}$$
$$\leqq 2 \min_{j=1}^{m}\text{-}r\{q_j\}. \quad \#$$

The following result is obvious and stated without proof.

Lemma 5. *If* $a \geqq b > 0$ *and* $c \geqq d > 0$, *then*

$$\frac{a+c}{a+d} \leqq \frac{b+c}{b+d}.$$

The following definition is central to the series ratio method. Two series are specified and the value K_r^m is defined to be the least upper bound of their ratios.

Definition 2. Let $p=(p_1, \ldots, p_m)$ and $q=(q_1, \ldots, q_m)$ be two sequences of non-negative numbers such that $\sum_{j=1}^{m} p_j = \sum_{j=1}^{m} q_j$ and $\max_{j=1}^{m}\{q_j\} \leqq 2 \min_{j=1}^{m}\text{-}r\{q_j\}$. For $1 \leqq r \leqq m$ define

$$K_r^m = \operatorname*{lub}_{p,q}\left[\frac{\sum_{j=1}^{m}(q_j)^2}{\sum_{j=1}^{m}(p_j)^2}\right].$$

[It is easy to show that such a lub exists and hence the definition is proper.]

Lemma 6 states that the ratio of the approximate to the optimal sum of squares can be made to fit the format of the series ratio method. The absolute performance ratio is therefore bounded above by K_r^m.

Lemma 6. *Suppose that r clones of n tasks are to be assigned to m processors so that distinct clones of any task are assigned to distinct processors. Let* $q_1, \ldots, q_m$ *be the processor utilizations resulting from algorithm Allocate, ordered so that* $q_1 \geqq \ldots \geqq q_m$. *Let* $q_1^*, \ldots, q_m^*$ *be the processor utilizations in an optimal assignment. Then*

$$\frac{\sum_{i=1}^{m}(q_j)^2}{\sum_{j=1}^{m}(q_j^*)^2} \leqq K_r^m.$$

Proof. Let s and $q_1', \ldots, q_m'$ be as in Lemma 3, i.e. $q_j' = q_j$ for $1 \leqq j < s$ and

$$\sum_{j=1}^{m}(q_j')^2 \leqq \sum_{j=1}^{m}(q_j^*)^2.$$

Then

$$\frac{\sum_{j=1}^{m}(q_j)^2}{\sum_{j=1}^{m}(q_j^*)^2} \leqq \frac{\sum_{j=1}^{m}(q_j)^2}{\sum_{j=1}^{m}(q_j')^2}$$

$$= \frac{\sum_{j<s}(q_j)^2 + \sum_{j\geqq s}(q_j)^2}{\sum_{j<s}(q_j')^2 + \sum_{j\geqq s}(q_j')^2}$$

$$= \frac{\sum_{j<s}(q_j)^2 + \sum_{j\geqq s}(q_j)^2}{\sum_{j<s}(q_j)^2 + \sum_{j\geqq s}(q_j')^2} \tag{10}$$

$$\leqq \frac{\sum_{j<s}(q_s)^2 + \sum_{j\geqq s}(q_j)^2}{\sum_{j<s}(q_s)^2 + \sum_{j\geqq s}(q_j')^2} \tag{11}$$

$$\leqq K_r^m. \tag{12}$$

Here (11) follows from (10) by noting that $q_j > q_s$ for $1 \leqq j < s$ and then applying Lemma 5. From Lemma 4 $q_s \leqq 2 \min_{j=s}^{m}\text{-}r\{q_j\}$ since $q_s, \ldots, q_m$ represents the allocation that results from applying algorithm Allocate to tasks $k+1, \ldots, n$ on processors $s, \ldots, m$ (where k is used as in the proof of Lemma 3). Now let $p_j = q_s$ for $1 \leqq j \leqq s$, and let $p_j = q_j$ for $s < j \leqq m$. Likewise let $p_j' = q_s$ for $1 \leqq j \leqq s$, and let $p_j' = q_j'$ for $s < j \leqq m$. The sequences $p_1, \ldots, p_m$ and $p_1', \ldots, p_m'$ appear in the numerator and denominator, respectively, of (11). We also see that

$$\max_{j=1}^{m}\{p_j\} = q_s \leqq 2 \min_{j=1}^{m}\text{-}r\{p_j\}$$

and $\sum_{j=1}^{m} p_j = \sum_{j=1}^{m} p_j'$. Equation (12) then follows from (11) by Definition 2 of K_r^m. #

We now compute a tight bound on K_r^m.

Lemma 7. *For* $m \geqq r \geqq 1$

$$K_r^m \leqq \frac{9}{8}\frac{m}{m-r+1}.$$

Proof. Let $q(t)$ be a nonincreasing step function (with steps at integers) defined for $0 \leqq t \leqq m$ such that:

$$q(0) \leqq 2q(m-r+1), \tag{13}$$

$$\int_0^m q(t)\,dt = 1, \tag{14}$$

$\int_0^m q^2(t)\,dt$ is maximized for the given m and r. (15)

Let $q_*(t)$ be such a function. Suppose that $p=(p_1, \ldots, p_m)$ and $q=(q_1, \ldots, q_m)$ are sequences of nonnegative numbers such that

$$\sum_{j=1}^{m} p_j = \sum_{j=1}^{m} q_j = \alpha \quad \text{and} \quad \max_{j=1}^{m} \{q_j\} \leqq 2 \min_{j=1}^{m}\text{-}r \{q_j\}.$$

Then

$$\sum_{j=1}^{m} (q_j)^2 \leqq \alpha^2 \int_0^m q_*^2(t)\, dt. \tag{16}$$

Moreover

$$\sum_{j=1}^{m} (p_j)^2 \geqq \frac{\alpha^2}{m}. \tag{17}$$

Hence from (16) and (17) we have

$$\frac{\sum_{j=1}^{m} (q_j)^2}{\sum_{j=1}^{m} (p_j)^2} \leqq m \int_0^m q_*^2(t)\, dt. \tag{18}$$

From (18) and Definition 2 of K_r^m we then have

$$K_r^m \leqq m \int_0^m q_*^2(t)\, dt. \tag{19}$$

Next we will show that $q_*(t)$ must have the following form:

$$q_*(t) = \begin{cases} 2x & 0 \leqq t < k \\ x & k \leqq t \leqq m-r+1 \\ 0 & m-r+1 < t \leqq m \end{cases} \tag{20}$$

for some values of k and x. It is clear that $q_*(t)=0$ for $m-r+1<t\leqq m$. Otherwise, we could form another step function $q_{**}(t)$ satisfying (13) and (14) but with

$$\int_0^m q_{**}^2(t)\, dt > \int_0^m q_*^2(t)\, dt$$

as shown in Fig. 4. Furthermore, it is clear that $q_*(t)$ cannot have one single step for $0\leqq t\leqq m-r+1$, as shown in Fig. 5. Moreover, $q_*(t)$ cannot have three or more steps for $0\leqq t\leqq m-r+1$, as shown in Fig. 6. This leaves (20) as the only form that $q_*(t)$ can assume.

From (14) and (20) we see that

$$\int_0^m q_*(t)\, dt = 2xk + x(m-r-k+1) = 1.$$

Hence $k=1/x-(m-r+1)$ and

$$\begin{aligned} \int_0^m q_*^2(t)\, dt &= 4x^2(1/x-(m-r+1)) + x^2\left[(m-r+1) - \left(\frac{1}{x} - (m-r+1)\right)\right] \\ &= 3x - 2(m-r+1)x^2. \end{aligned} \tag{21}$$

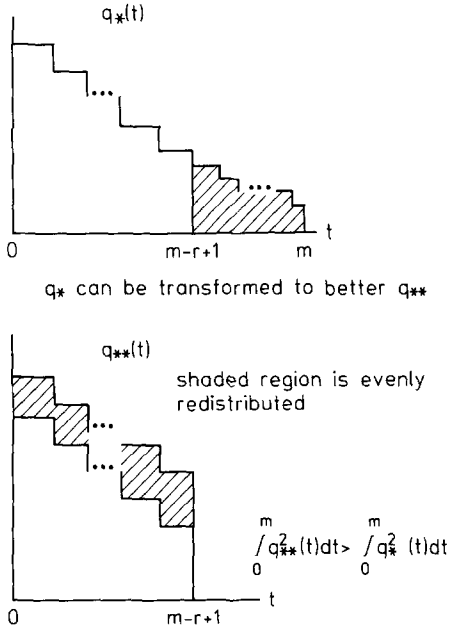

Fig. 4. $q_*(t)$ must be zero for $t > m-r+1$

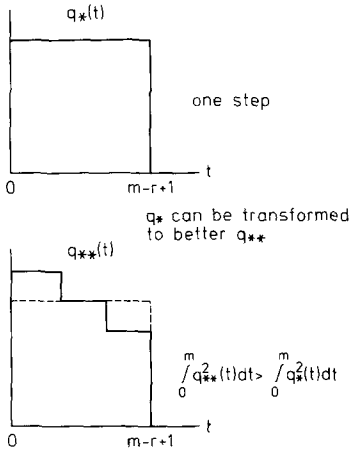

Fig. 5. $q_*(t)$ cannot have one step

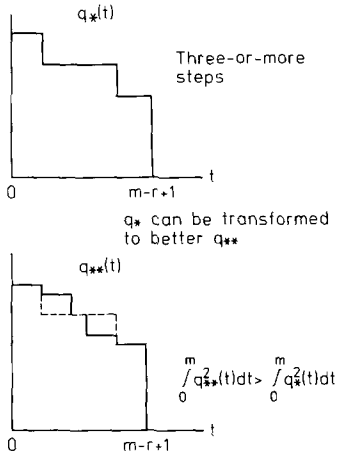

Fig. 6. $q_*(t)$ cannot have three or more steps

Using the derivative of (21) to solve for the maximum we find that $\int_0^m q_*^2(t)\,dt$ attains a maximum of $9/(8(m-r+1))$ when $x=3/(4(m-r+1))$, i.e.

$$\int_0^m q_*^2(t)\,dt = \frac{9}{8}\,\frac{1}{m-r+1}. \tag{22}$$

Using (19) and (22) we have

$$K_r^m \leqq \frac{9}{8}\,\frac{m}{m-r+1}. \quad \#$$

The main result follows.

Theorem 1. *Suppose that r clones of n tasks are to be assigned to m processors so that distinct clones of any task are assigned to distinct processors. Let* $q_1, \ldots, q_m$ *be the processor utilizations resulting from algorithm Allocate, and let* $q_1^*, \ldots, q_m^*$ *be the processor utilizations in an optimal assignment. Then*

$$\frac{\sum_{j=1}^{m} (q_j)^2}{\sum_{j=1}^{m} (q_j^*)^2} \leqq \frac{9m}{8(m-r+1)}.$$

Proof. The theorem follows directly from Lemma 6 and Lemma 7. #

It can be seen that the upper bound of Theorem 1 is not tight. For instance, in the case when $r=m$ algorithm Allocate produces optimal assignments (all processors are equally utilized), whereas Theorem 1 gives a very weak performance guarantee that is an increasing function of the number of processors ($9m/8$). Moreover, Theorem 1 gives a performance bound of 9/8 when $r=1$, but a better bound of 25/24 is known [3].

The bound derived in [3] may be used to improve Theorem 1. In the case when r evenly divides m, algorithm Allocate can be seen to produce assignments that are identical to those produced by the placement algorithm analyzed in [3]. We then have the following strengthening of Theorem 1:

$$\frac{\sum_{j=1}^{m} (q_j)^2}{\sum_{j=1}^{m} (q_j^*)^2} \leqq \begin{cases} \dfrac{25}{24} & \text{if } r \text{ divides } m, \\ \dfrac{9}{8}\,\dfrac{m}{m-r+1} & \text{if } r \text{ does not divide } m. \end{cases}$$

5. Empirical Results with the Approximation Algorithm

Algorithm Allocate was implemented as a Pascal program and run with data adapted from [16]. The results of this experiment are described in the following.

The design of the SIFT computer was preceded by a feasibility study that sought to characterize the application (digital flight control) in quantitative terms. This study proposed a logical partitioning of the application into a number of canonical tasks and established computational requirements for these tasks. The computational requirements for these tasks consisted of estimates of how often the task had to be executed (the task's frequency), approximately how many operations are executed by the task per second, and how much memory space is required by the task. All data was stated with respect to a hypothetical "benchmark" processor. From this data it is possible to derive input parameters for the TAP. This data is shown in Table 1. The data is stated with respect to a basic processor with an instruction execution rate of 0.5 MIPS and an address space of 64 Kbytes. We allocated groups of six, seven and eight processors to triplicated versions of the tasks shown in Table 1.

Table 1. Avionic task characteristics

Task No.	Task Descrip.	Iterations per Sec.	Instructions per Iteration	Utilization	Mem.
1	Attitude Control	20	1,228	0.04912	2,075
2	Flutter Control	250	138	0.06900	92
3	Gust Control	240	58	0.02784	60
4	Autoland	160	342	0.10944	1,025
5	Autopilot	5	200	0.00200	250
6	Attitude Director	30	2,560	0.15360	1,310
7	Inertial Navigation	25	1,350	0.06750	2,250
8	VOR/DME	5	770	0.00700	300
9	Omega	5	800	0.00800	505
10	Air Data	5	200	0.00200	135
11	Signal Processing	0.2	1,750	0.00070	315
12	Flight Data	5	5,520	0.05520	550
13	Airspeed	16	549	0.01757	430
14	Graphics Display	8	3,975	0.06360	6,250
15	Text Display	10	1,900	0.03800	9,340
16	Collision Avoidance	670	32	0.04288	1,150
17	Onboard Communication	250	28	0.01400	705
18	Offboard Communication	4	155	0.00124	687
19	Data Integration	4	360	0.00288	1,300
20	Instrumentation	5	2,792	0.02792	1,900
21	System Management	0.5	2,320	0.00232	950
22	Life Support	0.5	2,320	0.00232	950
23	Engine Control	33	3,597	0.23740	1,500
24	Executive	5	200	0.00200	1,100

The results of applying algorithm Allocate to the task set are shown in Tables 2, 3 and 4. Algorithm Allocate produced nearly perfectly balanced processor utilizations, and with eight digits of precision the approximate and optimal sums of squares are identical. Since no attempt was made to balance memory usage, memory balance leaves much to be desired.

Table 2. Utilizations of six processors

Processor	Utilization	Memory
1	0.50194	22,579
2	0.50194	22,579
3	0.50194	22,579
4	0.50227	12,550
5	0.50227	12,550
6	0.50227	12,550

Table 3. Utilizations of seven processors

Processor	Utilization	Memory
1	0.43011	15,657
2	0.43047	18,835
3	0.43053	22,712
4	0.43037	14,082
5	0.43014	8,067
6	0.43073	8,212
7	0.43031	17,822

Table 4. Utilization of eight processors

Processor	Utilization	Memory
1	0.37614	20,090
2	0.37658	5,332
3	0.37658	5,447
4	0.37605	8,110
5	0.37656	13,179
6	0.37685	15,800
7	0.37702	16,577
8	0.37688	20,852

6. Summary and conclusions

This paper has presented a TAP for certain fault-tolerant real-time distributed systems, the SIFT-like system. The TAP captures several important elements of these systems: separation constraints for replicated modules, capacity constraints imposed by limited local processor memory, scheduling constraints needed to ensure that the allocation will support the scheduling of periodic real-time tasks, and load balancing for reliability enhancement.

We have also presented a technique for the solution of the TAP. This approach involved the use of a fast algorithm that produces good but non-optimal allocations. We have shown that the allocations produced by this algorithm have normalized coefficients of variation that are guaranteed to be

less than $9m/(8(m-r+1))$ times that of the optimal allocation, where m is the number of processors and r is the number of times each task is replicated. For situations where m is large in relation to r (as is normally the case), this indicates very reasonable performance, with the performance ratio tending toward 1.125.

Acknowledgement. We would like to thank Robert Geist, Erol Gelenbe, and a referee for his comments.

References

1. Bokhari, S.H.: Dual Processor Scheduling with Dynamic Reassignment. IEEE Trans. Software Engrg. SE-**5**, 341–349 (1979)
2. Bryant, R.M., Agre, J.R.: A Queueing Network Approach to the Module Allocation Problem in Distributed Systems. Performance Evaluation Review **10**, 191–204 (1981)
3. Chandra, A., Wong, C.K.: Worst Case Analysis of a Placement Algorithm Related to Storage Allocation. SIAM J. Comput. **4**, 249–263 (1975)
4. Chou, T.C.K., Abraham, J.A.: Load Balancing in Distributed Systems. IEEE Trans. Software Engrg. SE-**8**, 401–412 (1982)
5. Chu, W.W., Holloway, L.J., Lan, M.-T., Efe, K.: Task Allocation in Distributed Data Processing. IEEE Comput. **13**, 57–69 (1980)
6. Dhall, S.K., Liu, C.L.: On a Real-Time Scheduling Problem. Operations Research **26**, 127–140 (1978)
7. Efe, K.: Heuristic Models of Task Assignment and Scheduling in Distributed Systems. Computer **15**, 50–56 (1982)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, W.H. (ed.). San Francisco 1979
9. Geist, R.M., Trivedi, K.S.: Optimal Design of Multilevel Storage Hierarchies. IEEE Trans. Comput. C-**31**, 249–260 (1982)
10. Gylys, V.B., Edwards, J.A.: Optimal Partitioning of Workload for Distributed Systems. Digest of Papers. COMPCON **76**, 353–357 (1976)
11. Ignizio, J.P., Palmer, D.F., Murphy, C.M.: A Multicriteria Approach to Supersystem Architecture Definition. IEEE Trans. Comput. C-**31**, 410–418 (1982)
12. Leung, J.Y-T., Whitehead, J.: On the Complexity of Fixed-Priority Scheduling of Real-Time Tasks. Proceedings of the Eighteenth Annual Allerton Conference on Communication. Control and Computing, pp. 464–470, 1980
13. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM **20**, 46–61 (1973)
14. Ma, P.-Y.R., Lee, E.Y.S., Tsuchiya, M.: A Task Allocation Model for Distributed Computing Systems. IEEE Trans. Comput. C-**31**, 41–47 (1982)
15. Rao, G.S., Stone, H.S., Hu, T.C.: Assignment of Tasks in a Distributed Proceesor System with Limited Memory. IEEE Trans. Comput. C-**28**, 291–299 (1979)
16. Ratner, R.S., Shapiro, E.B., Zeidler, H.M., Wahlstrom, S.E., Clark, C.B., Goldberg, J.: Design of a Fault Tolerant Airborne Digital Computer, vol. 2: Computational Requirements and Technology. SRI Final Report, NASA Contract NAS1-10920, 1973
17. Siewiorek, D.P., Gordon Bell, C., Newell, A.: Computer Structures: Principles and Examples. New York: McGraw-Hill 1982
18. Stone, H.S.: Multiprocessor Scheduling with the Aid of Network Flow Algorithms. IEEE Trans. Software Engrg. SE-**3**, 85–93 (1977)
19. Stone, H.S., Bokhari, S.H.: Control of Distributed Processes. Computer **11**, 97–106 (1978)
20. Trivedi, K.S., Wagner, R.A., Sigmon, T.M.: Optimal Selection of CPU Speed. Device Capacities, and File Assignments. JACM **27**, 457–473 (1980)
21. Uhrig, J.L.: Mathematical Programming Approaches to System Partitioning. IEEE Trans. Syst. Man, Cybernetics SMC-**8**, 540–548 (1978)

22. Wagner, R.A., Trivedi, K.S.: Hardware Configuration Selection Through Discretizing a Continuous Variable Solution. In: Proc. 7th IFIP Int. Symp. Comp. Performance Modeling, Measurement, and Evaluation. Toronto, Canada, pp. 127–142, 1980
23. Weinstock, C.B.: SIFT: System Design and Implementation. Proc. Tenth International Symposium Fault Tolerant Computing, pp. 75–77, 1980
24. Wensley, J.H., Goldberg, J., Green, M.W., Kautz, W.H., Levitt, K.N., Mills, M.E., Shostak, R.E., Whiting-O'Keefe, P.M., Zeidler, H.M.: Design Study of Software-Implemented Fault-Tolerance (SIFT) Computer. SRI Interim Technical Report 1, NASA Contract NAS1-13792, 1978
25. Wensley, J.H., Lamport, L., Goldberg, J., Green, M., Levitt, K.N., Melliar-Smith, P.M., Shostak, R., Weinstock, C.B.: SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. Proc. IEEE **66**, 1240–1255 (1978)

Received September 21, 1982/May 13, 1983