THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

ON THE DESIGN AND ANALYSIS OF REAL-TIME COMPUTERS

C. M. Krishna

CRL-TR-37-84

September 1984

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

ON THE DESIGN AND ANALYSIS OF REAL-TIME COMPUTERS

by

C. M. Krishna

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
1984

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PART I

INTRODUCTION

CHAPTER 1

INTRODUCTION

There are indications that progress toward higher chip density, lower cost, and greater reliability of microprocessors and memories will continue into the next decade. This trend naturally leads to the design of faster and more reliable multiprocessors than their uniprocessor counterpart. However, use of multiple microprocessors to speed up general-purpose computations requires the solution of such important problems as task partitioning, interconnection/intercommunication, synchronization, reliability, I/O interface and handling, software structure and programmability, etc. The efficacy of the multiprocessor depends crucially on the application tasks that it executes, and no single multiprocessor can at present embody the optimal solution to the above issues for general-purpose computations. Consequently, it has been the tendency to develop special-purpose multiprocessors. One such example is real-time multiprocessors whose primary function is control of critical real-time systems, e.g. aircraft, spacecraft, nuclear reactor, power distribution and monitoring, etc. Use of multiple processors/memories for real-time control is motivated by its potential for high operating speed and improved reliability through component multiplicity [1], [2].

A real-time control system comprises two components: a *controlled process* and a *control computer*. Despite their synergistic relationship, these two components have been designed and analyzed separately in isolation: the former by control scientists and the latter by computer designers. Moreover, control computer (called for brevity the controller) design has usually relied on ad hoc/empirical methods whereas there has been a significant progress in the theory and design of controlled processes. In order to narrow this gap and provide a bridge between these two components, this dissertation considers the controller with the controlled processes taken into account.

A computer controller has three communicating functions: *data acquisition, data processing* and *output* functions. The data acquisition is responsible for gathering input (feedback) data from sensors, input panels and other associated equipment; the processing function done by the computer (in our case the multiprocessor) generates output control/display signals from input data and the output function sends the processed results to mechanical actuators, displays and other output devices. The system may thus logically be regarded as a three-stage pipe.

The controller software in the processing section consists of a set of *tasks,* each of which corresponds to some job to be performed repetitively in response to particular sets of environmental stimuli (triggers). These include both regular task triggers according to a predetermined schedule, as well as unexpected, situation-dependent, task triggers. The set of tasks to be executed by the controller is predetermined and the stochastic nature and behavior of the software known in advance -- at least in outline -- to the designer. This fact makes it both easier and more necessary to obtain a reasonably good performance analysis of the system.

The determining characteristic of a real-time multiprocessor's performance is a combination of reliability and high throughput. The throughput requirements arise from the need for quick system response to environmental stimuli. Speed is of the essence in a real-time controller since failure can occur not only through massive hardware failures in the system, but also on account of the system's not responding quickly enough to events in the environment.

As a result of these special performance requirements, performance measures used to characterize general-purpose uniprocessor systems are no longer appropriate for real-time multiprocessors. Conventional throughput, reliability, and availability by themselves alone have little meaning in the context of control; a suitable combination of these is necessary. New performance measures are required: measures that are congruent to the application, permit the expression of specifications that reflect without contortion true system characteristics and application requirements, in addition to allowing an objective comparison of rival systems for particular applications.

We cannot stress too heavily that it is meaningless to speak of the performance of a computer out of the context of its application. The form the performance measures take must reflect the needs of the application, and the computer system must be modelled within this context. The multiprocessor controller and the controlled process form a synergistic pair, and any effort to study the one must take account of the needs of the other.

It is important that performance measures should depend on variables that can be definitively estimated or objectively measured. It is our policy in this dissertation, therefore, to always base performance indices on experimentally-measurable quantities: controller response times for the various system tasks.

It must also be realized that there is a distinction to be drawn between the measurement of performance parameters and their interpretation. In parameter measurement, we are concerned, for example, with the ease and accuracy with which the parameter can be measured. On the other hand, interpretation consists of a procedure to integrate the results of the measurement into a complete picture of the computer's performance. Different parameter values (reliability, throughput, etc.) can depend on one another in a quite complex way: we do not have the luxury of assuming that they are independent of each other. We have either to present computer performance as a vector (which makes comparison between different systems difficult) or to derive an objective metric for the performance vector. It is the purpose of our research to develop one such metric and then use it for the design and analysis of real-time computers.

Performance measures that partially meet real-time requirements have been suggested by the following authors. Beaudry [3] considers measures emanating from the volume of computation from a computer system over a given period of operation. Mine and Hatayama [4] consider *job-related reliability,* by which they mean the probability that the system will successfully complete a certain job. Huslende [5] attempts to be as general as possible, and presents what amounts to a re-statement of Markov modeling with traditional measures. Chou and Abraham [6] present performance-availability models, Castillo and Siewiorek [7] performance-reliability models. Osaki and Nishio [8] consider the "reliability of information", by which they mean the expected percentage of wrong outputs per unit time in steady state.

All these measures consider the computer system in isolation, i.e. without explicit regard to the requirements of the operating environment. For this reason, they are quite unsuitable for use in real-time control situations.

Among all existing performance measures, Meyer's ***performability*** [9] seems to meet, though in an abstract form, the real-time requirements discussed above. His measure explicitly links the application with the computer by listing "accomplishment levels", which are expressions of how well the computer has performed within the context of the application. His work focuses on the development of a framework for modeling and performance evaluation, rather than on methodology for deriving the performance measures themselves. No guidelines are given for appropriately specifying the accomplishment levels: what is provided is a set of mathematical tools for their computation, once they have been defined. Meyer's more recent work continues this trend, developing the theory of stochastic Petri nets.

By contrast we focus, in part of this dissertation, on presenting a methodology for objectively characterizing and determining controller performance. If one wished to translate our work into the terms of Meyer's performability, we show how to derive a set of uncountably many accomplishment levels that are completely objective and capable of definitive estimation and/or measurement. It is this that makes our measures complementary to that of Meyer.

The next step is to apply the performance measures to design and analysis of real-time computers. We consider problems of of trading off the number of processors in a system against their processing speed, of reliable synchronization and fault-location, and of generating fault-tolerant schedules for real-time tasks.

This dissertation is organized as follows. In Chapter 2, controlled systems are discussed, and Chapter 3 introduces our performance measures. Chapter 4 contains two examples to show how to determine the performance measures: one is an idealized motion control problem and the other is a more realistic example, the aircraft landing problem. Chapter 5 to 7 are allied problems in real-time computer design, in which the use of our measures is demonstrated. In Chapter 5, we introduce and consider the number-power tradeoff. In Chapter 6, synchronization and fault-masking in real-time systems are considered. In Chapter 7, we provide an algorithm for the generation of fault-tolerant schedules. The dissertation concludes with Chapter 8.

CHAPTER 2

REAL-TIME SYSTEMS

Figure 1 shows the block diagram of a typical real-time control system.

The inputs to the control computer are from sensors that provide data about the controlled process, and from the environment. This is typically fed to the control computer at regular intervals. Data rates are usually low: generally fewer than 20 words a second for each sensor. The job list represents the fact that all the control software is pre-determined and partitioned into individual jobs.

Central to the operation of the system is the trigger generator that initiates execution of one or more of the control programs. In most systems, this is physically part of the controller itself, but we separate them here for purposes of clarity. Triggers can be classed into three categories.

(1) *Time-generated trigger:* These are generated at regular intervals, and lead to the corresponding controller job(s) being initiated at regular intervals. In control-theoretic terms, these are open-loop triggers.

(2) *State-generated trigger:* These are closed-loop triggers, generated whenever the system is in a particular set of states. A simple example is a thermostat that switches on or off according to the ambient temperature. For practicality, it might be necessary to space these triggers by more than a specified minimum

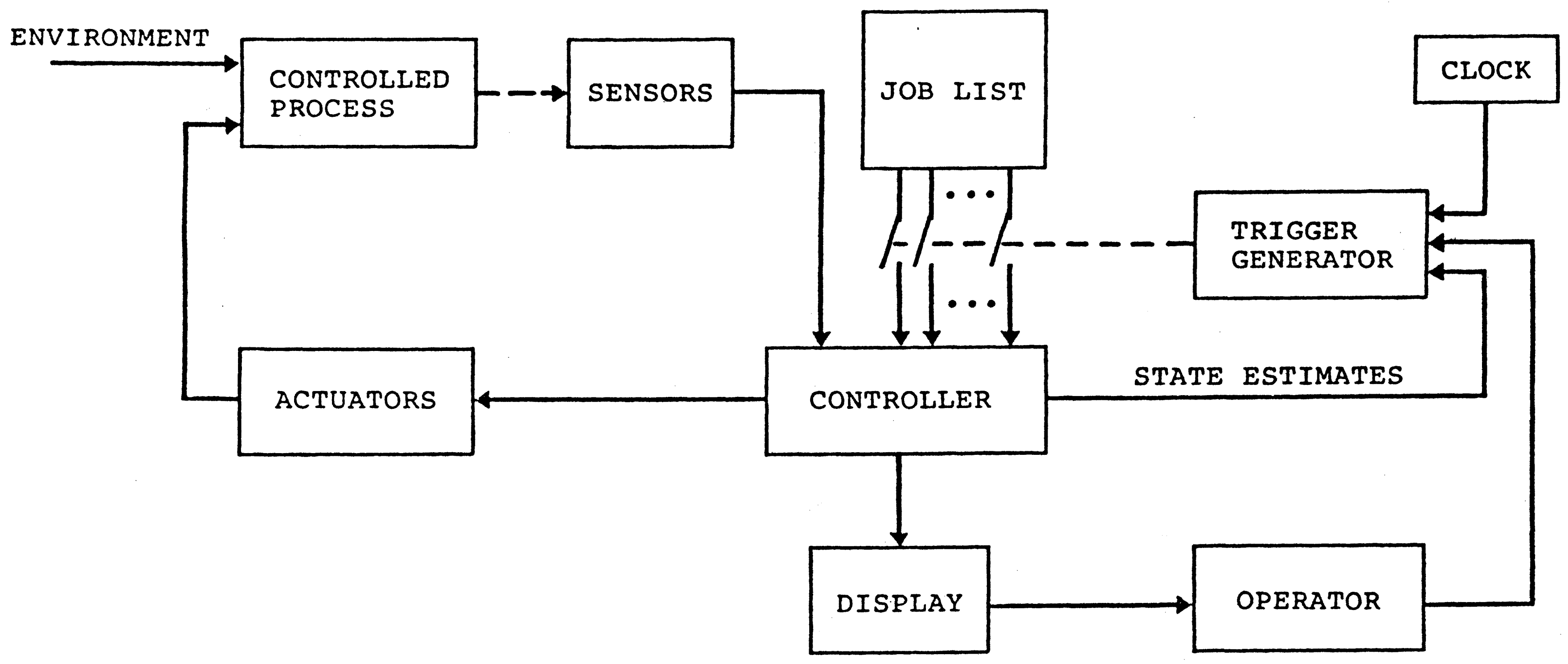


Figure 1. A Typical Real-Time Control System.

duration. If time is to be regarded as an implicit state variable, the time-generated trigger is a special case of the state-generated trigger. One can also have combinations of the two.

(3) *Operator-generated trigger:* The operator can generally over-ride the automatic systems, generating and cancelling triggers at will.

The output of the controller is fed to the actuators and/or the display panel(s). Since the actuators are mechanical devices and the displays are meant as a human interface, the data rates here are usually very low. Indeed, a control-computer system generally exhibits a fundamental dichotomy from many points of view. Firstly, the I/O is carried out at rather low rates (the only exceptions to this that we know of are control systems that depend on real-time image-processing: such applications have extremely high input data rates), and the computations have to be carried out at very high rates owing to real-time constraints on control. Secondly, the complexity of the data processing carried out at the sensors and the actuators is much less than that carried out in the main data-processing area. Thirdly, the sensors, actuators, and the associated equipment are entirely dedicated to the performance of a particular set of tasks, while the hardware in the region where the complex data processing takes place is usually not dedicated.

It is therefore possible to logically partition real-time computer systems into *central* and *peripheral* areas. The peripheral area consists of the sensors, actuators, displays, and the associated processing elements used for the pre-processing and formatting of data that is to be put into the central area, and the "unpacking" of data that are put out by the central area to the actuators and/or displays. The central area consists of the processors and associated hardware where all the higher-level computation takes place. Designing the peripheral area is relatively straightforward; the most difficult design problems that arise in these systems usually concern the central area. Figure 2 and Table 1 emphasize these points.

A control system executes "missions." These are periods of operation between successive periods of maintenance. In the case of aircraft, a mission is usually a single flight. The operating interval can sometimes be divided down into consecutive sections that can be distinguished from each other. These sections are called *phases*. For example, Meyer *et al.* [11] define the following four distinct phases in the mission lifetime of a civilian aircraft:

(a) Takeoff/cruise until VHF Omnirange (VOR)/Distance Measuring Equipment (DME) out of range.

(b) Cruise until VOR/DME in range again.

(c) Cruise until landing is to be initiated.

(d) Landing.

The current phase of the controller partially determines its job load, job mix, job priorities, and so on.

A real-time system typically has to function under more constraints than does its general-purpose counterpart. Firstly, there are *hard deadlines,* which if missed, can lead to catastrophic failure. Timing is therefore crucial to job execution. Secondly, there are physical constraints that are not quite so restricting for the general-purpose computer. Examples are weight and power consumption.

The applications software has the following properties.

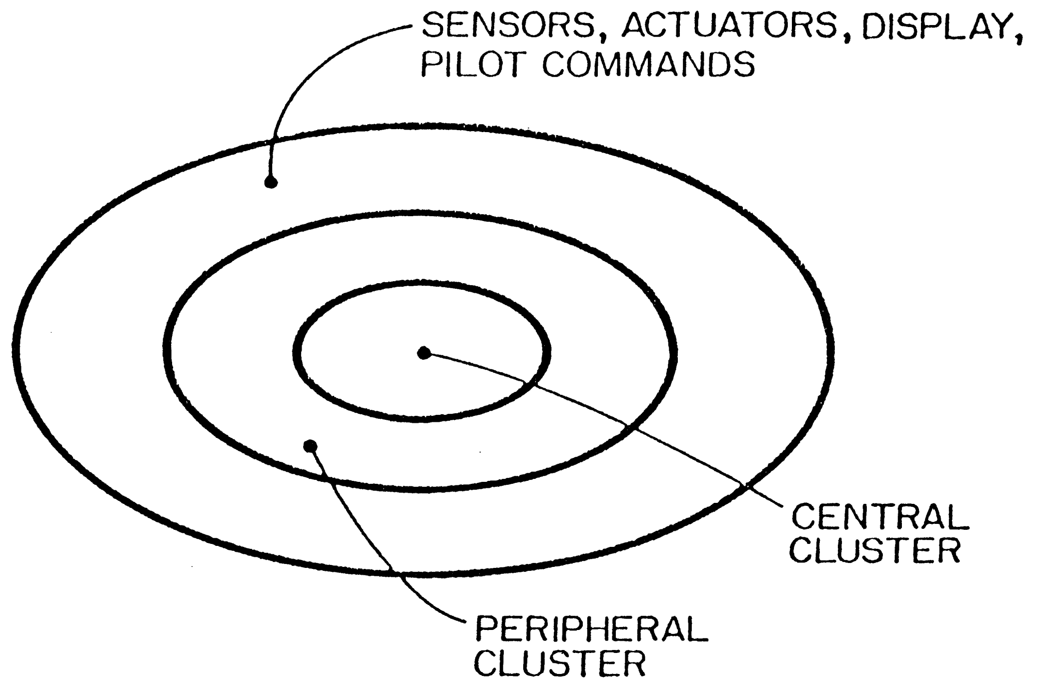


Figure 2. Schematic Decomposition of a Real-Time Control Computer.

Peripheral Area	**Central Area**
Low baud rates	High baud rates
Complete dedication	Complete generality of function
Low-capability processors	High-capability processors
Simple interconnection structure	Complex interconnection structure
Almost totally decoupled processors	Processors highly coupled in many cases
Trivial executive software	Complex executive software

Table 1. Difference between Central and Peripheral Areas.

(1) The interaction between individual processes is minimal.

(2) The effects of (computing) processes upon one another is well understood.

(3) Clear lines of authority are recognized.

(4) Clear lines of information flow are recognized.

(5) The products of the (computing) processes are well defined.

These are precisely the five conditions for efficiency in a distributed system as listed by Fox [12]. Because of this and also due to their potentially high reliability, distributed systems are particularly suited to real-time use. Also, the problems that arise when one attempts to partition programs in general-purpose applications for implementation on a distributed computer do not usually arise in the real-time context. The software for a control computer is not so much a single partitionable package, as a set of cleanly interacting subroutines. Macro-instruction languages show much promise in this context [13].

The constraints on real-time systems, as well as the properties of the applications software, have a very great influence on the system architecture and the executive software.

The overall computer system has to be much more reliable than any of its components, so that fault-tolerance is essential. Massive replication of hardware is commonplace, as also are high interconnection-link bandwidths. The system must be as symmetric as possible so that reconfiguration is easy.

The nature of the executive software must reflect constraints on time and resources. The executive is responsible for the control of queues at shared resources, for the scheduling of events, for the handling of interrupts, and the allocation of memory. While all these tasks are common to general-purpose systems, the existence of hard deadlines makes the efficient execution of such activities imperative. The designer of the real-time system does not have the luxury of assuming that occasional serious degradation of performance is acceptable, if unfortunate.

An additional important task of the executive is fault-handling and recovery. This includes reconfiguration where that is possible, and the rescheduling of tasks upon processor failure. Here again, the constraints on time make this a difficult problem.

PART II

THE PERFORMANCE MEASURES

CHAPTER 3

THE PERFORMANCE MEASURES

A. Terminology and Notation

A real-time computer executes pre-defined control jobs repeatedly, upon environmental or other stimuli. A *job* is a well-defined stretch of software, e.g., a subroutine. Each job maps into one or more *tasks*. The mapping is determined by the current state of the controlled process. This is further clarified later in this Chapter. *System response time* is defined as the time between the initiation/triggering of a control job and the actuator and/or display output that results. This quantity is the sum of *controller response time* and *actuation time*. Environmental or other occurrences trigger the tasks, a unique *version* being created as a result. This is said to be an *extant version* as long as it continues to execute in the system. Versions of task i are denoted by V_{ij}, which represents the j-th execution of task i. Denote the response time of a no-longer-extant version V_{ij} by RESP(V_{ij}). The response time of an extant version is undefined. The *extant time* of a version V_{ij} triggered at time τ_{ij} when the system is in state n_{ij} is given by $\Xi(V_{ij},\tau_{ij},n_{ij},t) = \min(t-\tau_{ij},\ RESP(V_{ij}))$. Note, however, that this does *not* imply that no state changes occur during the course of a task execution. $RESP(V_{ij})$ is an implicit function of n_{ij}.

A controller task is said to be *critical* if it has an associated finite *hard deadline* [14], which if exceeded by any of its versions, results in catastrophic or *dynamic* failure. Hard deadlines do not exist for *non-critical* tasks; although in some cases, it might be convenient to pretend they exist and set them at infinity.

Ordinarily, repair to the real-time computer is not allowed while the computer is in operation. In this connection, we define the *mission lifetime* as the duration of operation between successive stages of service. We let the mission lifetime be a random variable with probability distribution function $L(t)$. At the beginning of a mission (i.e. immediately after service), a system is assumed to be free of faults.

B. Definition of the Performance Measures

Our performance measures are all based on the extant and response times. For critical tasks i, with hard deadlines t_{di}, the cost function is defined by:

$$C_i(\Xi) \equiv \begin{cases} g_i(\Xi) & \text{if } 0 \leq \Xi \leq t_{di} \\ \infty & \text{if } \Xi > t_{di} \end{cases} \tag{1}$$

where g_i is called the *finite cost function* of task i, and is only defined in the interval $[0, t_{di}]$, and we omit for notational convenience the arguments of Ξ, the extant time of the task.

For non-critical tasks, the same definition for the cost function can be used, with the associated hard deadline set at infinity.

Let $q_i(t)$ denote the number of times task i is initiated in the interval $[0,t)$. Then, the ***cumulative cost function*** for the task i is defined as

$$\Gamma_i(t) \equiv \sum_{j=1}^{q_i(t)} C_i(\Xi(V_{ij},\tau_{ij},n_{ij},t)) \tag{2}$$

and the ***system cost function*** is defined as

$$S(t) \equiv \sum_{i=1}^{r} \Gamma_i(t) \tag{3}$$

where r is the number of tasks in the system. Both Γ_i and S are clearly defective random variables. Our performance measures are then given by:

$$\textit{Cost Index, } K(\chi) \equiv \int_0^\infty Prob\{S(t) \leq \chi\}\, dL(t) \tag{4}$$

$$\textit{Probability of dynamic failure, } p_{dyn} \equiv \int_0^\infty Prob\{S(t) = \infty\}\, dL(t) \tag{5}$$

$$\textit{Mean Cost, } M \equiv \int_0^\infty E\,\{S(t) \mid \textit{no hard deadlines are missed}\}\, dL(t) \tag{6}$$

$$\textit{Variance Cost, } V \equiv \int_0^\infty Var\,\{S(t) \mid \textit{no hard deadlines are missed}\}\, dL(t) \tag{7}$$

where $E\{\bullet|\bullet\}$ and $Var\{\bullet|\bullet\}$ represent conditional expectation and variance, respectively. The probability of dynamic failure subsumes the traditional probability of failure (called here for distinction the ***probability of static failure***) since the latter can be viewed as the probability that the expected system response time is infinity. Clearly, in the case of non-critical tasks, the probabilities of static and of dynamic failure are equal.

The following auxiliary measures are useful when one focuses on the contribution to the cost of individual tasks.

$$\textit{Cost Index for task } i,\ K_i(\chi) \equiv \int_0^\infty Prob\{\Gamma_i(t) \leq \chi\}\, dL(t) \tag{4a}$$

$$\textit{Mean Cost for task } i,\ M_i \equiv \int_0^\infty E\,\{\Gamma_i(t) \mid \textit{no hard deadlines are missed}\}\, dL(t) \tag{5a}$$

$$\textit{Variance Cost for task } i,\ V_i \equiv \int_0^\infty Var\,\{\Gamma_i(t) \mid \textit{no hard deadlines are missed}\}\, dL(t) \tag{7a}$$

The computation of these measures can sometimes be complicated by the fact that the mission might end while one or more versions are still extant. In most instances, however, the mission lifetimes are very much longer than individual task execution times -- for example, several hours for aircraft and several days or even months for spacecraft -- and the number of times tasks are executed to completion before the mission ends is also very large. For this reason, it is usually an acceptable approximation to compute the costs assuming that all jobs that enter the system during the mission complete executing before the mission ends (as long as they do not miss any hard deadlines).

In what follows, we consider how to determine these performance measures, beginning with the determination of the hard deadline.

C. Obtaining the Hard Deadline

The dynamics and the nature of the operating environment of the critical process are both known *a priori*. This follows from the critical nature of the process -- for example, the dynamics and operating environment of aircraft have both been studied carefully -- and advances in the theory and design of controlled processes.

The process can most conveniently be expressed by a state-space model. Let $\mathbf{x} \in \mathbf{R}^n$ denote the process state, $\mathbf{u} \in \mathbf{R}^m$ the input vector, and t the time. The input vector is made up of two sub-vectors, $\mathbf{u}_c \in \mathbf{R}^{m_c}$ and $\mathbf{u}_e \in \mathbf{R}^{m_e}$. $\mathbf{u}_c$ denotes the input delivered at the command of the computer, and $\mathbf{u}_e$ the input generated and then applied by the operating environment. We characterize state transitions by the mapping $\phi: T \times T \times \mathbf{X} \times U \rightarrow \mathbf{X}$ where $T \subset \mathbf{R}$ represents time, $\mathbf{X} \subset \mathbf{R}^n$ the state-space and $\mathbf{U} \subset \mathbf{R}^m$ the input space.

$$\mathbf{x}(t) = \phi\ (t, t_0, \mathbf{x}(t_0), \mathbf{u}) \tag{9a}$$

Measurement of the system is described by a vector $\mathbf{y} \in \mathbf{R}^l$ and a mapping $\eta: \mathbf{X} \times \mathbf{U} \times T$.

$$\mathbf{y}(t) = \eta\ (\mathbf{x}(t), \mathbf{u}(t), t) \tag{9b}$$

Catastrophic failure can follow if the process leaves the "safe" region of the state space. For example, a boiler may explode if its temperature becomes too high. This is formally expressed by defining an *allowed state space, $\mathbf{X}_a(t)$*, which defines the "safe" region of operation.

The task of the controller or real-time computer is to derive the optimal control, $\mathbf{u}_c(t)$, as a function of the perceived process state. Since the response time, denoted by ω, is positive, we have $\mathbf{u}_c(t) = \mathbf{h}(\mathbf{x}(t-\omega), \mathbf{u}_e(t-\omega), t)$ where $\mathbf{h}$ expresses the control algorithm for the task in question. Then, the hard deadline associated with this task is given by the maximum value of ω that may be permitted if the process is to remain in $\mathbf{X}_a$ with probability one. More precisely, the hard deadline associated with controller task α triggered at t_0 when the system is in state $\mathbf{x}(t_0)$ is given by:

$$t_{d\alpha}(\mathbf{x}(t_0)) \equiv \inf_{\mathbf{u} \in \Omega \subset \mathbf{U}} \sup\{\tau \mid \phi(t_0+\tau, t_0, \mathbf{x}(t_0), \mathbf{u}) \in \mathbf{X}_a\} \tag{10}$$

where Ω is the admissible input space. One can also define *conditional hard deadlines* if it is only required to perform the computation over a certain subset of the admissible input or state space. The conditional hard deadline of task α, denoted by $t_{d\alpha|\omega,\sigma}$, is defined as

$$t_{d\alpha|\omega,\sigma}(\mathbf{x}(t_0)) \equiv \inf_{\mathbf{u}\in\omega\subset\Omega} sup\{\tau \mid \phi(t_0+\tau,t_0,\mathbf{x}(t_0),\mathbf{u})\in\sigma\subset\mathbf{X}_a\} \tag{11}$$

The hard deadline, defined in this general way, is a function both of the process state at the moment of task initiation, and of time. It is a random variable if the environment is stochastic. Consider the following example to see how the hard deadline can be determined.

Example 1: A body of mass m is constrained to move in one dimension. Its state-vector consists of three components: position (x_1), velocity (x_2), and acceleration (x_3). The allowed state-space is defined by $\mathbf{X}_a=\{\mathbf{x} \mid |x_1|\leq b,\ x_2<\infty,\ x_3<\infty\}$ where $b>0$ is a constant. The body is subject to impact from either direction with equal probability. Each impact changes the velocity of the body by $k>0$ units, in the appropriate direction. The change of velocity takes place in a negligible duration. The body has devices that can exert thrust of magnitude H in either direction. This thrust is imposed only after the controller has recognized an impact and has determined how to react to it. It takes a negligible amount of time to switch the thrust on or off in either direction. The controller's job is to bring the body to $\mathbf{x}=0$. The controller operates in open-loop: when it recognizes an impact, it computes the thrusts as a function of time, following which the control response is assumed to be instantaneous. The problem now is to compute the hard deadline associated with this task.

The hard deadline is only a function of the state and $\mathbf{X}_a$. In computing it, we do not need to take into account the possibility of a second impact before the controller has finished responding to the first, since the state of the process contains all necessary information.

The allowed state space is static and simply connected, so that if when the body is brought to rest for the first time following the impact it is in $\mathbf{X}_a$, it must have been within $\mathbf{X}_a$ throughout the period following the impact, assuming only that it was within $\mathbf{X}_a$ at the moment of impact. Therefore, we have only to compute the position of the body when it first comes to rest (after the impact) as a function of the response time, ξ, and set the hard deadline equal to the greatest ξ for which the body comes to rest within $\mathbf{X}_a$. Let the initial state of the body be $\mathbf{x}_i^T=[x_{1_i}, x_{2_i}, x_{3_i}]$. Since the impact duration and the switch-off or on time for the thrust are assumed to be zero, we can always take $x_{3_i}=0$. Define

$$t_1 = \left| \frac{m}{H}[x_{2_i}+k] \right|, \qquad t_2 = \left| \frac{m}{H}[x_{2_i}-k] \right|$$

By an elementary derivation, we arrive at the following.

Case 1: $x_{2_i}<-k$:

$$t_d(\mathbf{x}_i)=\min\left\{ \frac{b + x_{1_i} + (x_{2_i}+k)t_1 + \dfrac{Ht_1^2}{2m}}{-(x_{2_i} + k)}, \frac{b + x_{1_i} + (x_{2_i}-k)t_2 + \dfrac{Ht_2^2}{2m}}{k - x_{2_i}} \right\} \tag{12}$$

For future convenience, denote the right hand side of the above by $t^{(1)}$.

Case 2: $|x_{2_i}|<k$:

$$t_d(\mathbf{x}_i)=\min\left\{ \frac{b - x_{1_i} - (x_{2_i}+k)t_1 + \dfrac{Ht_1^2}{2m}}{x_{2_i} + k}, \frac{b + x_{1_i} + (x_{2_i}-k)t_2 + \dfrac{Ht_2^2}{2m}}{k - x_{2_i}} \right\} \tag{13}$$

Denote the right hand side of the above equation by $t^{(2)}$.

Case 3: $x_{2i}>k$:

$$t_d(\mathbf{x}_i)=\min\left\{\frac{b-x_{1i}-(x_{2i}+k)t_1+\frac{Ht_1^2}{2m}}{x_{2i}+k},\ \frac{b-x_{1i}-(x_{2i}-k)t_2+\frac{Ht_2^2}{2m}}{x_{2i}-k}\right\} \tag{14}$$

Denote the right hand side of the above equation by $t^{(3)}$.

If the velocity imparted to the body upon an impact is not constant at k, but is a random variable (which would be more realistic) the magnitude of which has probability distribution function F_{impact}, then the hard deadline will be a random variable, whose distribution is a function of the state at the moment of impact. The following can be written down by inspection:

Case 1, $x_{2i}<0$:

$$t_d(\mathbf{x}_i)=\begin{cases} t^{(2)} & \textit{with probability } F_{impact}(|x_{2i}|) \\ t^{(1)} & \textit{with probability } 1-F_{impact}(|x_{2i}|) \end{cases} \tag{15}$$

Case 2, $x_{2i}>0$:

$$t_d(\mathbf{x}_i)=\begin{cases} t^{(2)} & \textit{with probability } 1-F_{impact}(|x_{2i}|) \\ t^{(3)} & \textit{with probability } F_{impact}(|x_{2i}|) \end{cases} \tag{16}$$

We could similarly treat the case when the allowed state-space is stochastic.

The above example is meant only to illustrate the hard deadlines and should not lull the reader into a false sense of security. Obtaining closed-form expressions for the deadlines of any but the most trivial systems and static allowed space is usually extremely difficult, if not impossible. For example, if we relax the assumption that the controller acts in open-loop, the equations of motion become too difficult to solve exactly in closed form.

It is generally necessary to resort to numerical methods to obtain deadlines for real-life systems. Since most of the state-spaces one uses in practice have uncountably many points, we must define hard deadlines as functions of sets of states, not of the states themselves, if the entire allowed state-space is to be covered. Subdividing the state-space into these subsets while keeping errors low is not always easy. For an example of subdivision where the application is the control of aircraft elevator deflections, see the case study that follows in Chapter 4.

A further remark is in order here. The hard deadlines are not dependent upon the performance functional (time, energy, etc.) that the controller is attempting to optimize, since the paramount duty of the controller is to keep the system within the allowed state-space, and only secondarily to optimize the performance functional.

D. Obtaining the Finite Cost Functions

Performance functionals have been known for a long time in control theory as optimization criteria and measures of controlled process performance. We exploit this fact to derive the control-computer cost functions, by linking directly the performance of the controller to the value of the controlled process performance functional that results.

Performance functionals in control theory are functionals of system state and input and express the cost of running the process over some interval $[t_0, t_f]$. The performance functionals can be stated as:

$$\Theta(\mathbf{x}_0, t_0, t_f) = \int_{t_0}^{t_f} E\,[\, f_0\,(\mathbf{x}(t), \mathbf{u}(t), \mathbf{x}_0, t) \mid \mathbf{y}(\tau), t_0 \leq \tau < t\,]dt \tag{17}$$

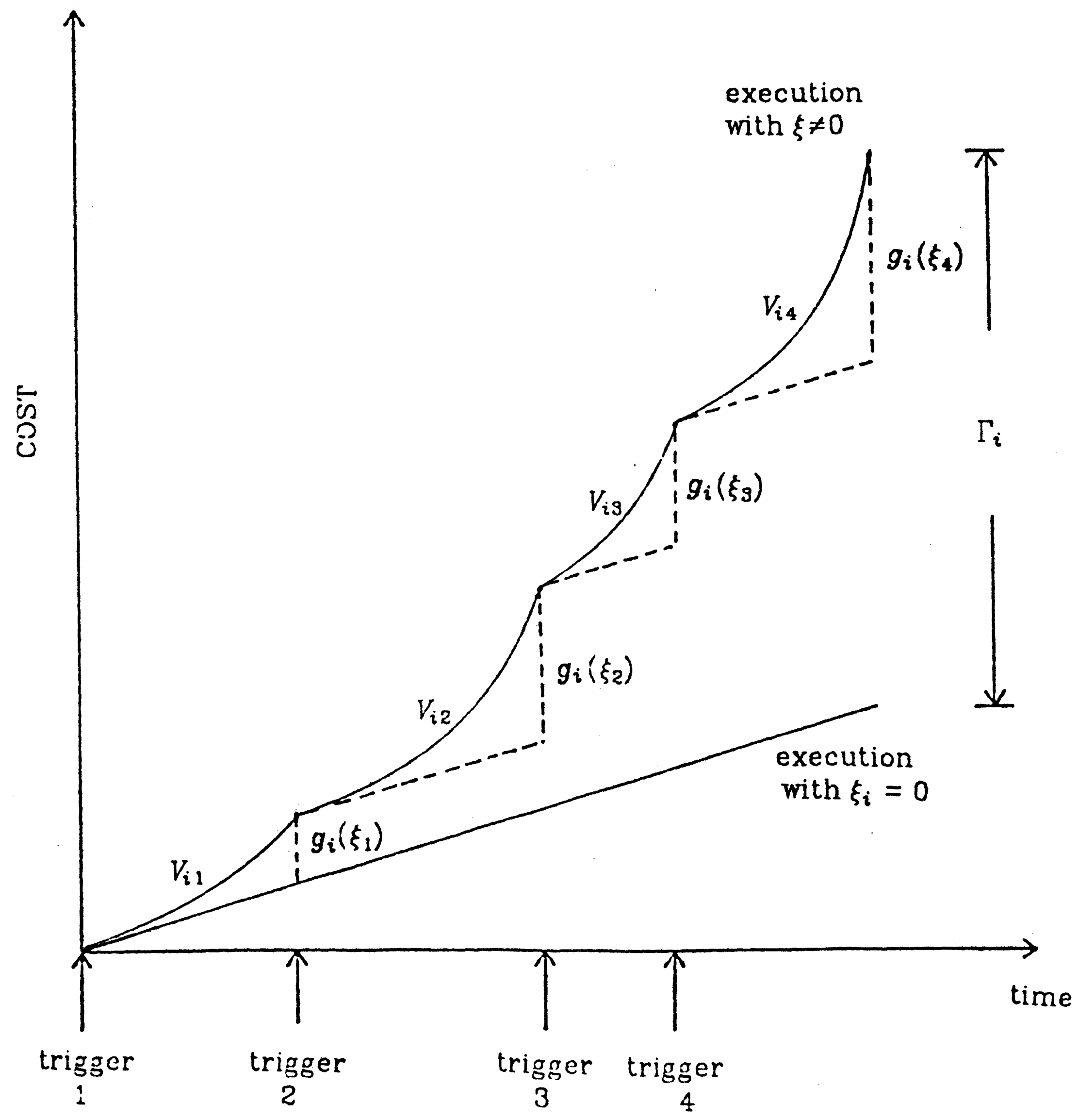
where $\mathbf{x}(t_0) = \mathbf{x}_0$, and f_0 is the instantaneous performance functional. Since the controller response time affects the state trajectory of the controlled process, it affects the performance functional as well. If we use the expected contribution to the performance functional, $\Theta(\mathbf{x}_0, t_0, t_f)$, of the control delivered as a result of executing task i with response time ξ, we can derive the finite cost function as:

$$g_i(\mathbf{x},\xi) = \begin{cases} \Omega(\mathbf{x},\xi) - \Omega(\mathbf{x},0) & \text{for } 0 \leq \xi \leq t_{di} \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

where $\Omega(\mathbf{x},\xi)$ denotes the contribution to $\Theta(\mathbf{x}_0, t_0, t_f)$ of a task with response time ξ, and initiated when the process state was $\mathbf{x}$. By doing so, we can directly couple the response time of the controller to the fuel, energy, or other commodity by the controlled process. See Figure 3.

Notice that while we use response time to compute the cost function, the finite costs were originally defined as functions of the extant time. The latter is the case since we wish costs to accrue as the execution proceeds, so that the system cost function is continuous. This ensures if two systems are compared under an identical load with the first faster than the second with regard to a particular task, that the faster system will never exhibit a mean cost greater than the slower system as long as both have approximately the same probability of dynamic failure.

The possibility of correlation of successive tasks can complicate calculations considerably. To see this, take the system in the example above. If a second impact comes in *before* the system has finished reacting to the first, then, the energy or



$\xi_j = RESP(V_{ij})$ for $j = 1,2,3,4$.

Figure 3. Illustration of Cost Functions.

time to be expended will not, in general, be the sum of the energies or the times that would have to be expended if the second impact had arrived *after* the system had finished reacting to the first (i.e. had arrived at $\mathbf{x}=0$). Assuming that successive tasks are decoupled leads to a certain measure of "double-counting" of the energy or time spent. The same remark would apply to fuel, force, or any other performance functional used for the controlled process.

Due to this double-counting, assuming that successive tasks are decoupled leads to an upper bound to the energy, time, or other quantity expended. If we find an upper bound acceptable, we can simplify our computations greatly. If exact figures are called for, a detailed and complicated model has to be worked out in which each instance of inter-task coupling is itemized and its probability of occurrence computed. Whether or not this is worth the effort depends entirely on the requirements of the analysis.

There is also an irritating anomaly. Since the mean costs are defined by an expectation that is conditioned on not failing in the mission lifetime, it is possible to construct pathological examples where a system with a probability of dynamic failure of, say, 0.5 over a given lifetime, will exhibit a *lower* mean cost over that lifetime than another that has a p_{dyn} of 10^{-10}: we shall see examples of them in Chapter 5. Such cases are, however, generally no more than an academic curiosity.

Example 2: Consider again the controlled process described in Example 1. This time, we set out to compute the finite cost function associated with the task under review.

As before, assume that the state of the body at the moment of impact is given by $\mathbf{x}^T=[x_{1i},x_{2i},x_{3i}]$. We make the assumption that a function that provides an upper bound of the cost expended is sufficient, so that it is not necessary to consider the correlative effect of successive jobs. We provide cost functions relating to two different control policies.

Case A: Assume that the duty of the controller is to bring the body back to $\mathbf{x}=0$ within as short a time period as possible. (Note that $\mathbf{x}=0$ means that all three components -- position, velocity, and acceleration -- are zero). The cost function is the time taken. This is the well-known minimum-time problem in optimal control theory [15]. If, after the impact, the body is moving away from $x_1=0$, it must be stopped, and brought back using bang-bang control. If it is moving toward $x_1=0$, depending on the velocity after impact and the response time of the controller, the body is either first accelerated toward $x_1=0$ and then decelerated, or first brought to a stop on the other side of $x_1=0$ and then brought back to the origin using bang-bang control. The derivation of the time taken is elementary, if tedious, and is excluded. See the Appendix for expressions of the finite cost function under such a control policy. The case when the velocity imparted upon impact is not constant, but a random variable, can be handled as in Example 1.

Case B: Suppose the controller is to minimize the energy expended while, after every impact, keeping the body within the allowed state-space. Then, the control policy is simply to bring the system to rest anywhere inside $\mathbf{X}_a$, and the cost function is in terms of energy. If the controller computer responds to an impact within the hard deadline, it can by definition, keep the system from failing. As may easily be verified, the energy expended in doing this is the energy required to bring the body to rest, which is equal to the energy of the body immediately after impact. Therefore, as long as the allowed state-space is not violated, the energy expended will remain the same no matter what the response time (assuming that the response time is within the hard deadlines derived in the preceding section). So, the finite cost function over the entire allowed state-space is here the zero function, which signifies that, as long as the hard deadlines are honored, it makes no difference to the overhead *under this control policy,* as to what the response time may be.

This example has served to emphasize the intimate relation between control policy and controller (finite) cost function. The *same* system, with the *same* constraints on the allowed state-space, has *different* cost functions based on what the duty of the controller is. It reflects our goal of having the cost functions express the control overhead *in the context of the application.* This, in fact, distinguishes our measures from those extant in the literature. See Table 2 for a comparison between our measures and those of others.

Once again, it should be noted that the simplicity of the above expository examples does not usually exist in real-life systems. Real-life analyses are much more difficult, and the same comments as applied to p_{dyn} above apply to the mean cost, also. There is also one additional complication. The controller is to optimize the performance functional subject to the condition that the system must not leave the allowed state-space, if this is at all possible. Such a difficulty did not arise in this simple example, but it can sometimes prove difficult to obtain optimal control policies under this requirement. This, however, is a problem for the designer of the controlled process, not of the controlling computer.

See Chapter 4 for a computation of the cost function in a realistic case.

Other Measures	**Our Measures**
Wide applicability to almost all applications of fault-tolerant, gracefully-degrading systems.	Limited Applicability. Aims specifically at real-time, especially at control, application.
Measures express performance in rather gross terms.	Measures express performance in rather exaet terms.
Performance linked to characteristics of the computer alone.	Performance measures specifically designed to reflect the overhead of the computer on the real-time system.

Table 2. Comparison of Traditional and New Methods of Characterizing Performance

E. Remark on Finite Cost Functions

It is not necessary that the process performance functional that is used to derive the cost function (e.g. energy, fuel, time, etc.) be the same as the process performance functional that the system is trying to optimize. For example, the system may be given the task of optimizing fuel, and the cost function may be measuring the extra energy consumed as a result of controller delay. However care should be taken to ensure that the two functionals (the one used to optimize the process, and the one used to express the cost with) do not conflict. For the functionals not to conflict, the optimal control actions (i.e. the controller decisions) taken on the basis of one functional should be identical to the optimal control actions that would have been taken on the basis of the other. For example, if the fuel consumed were linearly related to the energy expended, the cost function could be expressed in terms of energy, while the controller was trying to minimize the fuel used.

To see why conflicts must not be allowed, assume in the system of the above example, that the job of the controller is to minimize the time taken in bringing the body back to the origin, while the cost function is in terms of energy. Take the instance in which the speed of the body after the impact is a slow motion toward $x_1=0$. The controller should in such a case apply full thrust throughout the motion, first speeding the body up toward $x_1=0$, and then slowing it down to reach $\mathbf{x}=0$ in minimum time. However, since the cost function is in terms of energy, not time, it is easy to see that the shorter the time period over which the controller exerts thrust, the smaller is the value of the cost measured. Thus, over a certain range of states, the cost function would actually *decrease* with an increase in response time. This is not only counter-intuitive, but also results in inefficient operation. Task priorities, scheduling policies, etc., for the control computer are meant to be derived to optimize the mean cost as expressed by the cost functions. If inconsistencies such as the above arose, the operating system of the controller would tend to oppose the goals inherent in its own applications software, resulting in an unsatisfactory overall computer-controlled system.

F. Allowed State-Space and Its Decomposition

As we said above, it is difficult to determine the hard deadline and the finite cost function as a function of the state over the entire state space. The solution of the controlled process state equations cannot usually be obtained in closed form when controller delay is considered. To obtain the functional dependence of the hard deadlines or the finite cost function of each controller job on the current state vector is therefore impossible to do analytically, and prohibitively expensive to do numerically for a large number of sample states.

To get around this problem, we divide the allowed state-space down into *subspaces*. Subspaces are aggregates of states in which the system exhibits roughly the same behavior. Even if there do not exist clear boundaries for these subspaces, one can always force the allowed state space to be divided into subspaces so that a sufficient safety margin can be provided. This is a designer's choice for approximation. In each subspace, each critical controller job has a unique hard deadline.

Remark: In some subspaces, a job described in general as "critical" might not be critical in the sense that even if the execution delay associated with it is infinity, catastrophic failure does not occur. That is, the associated hard deadline may be infinity for a particular subspace. What *does* usually happen in these circumstances is that the system moves into a new subspace -- or at the least toward the subspace boundary -- in which the dangers of catastrophic failure are greater. In this subspace, the requirements on controller delay are more stringent, and there might well be a hard deadline, representing a critical task. Thus a "critical" job need not be truly critical in every subspace, it only has to map into a critical *task* -- defined in the sequel -- in at least one subspace. Also, subspaces are job-related, i.e. the same allowed state space can divide into a different set of subspaces for each control job.

For convenience, a controller "task" is defined as follows.

Definition: A controller task, often abbreviated to "task", is defined as a controller job operating within a designated subspace of the allowed state space.

Let S_i for i=0,1,...,s be disjoint subspaces of $\mathbf{X}_A$ with $\mathbf{X}_A = \bigcup_{i=1}^{s} S_i$ and let J denote a controller job. Then, we define the projection: $(J, \mathbf{X}_A) \rightarrow ((T_0, S_0), (T_1, S_1), \ldots, (T_s, S_s))$ where T_i is the controller task generated by executing J in S_i. With each controller task, we may now define a hard deadline without the coupling problem mentioned above. We denote it by $t_{d_i}^J$ for critical task T_i (for convenience, however, the superscript J will be omitted in the sequel). We will see that a critical job can possibly map into a non-critical task for one or more allowed subspace; it only needs to map into a critical task in *at least one* such subspace to be considered critical.

Allowed State-Space

The allowed state-space is the set of states that the system must not leave if catastrophic failure is not to occur. Consider the two sets of states $\mathbf{X}_A^1$ and $\mathbf{X}_A^2$ defined as follows.

(i) $\mathbf{X}_A^1$ is the set of states that the system must reside in if catastrophic failure is not to occur *immediately*. For example, we may define in the case of an aircraft, a situation in which the aircraft flies upside down as unacceptable to the passengers and as constituting failure. Notice that terminal constraints are not taken into consideration here unless the task in question is executed just prior to mission termination.

(ii) $\mathbf{X}_A^2$ is the set of acceptable states given the terminal constraints, i.e., it is the set of states from which, given the constraints on the control, it becomes possible to satisfy the terminal constraints.

Note that leaving $\mathbf{X}_A^1$ means that no matter how good our subsequent control, failure has occurred. (Strictly speaking, of course, there can be no subsequent control since by leaving $\mathbf{X}_A^1$ the system has failed catastrophically before the next control could be implemented.) On the other hand, altering the allowed input space, i.e. changing the control available can affect the set $\mathbf{X}_A^2$. The allowed state space is then defined as $\mathbf{X}_A \equiv \mathbf{X}_A^1 \cap \mathbf{X}_A^2$.

Obtaining state-space $\mathbf{X}_A^2$ can be difficult in practice. The curse of dimensionality ensures that even systems with four or five state variables make unacceptable demands on computation resources for the accurate determination of the allowed state-space. However, while it can be very difficult to obtain the entire allowed state-space, it is somewhat easier to obtain a reasonably large subset,

$\mathbf{X}_A^a \subset \mathbf{X}_A$. By defining this subset as the actual allowed state-space, (i.e., by artificially restricting the range of allowed states), we make a conservative estimate for the allowed state-space. Note that by making a conservative approximation, we err on the side of safety. Also, the information we need about $\mathbf{X}_A$ may be determined to as much precision as we are willing to invest in computing resources.

In what follows, to avoid needless pedantry, we shall refer to the artificially restricted allowed state-space, $\mathbf{X}_A^a$, simply as the "allowed state-space", $\mathbf{X}_A$.

On Obtaining the Subspaces

The job of dividing $\mathbf{X}_A$ into $\mathbf{S} = (\mathbf{S}_0, \mathbf{S}_1, ..., \mathbf{S}_s)$ is sometimes made easy by the existence of natural cleavages in the state-space, when the latter is viewed as an influence on system behavior. In most cases, however, such conveniences do not exist, and artificial means must be found. The problem then becomes one of finding discrete subdivisions of a continuum.

The method we employ is to quantize the state continuum in much the same way as analog signals are quantized into digital ones. Intervals of hard deadlines and expected operating cost (i.e. the mean of the cost function conditioned on the controller delay time, and using the distribution of the latter) are defined. Then, points are allocated to subspaces corresponding to these intervals. To take a concrete example, consider a state-space $\mathbf{X} \subset \mathbf{R}^n$ that is to be subdivided on the basis of the hard deadlines. The first step is to define a quantization for the hard deadlines. Let this be Δ. Then, define subspace $\mathbf{S}_i$ as containing all states in which the hard deadline lies in the interval $[(i-1)\Delta, i\Delta)$. Alternatively, one might define a sequence of numbers $\Delta_1, \Delta_2, ...$, such that the subspaces were defined by intervals with the $\Delta' s$ as their end-points. This would correspond to quantizing with variable step sizes. The subspace in which the job under consideration maps into a non-critical task is a special case and is denoted by $\mathbf{S}_0$.

Subspaces can also be defined based on a quantization of the expected operating cost or on both the operating cost and the hard deadlines. We provide an example of subdivision by hard deadlines in Chapter 4.

Computational Complexity

When closed-form solutions are available for the state equations, they can be solved backwards from the terminal constraints to obtain analytical results for $\mathbf{X}_A^2$. When closed-form solutions are not available, numerical techniques must be used, and solving differential equations backwards can give rise to numerical instability. In the most general case, the amount of computations required increases rapidly with the number of state variables and the complexity of the state equations; therefore, the exact amount of required computations depends on the solution approach employed and the actual system under consideration.

One begins by deriving the optimal trajectory, which requires the problem- and solution- specific amount of computations. Then, obtaining $\mathbf{X}_A^2$ consists of searching the n-dimensional state-space surrounding this trajectory. In general, this becomes an exhaustive search for the boundary of $\mathbf{X}_A^2$. If the search is confined to an $(n+1)$ dimensional parallelopiped (the $n+1$-th dimension represents time) with edge lengths $k_1,k_2,\ldots,k_{n+1}$, and the digital resolution is Δ, then defining $m_i=\lceil k_i/\Delta\rceil$, $i=1,\ldots,n+1$, the state equations must be solved $O(\prod_{i=1}^{n+1} m_i)$ times. The values for k_i must be set by the user.

In the event that $\mathbf{X}_A^2$ is known to be convex, a binary search can be instituted along one of the dimensions, reducing the search complexity to $O(m_1 \cdots m_{j-1}m_{j+1} \cdots m_{n+1} \log_2 m_j)$ where $m_j = \max\{m_1, \cdots, m_n\}$.

As for the cost functions and the hard deadlines, these must also be derived, in general, by an exhaustive analysis. The allowed state-space is quantized, and deadlines and cost functions must be found for each of the resulting "cells".

It is therefore clear that for a process with large n, these methods become prohibitively expensive. Two comments are now in order. The first is that this approach is not uniquely susceptible to the curse of dimensionality: all optimal control theory techniques suffer this problem when complex processes are considered. The second comment is that, in certain cases, additional knowledge about the process dynamics can be used to improve the efficiency of the search for the $\mathbf{X}_A^2$ boundary, and for the hard deadlines and cost functions.

G. A Final Remark on this Chapter

The relevance of the finite cost functions and of the performance measures -- mean and variance cost -- that are derived from them should be obvious. What is not immediately apparent is the practical difference between the traditional probability of failure and the probability of dynamic failure introduced here.

The standard, devil's advocate, objection runs as follows. Just as fault-tolerant systems are designed with considerable physical redundancy, so they can be designed also to allow considerable time redundancy. This would entail, for example, a task having a worst-case run time (given no environmental interference) of, say, a fifth or a tenth of the interval between its release and its deadline. Moreover, the tasks are usually executed many -- five or even ten -- times for every output that is required at the actuator. In such a case, the argument runs, the probability of dynamic failure is as close to the traditional probability of failure as makes no matter.

The fallacy in this argument arises from the lack of a fixed worst-case run-time for tasks. Environmental influences can on rare occasions, prolong the execution time of tasks almost indefinitely. A good example of this is lightning strikes on aircraft which cause electro-magnetic interference. This appears only recently to have been seriously studied as a source of disruption to avionic electronics. Unfortunately, since preliminary measurements are protected from unlimited disclosure, we have to fall back on a hypothetical example, with hypothetical results.

Assume that the prolongation of the execution time caused by lightning follows an exponential law, with a mean of 0.1 second. This is a good figure to choose: if lightning were, on the average, to have a much smaller duration, it would only rarely be noticed. Let the probability of a lightning strike on an aircraft during a flight of 10 hours' duration be 10^{-3}. Imagine a computer using 7-modular redundancy, with processor MTBF's being 10,000 hours, and consider a task that takes 0.01 second to execute under normal circumstances. and which is run continuously in a loop that ends only with the mission. The task is run ten times in a second. Failure will occur if not even one out of ten consecutive runs produces a result within 0.1 second of task release. Denoting the probability that a given processor fails over the ten hour flight by p, the traditional probability of failure -- which is the probability that more than 3 of the 7 processors fail during the 10 hours of flight -- is given by $\sum_{i=4}^{7}\binom{7}{i}p^i(1-p)^{7-i}$. With the numerical values as assumed above, this is equal to about 3.5×10^{-11}. The probability of dynamic failure, however, is dominated by the probability of a lightning strike delaying response beyond the hard deadline, and is lower bounded by the probability of a lightning strike lasting more than one second. Under the parameter values assumed above, this quantity is 4.5×10^{-8}, i.e. more than a thousand times the traditional probability of failure, and 45 times the NASA benchmark of 10^{-9}. Furthermore, the fact that the slack time was so large a multiple of the run time, made little difference to the probability of failure.

The data used above are purely hypothetical (although credible from a common-sense perspective), so that no absolute inferences may be drawn from these calculations, but the point that we wish to make should be clear: when ultra-reliable systems are being considered, *all* conceivable sources of failure must be considered, not just the traditional ones of massive hardware failure. Traditional reliability computations do not indicate any need to consider the effects of lightning in this example. Designers attempt to compensate for this oversight by making the task execute many times for each output that is absolutely required. Unfortunately, as we saw in this example, such attempts are not always viable when ultra-reliable systems are being considered.

CHAPTER 4

CASE STUDY

A control system executes "missions." These are periods of operation between successive periods of maintenance. In the case of aircraft, a mission is usually a single flight. The operating interval can sometimes be divided down into consecutive sections that can be distinguished from each other. These sections are called *phases*. As pointed out in Section 2, Meyer *et al.* [11] define four distinct phases in the mission lifetime of a civilian aircraft. The phase to be considered here is landing, it takes about 20 seconds. The controller job that we shall treat is the control of the aircraft elevator deflection during landing.

The specific system employed is assumed to be organized as shown in Figure 4. Sensors report on the four key parameters: altitude, descent rate, pitch angle, and pitch angle rate every 60 milli-seconds. We have a time-generated trigger, with a time period of 60 milli-seconds. Every 60 milli-seconds, the controller computes the optimal setting for the elevator, which is the only actuator used in the landing phase. (There are other actuators used aboard the aircraft for purposes of stability, horizontal speed control, etc. We do not however consider them here, concentrating exclusively on the control of the elevator.)

The execution time for the computation is nominally 20 milli-seconds, although this can vary in practice due to failures. Since the aircraft is a dynamical system,

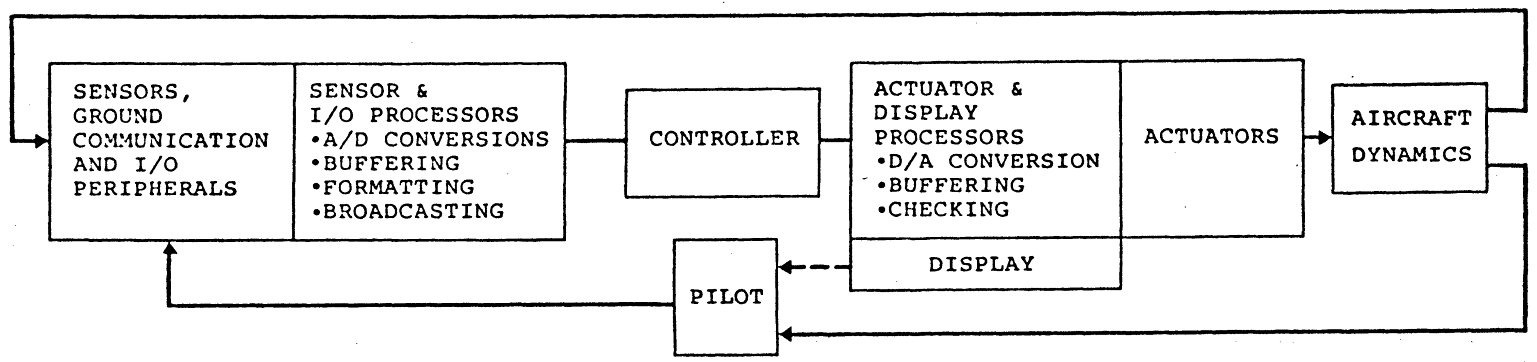


Figure 4. Aircraft Control System Schematic.

the effects of controller delay are considerable -- as we shall see in this Section.

Since the process being controlled is critical (i.e. in which some failures can lead to catastrophic consequences), variations of controller delay and other abnormal behavior by the controller must be explicitly considered. For simplicity, we do not allow job pipelining in the controller; in other words a controller job must be completed or abandoned before its successor can be initiated. The following controller abnormalities can occur:

(i) The controller orders an incorrect output to the actuator.

(ii) The controller takes substantially more than 20 milli-seconds (the nominal execution time) but less than the inter-trigger interval of 60 milli-seconds to complete executing.

(iii) The controller takes more than 60 milli-seconds to complete executing. In such a case, the abnormal job is abandoned and the new one initiated. We say that a control trigger is "missed" when this happens.

An analysis of controller performance during the landing phase must take each of the above abnormalities into account.

A. The Controlled Process

The model and the optimal control solution used are due to Ellert and Merriam [16].

The aircraft dynamics are characterized by the equations:

$$\dot{x}_1(t) = b_{11}x_1(t)+b_{12}x_2(t)+b_{13}x_3(t)+c_{11}m_1(t,\xi) \qquad \textbf{(19a)}$$

$$\dot{x}_2(t) = x_1(t) \qquad (19b)$$

$$\dot{x}_3(t) = b_{32}x_2(t)+b_{33}x_3(t) \qquad (19c)$$

$$\dot{x}_4(t) = x_3(t) \qquad (19d)$$

where x_2 is the pitch angle, x_1 the pitch angle rate, x_3 the altitude rate, and x_4 the altitude. m_1 denotes the elevator deflection, which is the sole control employed. The constants b_{ij} and c_{11} are given in Table 3. Recall that ξ denotes controller response time.

The phase of landing takes about 20 seconds. Initially, the aircraft is at an altitude of 100 feet, travelling at a horizontal speed of 256 feet/sec. This latter velocity is assumed to be held constant over the entire landing interval. The rate of ascent at the beginning of this phase is -20 feet/sec. The pitch angle is ideally to be held constant at 2 °. Also, the motion of the elevator is restricted by mechanical stops. It is constrained to be between -35 ° and 15 °. For linear operation, the elevator may not operate against the elevator stops for nonzero periods of time during this phase. Saturation effects are not considered. Also not considered are wind gusts and other random environmental effects.

The constraints are as follows: The pitch angle must lie between 0 ° and 10 ° to avoid landing on the nose-wheel or on the tail, and the angle of attack (see Figure 5) must be held to less than 18 ° to avoid stalling. The vertical speed with which the aircraft touches down must be less than around 2 feet/sec so that the undercarriage can withstand the force of landing.

The desired altitude trajectory is given by

Feedback Term	**Value**
b_{11}	-0.600
b_{12}	-0.760
b_{13}	0.003
b_{32}	102.4
b_{33}	-0.4
c_{11}	-2.374

Table 3. Feedback Equation Constant

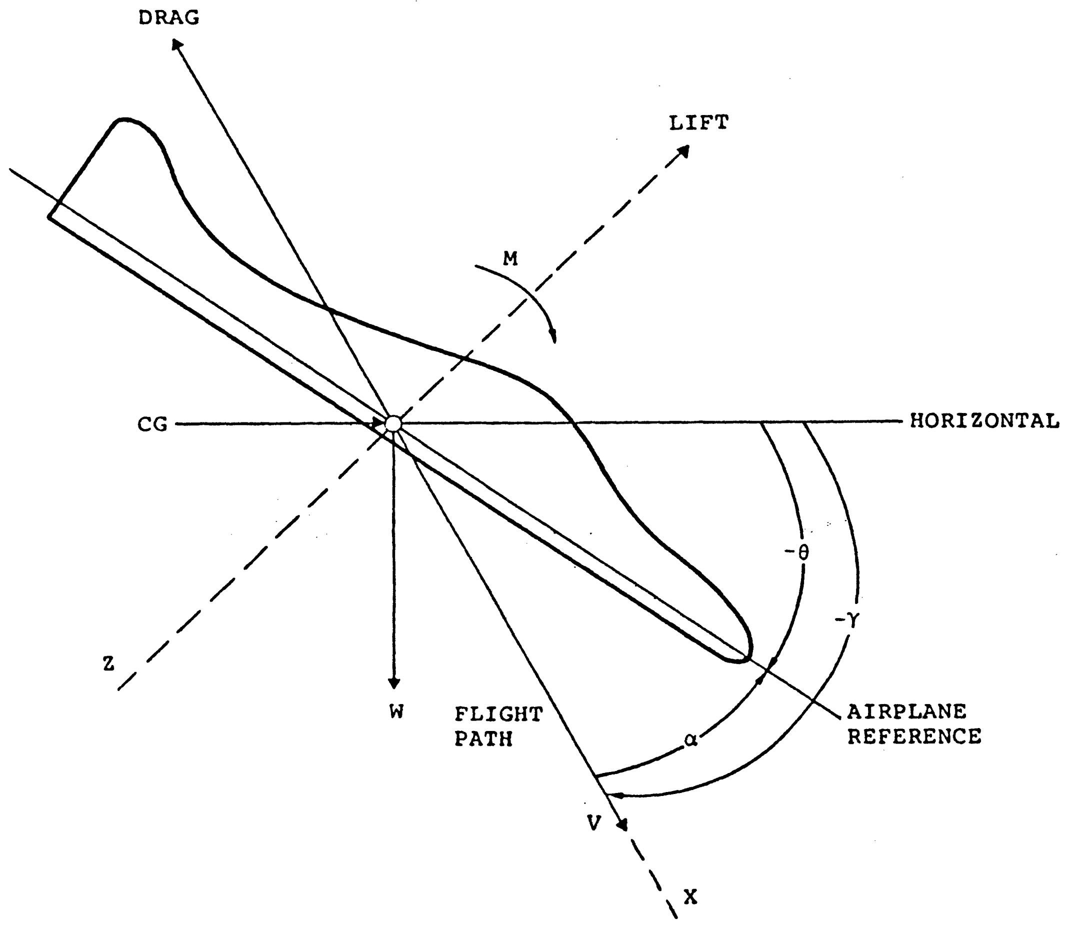


Figure 5. Definition of Aircraft Angles

$$h_d(t) = \begin{cases} 100e^{-t/5} & 0 \leq t \leq 15 \\ 20-t & 15 \leq t \leq 20 \end{cases} \tag{20}$$

while the desired rate of ascent is

$$\dot{h}_d(t) = \begin{cases} -20e^{-t/5} & 0 \leq t \leq 15 \\ -1 & 15 \leq t \leq 20 \end{cases} \tag{21}$$

The desired pitch angle is 2 ° and the desired pitch angle rate is 0 ° per sec.

The performance index (for the aircraft) chosen by Ellert and Merriam and suitably adapted here to take account of the nonzero controller response time ξ is given by

$$\Theta(\xi) = \int_{t_0}^{t_f} e_m(t,\xi)dt \tag{22}$$

where t represents time, and $[t_0,\ t_f]$ is the interval under consideration, and where

$$e_m(t,\xi) = \phi_h(t)[h_d(t)-x_4(t)]^2+\phi_{\dot{h}}(t)[\dot{h}_d(t)-x_3(t)]^2+\phi_\theta(t)[x_{2d}(t)-x_2(t)]^2$$
$$+\phi_{\dot{\theta}}(t)\ [x_{1d}(t)-x_1(t)]^2+[m_1(t,\xi)]^2. \tag{23}$$

where the d-subscripts denote the desired (i.e. ideal) trajectory. To ensure that the touch-down conditions are met, the weights ϕ must be impulse weighted. Thus we define:

$$\phi_h(t) = \phi_4(t) + \phi_{4,t_f}\delta(20-t) \tag{23a}$$

$$\phi_{\dot{h}}(t) = \phi_3(t) + \phi_{3,t_f}\delta(20-t) \tag{23b}$$

$$\phi_\theta(t) = \phi_{2,t_f}(t)\delta(20-t) \tag{23c}$$

$$\phi_{\dot{\theta}}(t) = \phi_1(t) \tag{23d}$$

where the functions ϕ must be given suitable values, and δ denotes the Dirac-delta function. The values of the ϕ are given based on a study of the trajectory that results. The chosen values are listed in Table 4.

The control law for the elevator deflection is given by:

$$m_1(t,\xi) = \omega_s^2 K_s T_s [k_{11}(t-\xi) - k_{11}(t-\xi)x_1(t-\xi) - k_{12}(t-\xi)x_2(t-\xi) \\ - k_{13}(t-\xi)x_3(t-\xi) - k_{14}(t-\xi)x_4(t-\xi)] \tag{24}$$

where the aircraft parameters are given by: $K_s = -0.95\ \mathrm{sec}^{-1}$, $T_s = 2.5$ sec, $\omega_s = 1$ *radian* sec^{-1} and the constants k are the feedback parameters derived (as shown in [16]) by solving the Riccatian differential equations that result upon minimizing the process performance index. For these differential equations we refer the reader to [16].

B Derivation of Performance Measures

We consider here only one controller task: that of computing the elevator deflection so as to follow the desired landing trajectory. The inputs for the controller here are the sensed values of the four states.

We seek the following information. As the controller delay increases, how much extra overhead is added to the performance index? Also, it is intuitively obvious that too great a delay will lead to a violation of the terminal (landing) conditions, thus resulting in a plane crash. This corresponds to dynamic failure, and we are naturally interested in determining the range of controller delays that permit a safe landing.

Consider first a formal treatment of the problem. The control problem is of the linear feedback form. The state equations can be expressed as:

Weighting Factor	**Value**
$\phi_1(t)$	99.0
$\phi_{2,t_f}(t)$	20.0
$\phi_3(t)$ $(0 \leq t < 15)$ $\phi_3(t)$ $(15 \leq t \leq 20)$ ϕ_{3,t_f}	0.0 0.0001 1.000
ϕ_4 ϕ_{4,t_f}	0.00005 0.001

Table 4. Weights for the Performance Index

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{25}$$

where the symbols have their traditional meanings. Define the feedback matrix by $\Xi(t)$. Then, clearly,

$$\mathbf{u}(t) = \Xi(t-\xi)\mathbf{x}(t-\xi) \tag{26}$$

For a small controller delay (i.e., a small ξ), the above can be expanded in a Taylor series and the terms of second order and higher discarded for a linear approximation. By carrying out the obvious mathematical steps, we arrive at the equation:

$$\dot{\mathbf{x}}(t) = \mathbf{E}(t,\xi)\mathbf{x}(t) + \beta(\xi) \tag{27}$$

as representing the behavior of the controlled process, assuming that the initial conditions are given. For further details, see Figure 6.

Given a closed-form expression for the $k_{ij}(t)$ that appear in $\mathbf{E}(t,\xi)$, we could then proceed to study the characteristics of the system as a function of the matrix E. However, in the absence of such closed formulations for the k_{ij}, we must take recourse to the less elegant medium of numerical solution.

The procedures we follow for obtaining the numerical solution are as follows. First, the feedback values are computed by solving the feedback differential equations that define the k_{ij}. These are not affected by the magnitude of the controller delay. Then, the state equations are solved as simultaneous differential equations. These are used to check that the terminal constraints have been satisfied, and in the event that they are the performance functional is evaluated. This procedure must be repeated for each new subspace. Since the environment is deterministic in this case (no wind gusts or other random disturbances are permitted in the model), the hard deadline associated with each process subspace is a constant and not a random variable.

$$E(t,\xi) = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & 1 & 0 & 0 \\ 0 & b_{32} & b_{33} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

where

$$a_{11} = [1-c_{11}^2 k_{11}(t)\xi]^{-1}\ [b_{11}-k_{11}(t)c_{11}^2-c_{11}^2\xi\{\phi_{\dot{\theta}}(t)+2b_{11}k_{11}(t)+k_{12}(t)-c_{11}^2 k_{11}^2(t)\}]$$

$$a_{12} = [1-c_{11}^2 k_{11}(t)\xi]^{-1}\ [b_{12}-c_{11}^2 k_{12}(t)-c_{11}^2\xi\{b_{11}k_{12}(t)+b_{12}k_{11}(t)+k_{22}(t)-c_{11}^2 k_{11}^2(t)\}]$$

$$a_{13} = [1-c_{11}^2 k_{11}(t)\xi]^{-1}\ [b_{13}-c_{11}^2 k_{13}(t)+b_{13}k_{11}(t)+k_{23}(t)-c_{11}^2 k_{11}(t)k_{13}(t)\}]$$

$$a_{14} = [1-c_{11}^2 k_{11}(t)\xi]^{-1}\ [-k_{14}(t)-b_{11}k_{14}(t)\xi-k_{24}(t)\xi+c_{11}^2 k_{11}(t)k_{14}(t)\xi]$$

b_{32} = Horizontal Velocity/T,

$b_{33} = -1/T$,

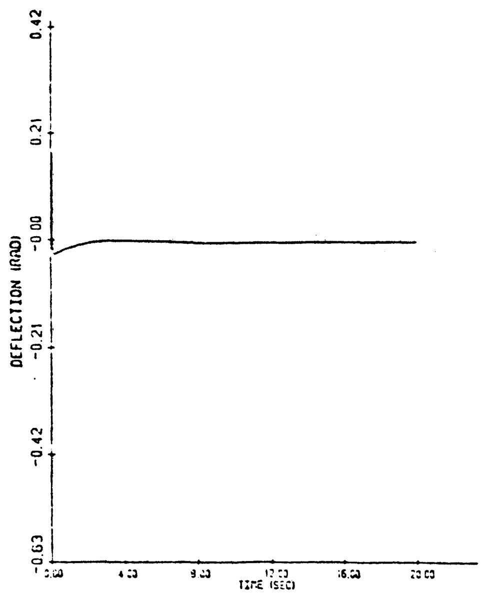
When the execution delay is ξ, the approximate state equations are

$$\dot{x}(t) = E(t,\xi)x(t) + \begin{bmatrix} c_{11}^2[k_1(t)+\xi\{\phi_{\dot{\theta}}(t)\theta_d(t)+b_{11}k_1(t)+k_2(t)-c_{11}^2 k_1(t)k_{11}(t)\}] \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

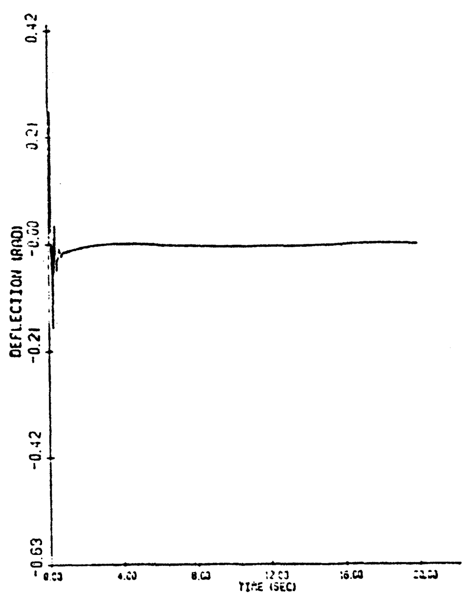
Figure 6. The Approximate State Equations.

The trajectory followed by the aircraft when the delay is less than about 60 milli-seconds follows the optimal trajectory closely, although the elevator deflections required would be intuitively assumed to increase as the delay increases. Also, the susceptibility of the process to failure in the presence of incorrect or no input is expected to rise with the introduction of random environmental effects.

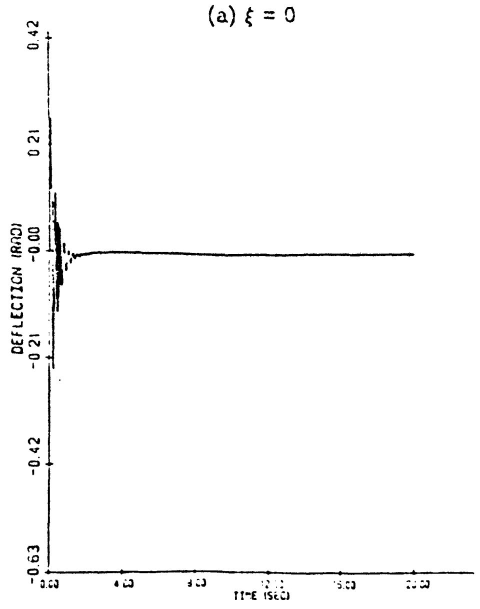
The control that is required for various values of controller delay is shown in Figure 7. Due to the absence of any random effects, elevator deflections for all the delays considered tend to the same value as the end of the landing phase (20 seconds) is approached, although much larger controls are needed initially. In the presence of random effects, the divergence between controls needed in the low and the high delay values of controller delay is even more marked. We present an example of this in Figure 8. The random effect considered here is the elevator being stuck at -35 ° for 60 milli-seconds 8 seconds into the landing phase due to a faulty controller order. The controlled process is assumed in Figure 8 to be in the subspace in which the landing job maps into a non-critical process (defined in the sequel as S_0). The diagrams speak for themselves. We shall show later that this demand on control is fully represented by the nature of the derived cost function. Also, above a certain threshold value for controller delay, we would expect the system to become unstable. This is indeed the case in the present problem, although this point occurs beyond a delay of 60 milli-seconds for all points in the allowed state space (obtained in the next section), which cannot by definition occur here.



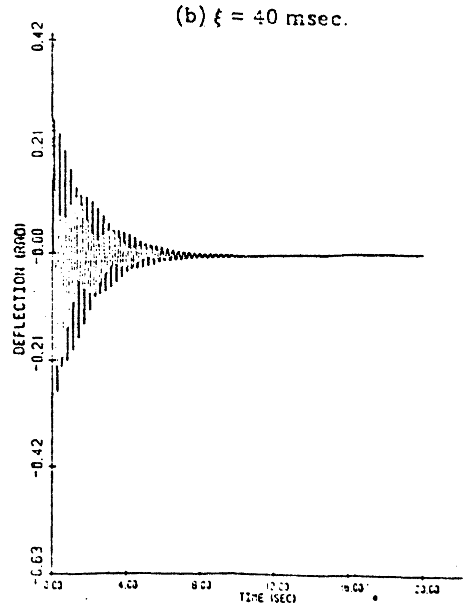
(a) $\xi = 0$



(b) ξ = 40 msec.

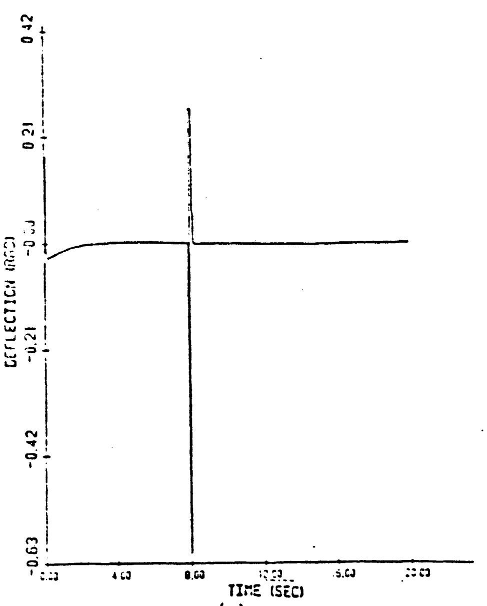


(c) ξ = 50 msec.

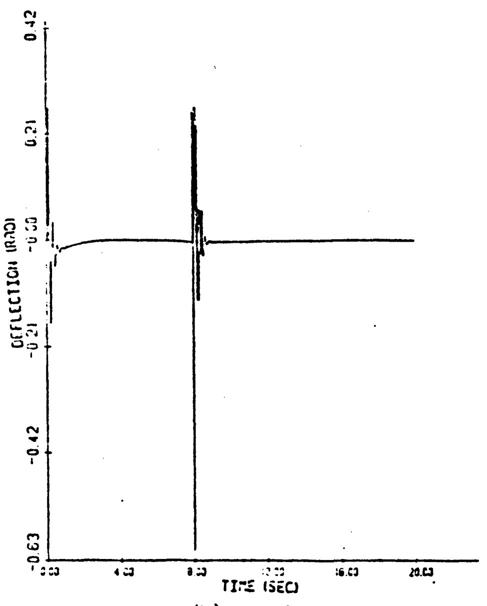


(d) ξ = 60 msec.

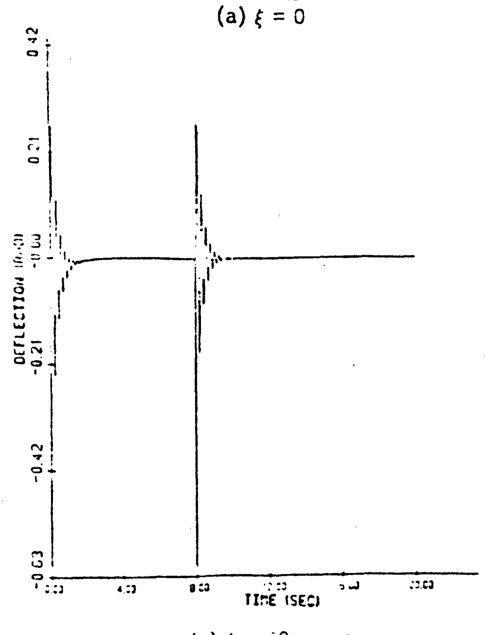
ctions without Abnormality



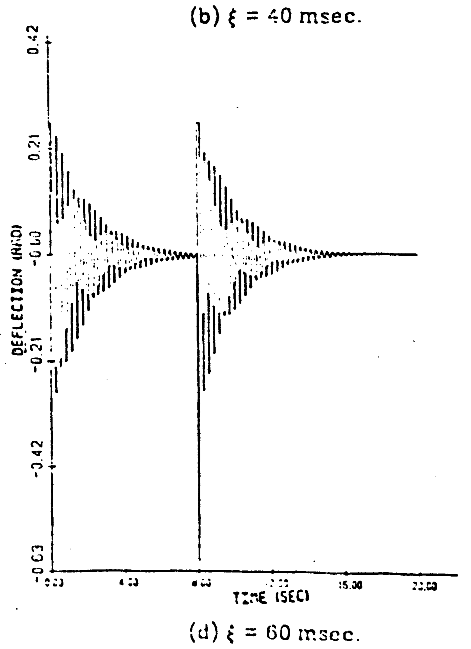
(a) $\xi = 0$



(b) ξ = 40 msec.



(c) ξ = 50 msec.



(d) ξ = 60 msec.

Figure 8. Elevator Deflections with Abnormality

C Allowed State Space

In this subsection, we derive the allowed state space of the aircraft system. To do so, note that in Ellert and Merriam's model, $\mathbf{X}_A^1$ does not exist. The reason is that the state equations do not take into account the angle of attack. In the idealized model we are considering, it is implicitly assumed that the constraint on the angle of attack is always honored, so that the only constraints to be considered are the terminal constraints.

The terminal constraints have been given earlier but are repeated here for convenience. The touchdown speed must be less than 2 feet/sec in the vertical direction, and the pitch angle at touchdown must lie between 0 ° and 10 °. To avoid overshooting the runway, touchdown must occur at between 4864 and 5120 feet in the horizontal direction from the moment the landing phase begins. The horizontal velocity is assumed to be kept constant throughout the landing phase at 256 feet/sec. Thus, touchdown should occur between 19 and 20 seconds after the descent phase begins. The only control is the elevator deflection which must be kept between -35 ° and 15 °.

The restriction that we place on the allowed state-space, $\mathbf{X}_A^s$, is for ease of representation. We define allowed "component state-spaces" for the individual components of the state-vector such that any point in the 4-dimensional state-space whose individual components each lie in the corresponding component state-spaces is in the overall allowed state-space. The component state-spaces are determined by perturbing the state values around the desired trajectory and determining the maximum pertubation possible under the requiement that no terminal constraint be violated. They were obtained by a search process and are plotted in Figure 9. It should again be stressed that what we plot in Figure 9 is a *subset* of $\mathbf{X}_A^a$.

D Designation of Subspaces

We subdivide the allowed state-space found above using the method described in Section 3. The criterion used is the hard deadline, since the finite cost function (derived in the next subsection) is found not to vary greatly within the whole of the allowed state-space. The value of Δ chosen is 60 milli-seconds. In other words, we wish to consider only the case where a trigger is "missed."

The allowed state-space in Figure 9 is subdivided into two subspaces, S_0 and S_1. These correspond to the deadline intervals $[120, \infty)$ and $[60, 120)$ respectively. S_0 is the non-critical region corresponding to the $[120, \infty)$ interval. Here, even if the controller exhibits any of the abnormalities considered earlier, the airplane will not crash. In other words, if the controller orders an incorrect output, exhibits an abnormal execution delay or simply provides no output at all before the following trigger, the process will still survive at the end of the current inter-trigger interval if, at the beginning of that interval, it was in S_0.

On the other hand, if the process is in S_1 at the beginning of an inter-trigger interval, it may safely endure a delay in controller response. However, if the controller behaves abnormally in either providing no output at all for the current trigger cycle or in ordering an incorrect output, there is a positive probability of an air crash.

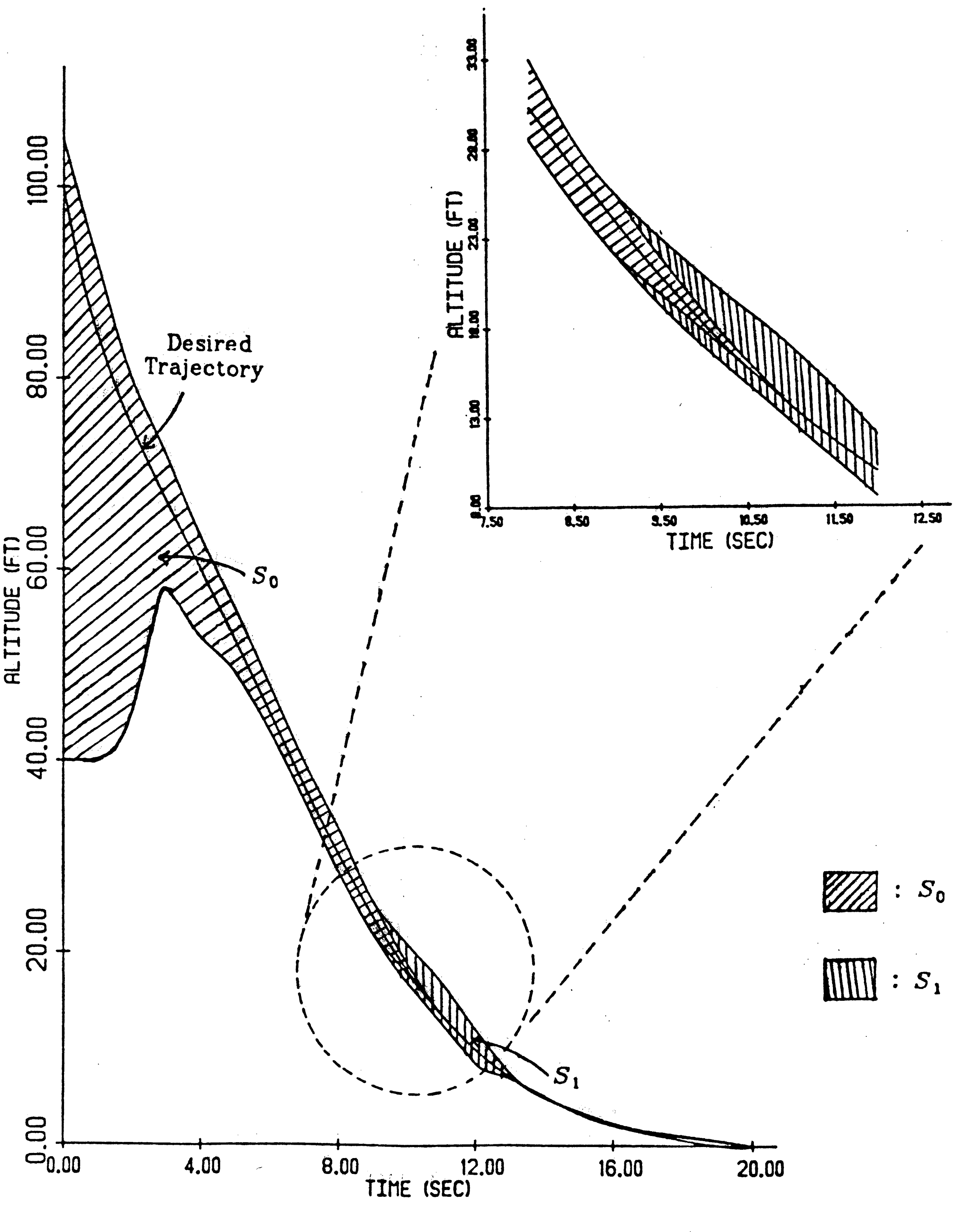


Figure 9(a). Allowed State Space: Altitude

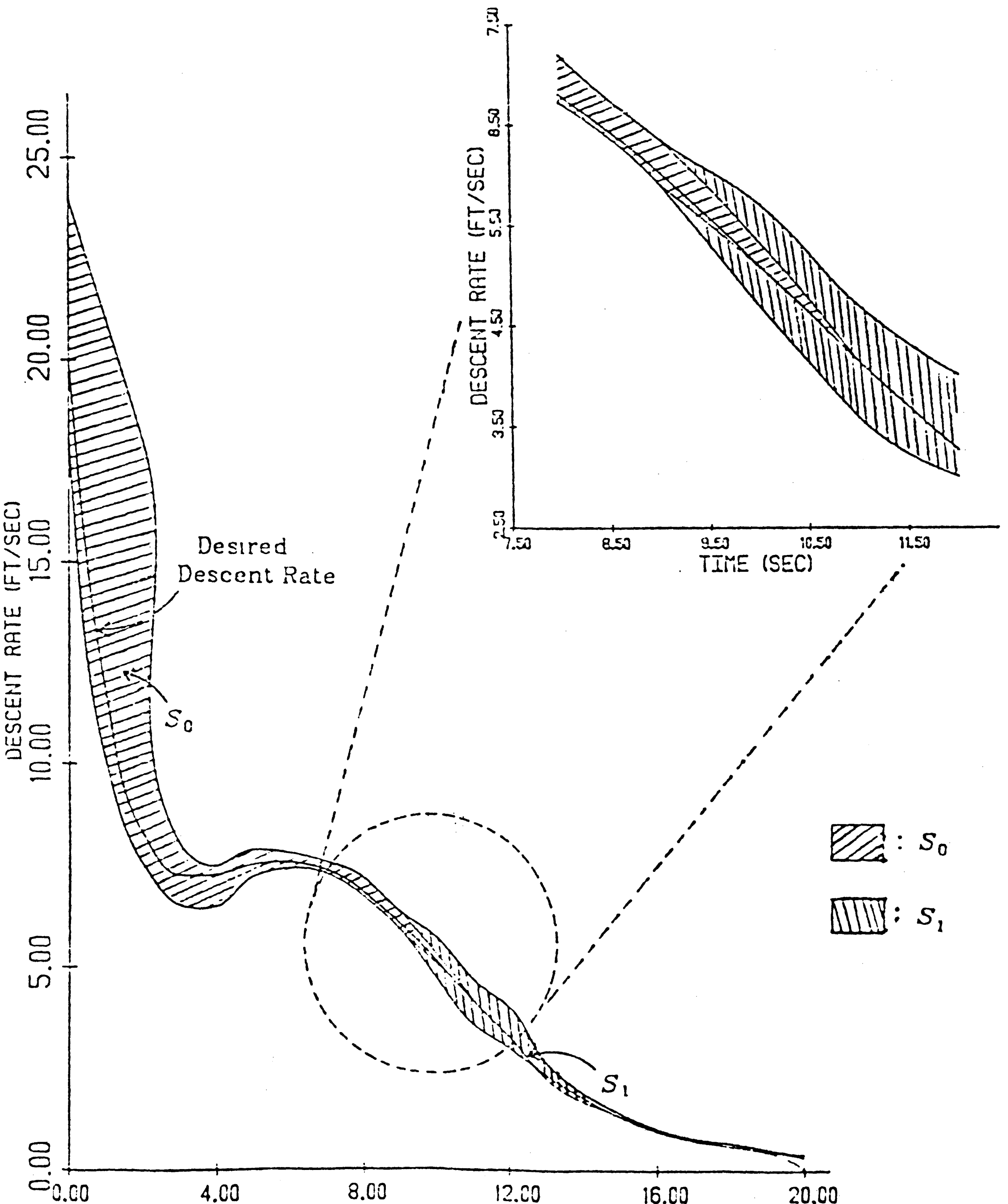


Figure 9(b). Allowed State Space: Descent Rate.

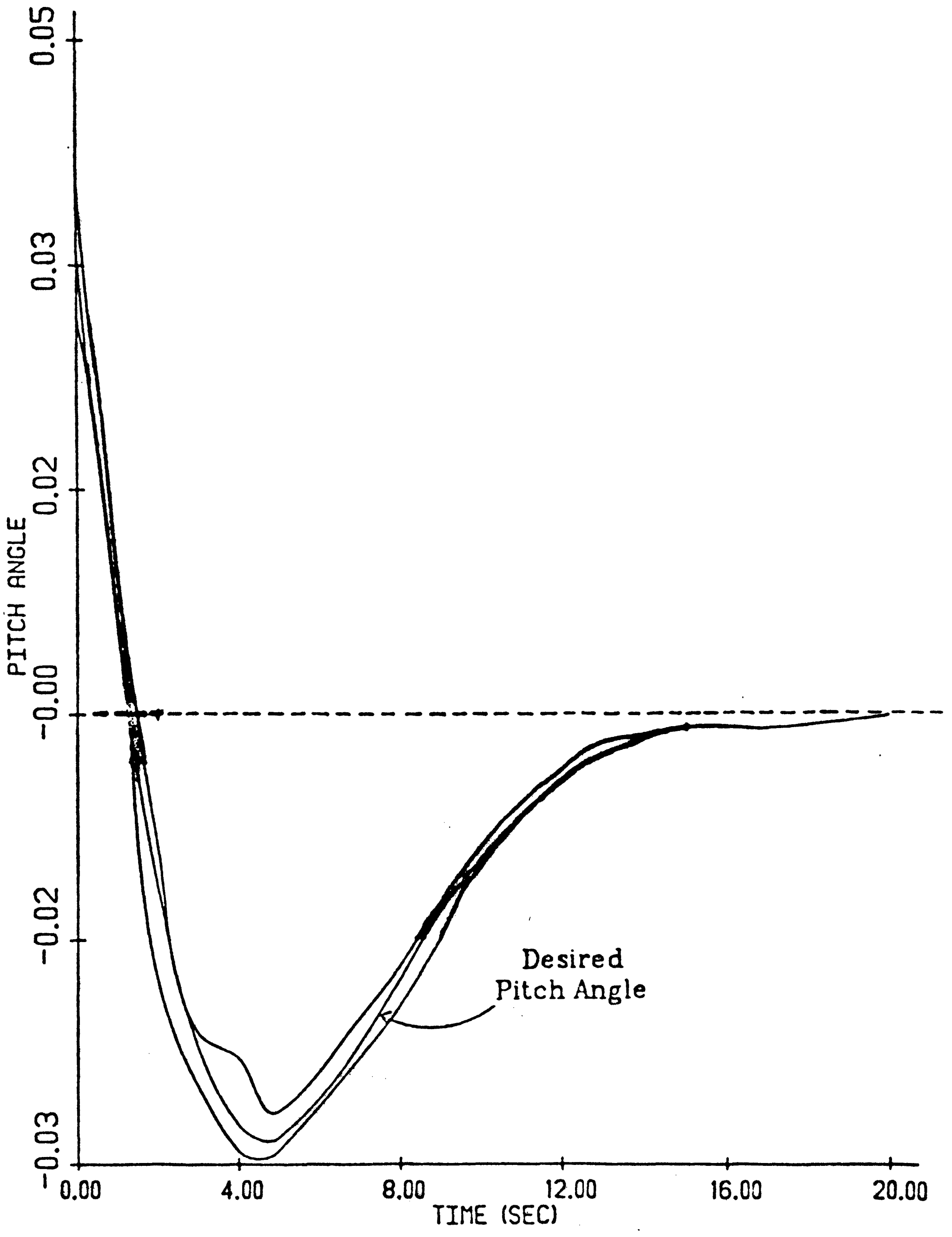


Figure 9(c). Allowed State Space: Pitch Angle

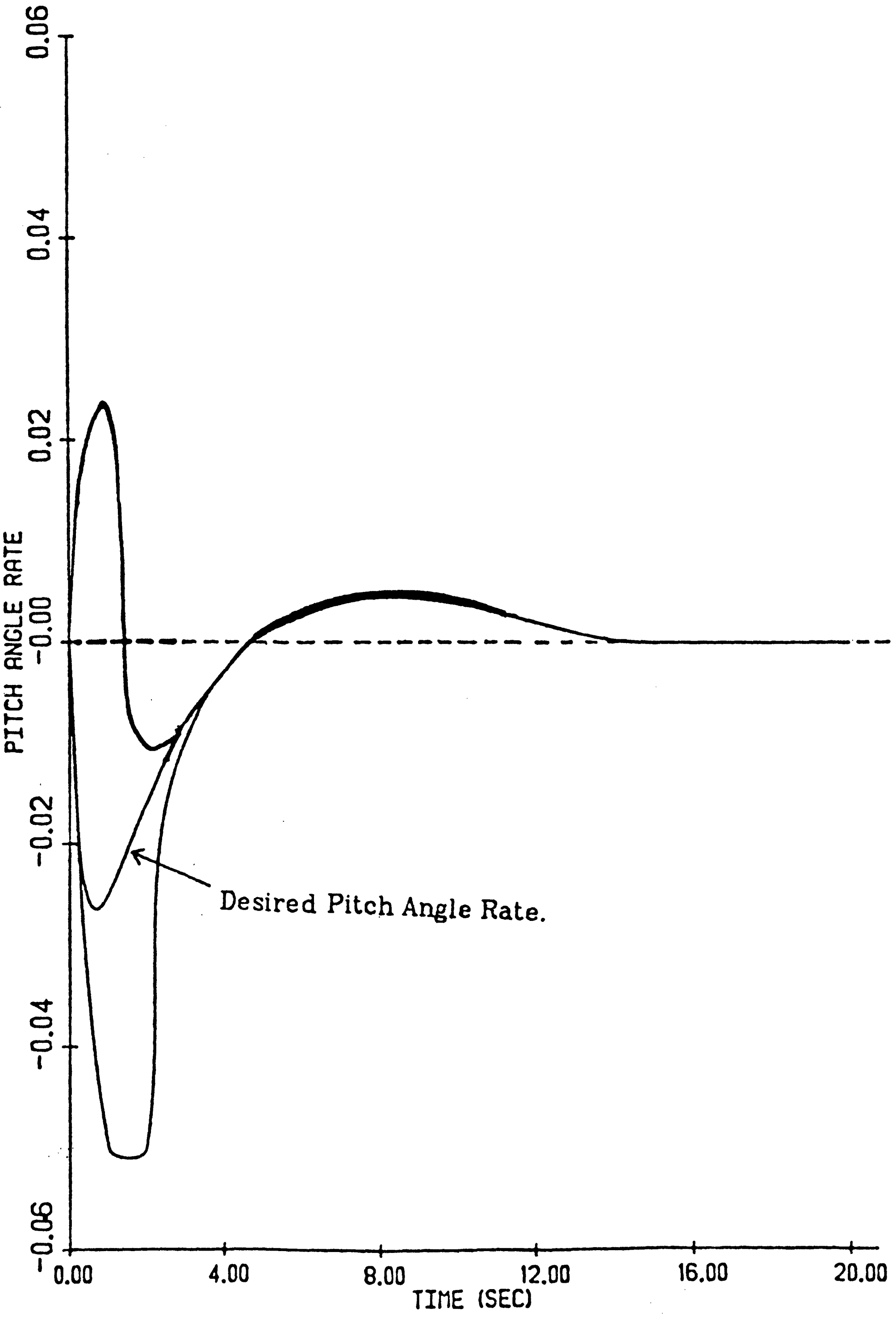


Figure 9(d). Allowed State Space: Pitch Angle Rate

Notice that we explicitly consider only missing a single trigger, not the case when two or more triggers might be missed in sequence. This is because dynamic failure is treated here as a function of the state at the moment of triggering. If two successive triggers are missed, for example, we have to consider two distinct states, namely the states the process is in at the moment of those respective triggers. To speak of deadline intervals beyond 120 milli-seconds is therefore meaningless in this case since the triggers occur once every 60 milli-seconds. This is why the second deadline interval considered is $[120, \infty)$, not $[120, 180)$.

The hard deadline may conservatively be assumed to be 60 milli-seconds in S_1. By definition it is infinity in S_0.

E Finite Cost Functions

The finite cost does not vary greatly within the entire allowed state-space. It is therefore sufficient to find a single cost function for S_0 or S_1.

The determination of the cost function is carried out as a direct application of its definition. That is, the process differential equations are solved with varying values of ξ. The value of ξ cannot be greater than the inter-trigger interval of 60 milli-seconds since, by assumption, no job pipelining is allowed and the controller terminates any execution in progress upon receiving a trigger. The finite cost function is found by computation to be approximately the same over the entire allowed state-space as defined in Figure 9.

In Figure 10, the finite cost function is plotted. The cost function is in the units of the performance index. Bear in mind that these measures are the result of an idealized model. We have, for example, ignored the effects of wind gusts and other random effects of the environment. When these are taken into account, the demands on controller speed get even greater, i.e. the costs increase.

The reader should compare the nature of the cost function with the plots showing elevator deflection in Figure 7, and notice the correlation between the marginal increase in cost with increased execution delay and the marginal increase in control needed, also as a function of the execution delay.

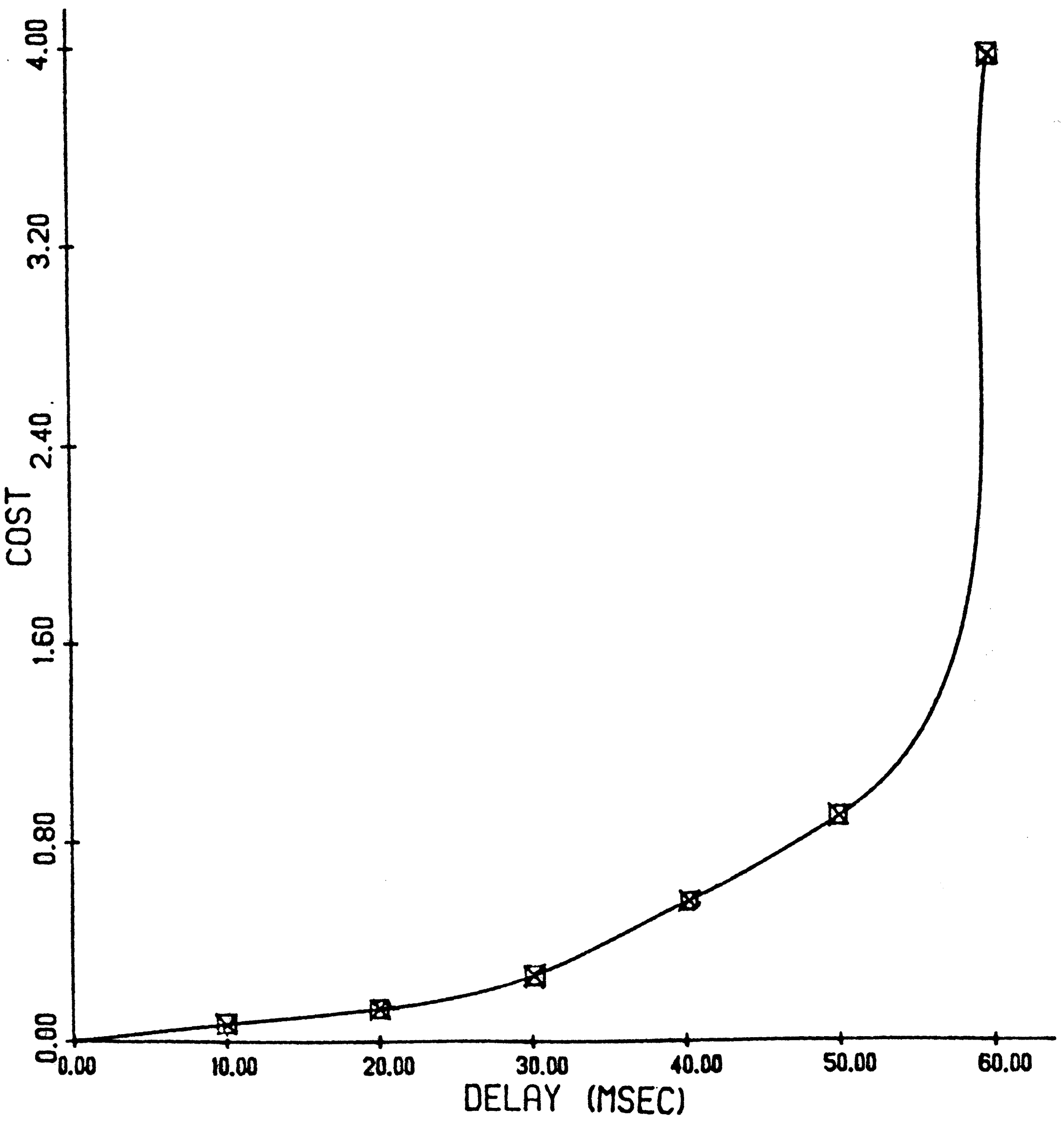


Figure 10. Landing Job Cost Function.

PART III

APPLICATIONS OF THE MEASURES

CHAPTER 5

THE NUMBER-POWER TRADEOFF

The number-power tradeoff problem can be stated as follows [17]. It appears intuitively obvious if there are no device failures, that a system with a *single* processor with exponentially distributed service time with mean $1/\mu$ is more efficient than an N-processor ($N>1$) system with each processor providing exponential service at rate μ/N. When failure is allowed for in the model, the above assertion is no longer obvious, and may not even be true in specific instances. So, we ask the question, "Given that the total processing power (number of processors $\times$ service rate per processor, called here the *number-power product*) is fixed at μ, what is the optimal number, N, of processors, that the system should start out with for a specific mission lifetime?" We extend an adaptation of this problem to demonstrate the use of our performance measures to real-time computers.

Two observations are in order here. Firstly, we tacitly assume that processors with *any* prescribed power are available. This is not true, although a wide variety of processors *is* available. Secondly, such a tradeoff depends for its resolution upon the cost functions and hard deadlines introduced above. It is this second point we pursue here. Specifically, we set out to determine for an example control computer, the configurations that meet specifications of reliability, and the sensitivity of reliability and the mean cost to changes in the number-power product under different operating conditions of hard deadline and mission lifetime. Implicit in all this will be the tradeoff between device redundancy and device speed.

A. System Description and Analysis

We use the multiprocessor system in Figure 11 to demonstrate the idea. Assume that there is a single job class that enters the system as a Poisson process with rate λ, and which requires an exponentially distributed amount of service with mean $1/\mu$. Then the system at any given time is an M/M/c queue (if the small dispatch time is ignored), where c is the number of processors functioning at that time. The distribution function for the response time for an M/M/c queue is well known [18]. It is given by:

$$F_{MMc}(t) = \frac{\{\lambda - c\mu + \mu W_c(0)\}(1 - e^{-\mu t}) + \mu\{1 - W_c(0)\}\{1 - e^{-(c\mu-\lambda)t}\}}{\lambda - (c-1)\mu} \tag{28}$$

where $W_c(0)$ is the probability that, when there are c processors functional, at least one functional processor is free. This is given by:

$$W_c(0) = 1 - \frac{c(\lambda/\mu)^c}{c!(c-\lambda/\mu)}\left[\sum_{n=0}^{c-1}\frac{1}{n!}\left(\frac{\lambda}{\mu}\right)^n + \frac{1}{c!}\left(\frac{\lambda}{\mu}\right)^c\left(\frac{c\mu}{c\mu-\lambda}\right)\right]^{-1} \tag{29}$$

This distribution function is not defined unless $c\mu > \lambda$.

Assume that the hard deadline has a probability distribution function F_d, and that the finite cost function for the task is denoted by g (as in Eq. (18)). Let the processors fail according to an exponential law with rate μ_p. Also, let n be the smallest integer for which $n\mu > \lambda$ -- clearly, if there are fewer than n processors function-

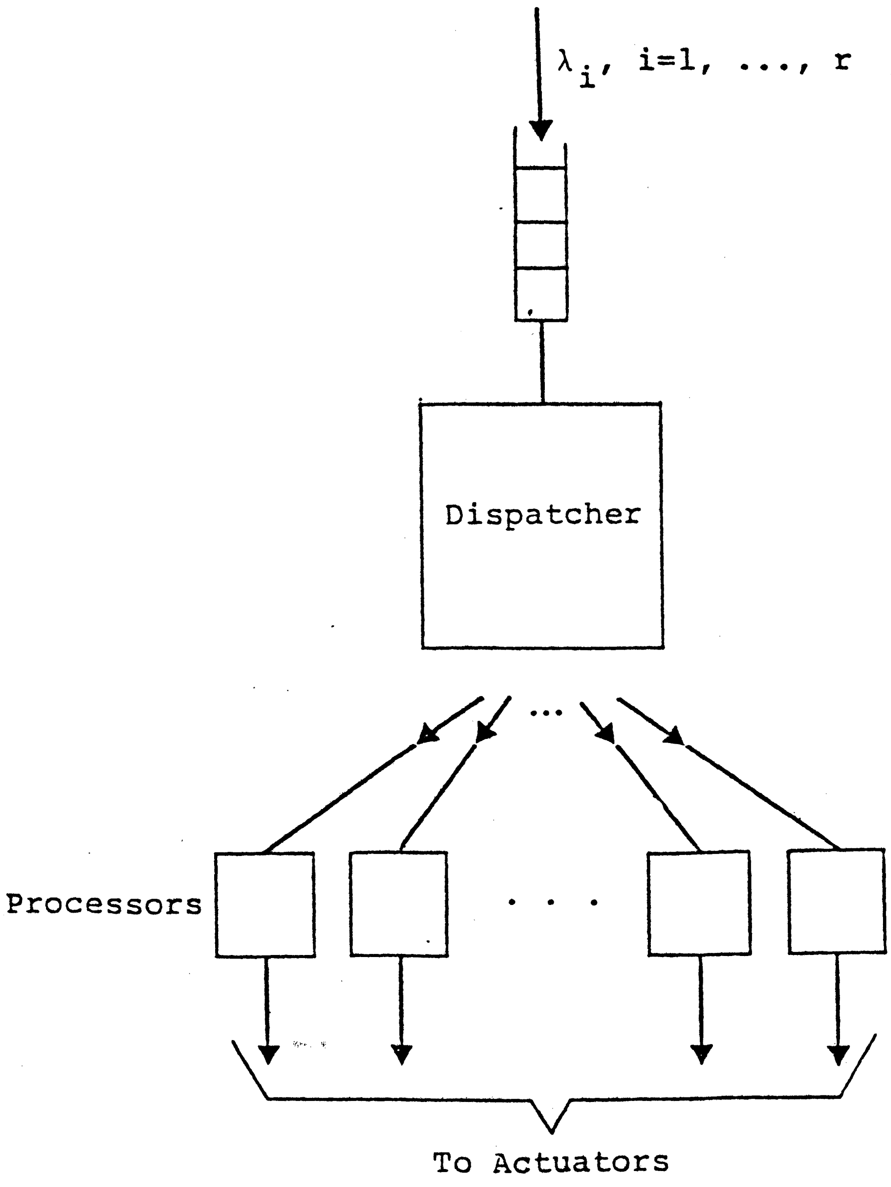


Figure 11. A Real-Time Multiprocessor.

ing, the utilization exceeds unity, and failure takes place with certainty.

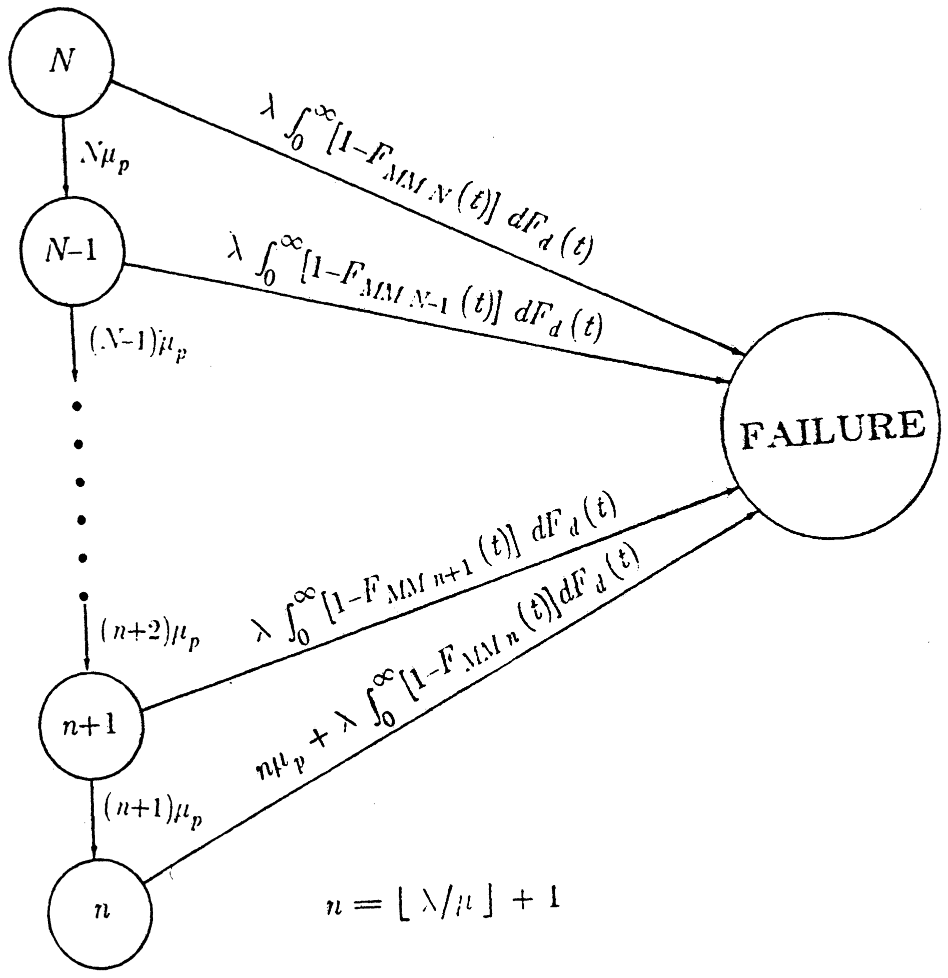
The system fails if a hard deadline is violated, or if there are fewer than n processors functioning. (We do not consider here the failure of the interconnection net, or of the dispatcher. Taking account of these is easy, but would obscure the analysis somewhat.) When the system is in a state $i > n$, the rate at which failure can happen is equal to the product of the task input rate and the probability that the response time of the system exceeds the hard deadline. If we assume that steady state is achieved between each state transition (i.e. between processor failures), then the probability distribution function of the response time at state i is always given by $F_{MM\,i}$. Such an assumption is valid, since the Mean Time Between Failures for components used in such systems typically ranges from 1,000 to 10,000 hours. The probability of dynamic failure can therefore be computed using the Markov model in Figure 12. Denoting the probability of being in state i by π_i, the quantity $\lambda[1 - F_{MM\,j}(t)]$ by $\alpha(j,t)$, the failure state by *fail*, and the number of processors at start-up by N, the following balance equations can be written

$$\dot{\pi}_N(t) = -[N\mu_p + \int_0^\infty \alpha(N,\xi)\, dF_d(\xi)\,]\pi_N(t), \qquad \pi_N(0)=1 \tag{30a}$$

$$\dot{\pi}_i(t) = -[\,i\mu_p+\int_0^\infty \alpha(i,\xi)\, dF_d(\xi)\,]\pi_i(t) + (i+1)\mu_p\pi_{i+1}(t), \;\; \pi_i(0)=0, \;\; \text{for } n \leq i < d \tag{30b}$$

$$\dot{\pi}_{fail}(t) = \sum_{i=n}^{N} \{\int_0^\infty \alpha(i,\xi)\, dF_d(\xi)\}\pi_i(t) \; + \; n\mu_p\pi_n(t), \qquad \pi_{fail}(0)=0. \tag{30c}$$

Clearly, $p_{dyn}(t) = \pi_{fail}(t)$ so that a solution of the above equations yields the probability of dynamic failure. Implicit in these calculations is the assumption that the hard deadline is very much smaller than the mission lifetime or the sojourn time in the various states. This is invariably the case in practice.



N
$N\mu_p$
$N-1$
$(N-1)\mu_p$
$(n+2)\mu_p$
$n+1$
$(n+1)\mu_p$
n
FAILURE
$\lambda \int_0^\infty [1-F_{MM\,N}(t)]\, dF_d(t)$
$\lambda \int_0^\infty [1-F_{MM\,N-1}(t)]\, dF_d(t)$
$\lambda \int_0^\infty [1-F_{MM\,n+1}(t)]\, dF_d(t)$
$n\mu_p + \lambda \int_0^\infty [1-F_{MM\,n}(t)] dF_d(t)$
$n = \lfloor \lambda/\mu \rfloor + 1$

Figure 12. Markov Model for Number-Power Tradeoff

To compute the mean finite cost over the mission lifetime, we first have to evaluate the distribution function of the response time, conditioned on the event that no hard deadline is violated. This distribution function for a c-processor system, denoted by $F_{MM\,c}^{finite}$, is given by:

$$F_{MM\,c}^{finite}(\xi) = \int_0^\infty \frac{F_{MM\,c}(\xi)}{F_{MM\,c}(\tau)}\, dF_d(\tau) \tag{31}$$

where the function is only defined for arguments less than the associated hard deadline. Then, the mean cost is defined by:

$$M = \sum_{c=n}^{N} \int_0^\infty \int_0^\infty \int_0^\xi \lambda\; \pi_c(t)\; g(\tau)\; dF_{MM\,c}^{finite}(\tau)\; dF_d(\xi)\; dL(t) \tag{32}$$

The above expressions are used in the following section to obtain values for the probability of dynamic failure, and the mean cost as a function of the mission lifetime.

B Numerical Results and Discussion

In what follows, it is assumed that the job arrival rate is $\lambda=100$, and that the processor failure rate is $\mu_p=10^{-4}$. All time units are in hours.

Figure 13 is the probability of dynamic failure when the task is non-critical, i.e. the deadline is at infinity. In such cases, the probability of dynamic failure reduces to being the probability of *static failure,* namely the probability of failure of hardware components to the point when the system utilization exceeds 100 %. This is because catastrophic failure does not occur in this case until the system utilization is greater than unity. This explains the monotonic nature of the plot.

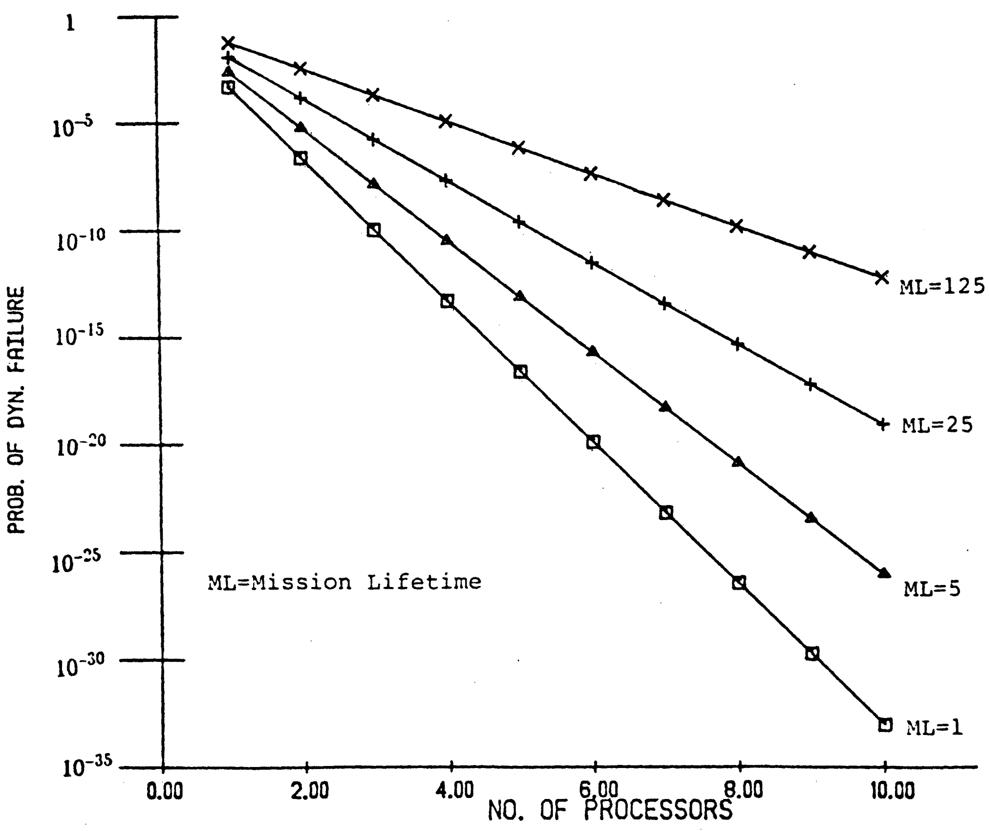


Figure 13. Dependence of Probability of Dynamic Failure on Processor Number when Tasks are Non-Critical.

Figure 14 shows the probability of dynamic failure when the task is critical with the deadlines for the respective curves noted in the figure. The number-power product is 2200. The curves that result for the failure plot form an inverted bell. The portions of the failure curve where the slope is positive can be explained as follows. When the number of processors increases, the response time distribution is skewed to the right. This in turn increases the probability of failing to meet the hard deadline to a greater extent than the static failure probability is reduced by the addition of further redundancy. The positive slope is the result of this tendency. When the hard deadline is smaller, the premium on speed is increased, and as a result, the trough of the curves moves to the left.

In the region corresponding to the fewest processors, there tends to be a trade-off between dynamic failure probability and the number of processors, leading to a negative slope for the failure curves. When there are few processors, the fault-tolerance is less and the probability of static failure is therefore greater. Since the total processing power of the processor bank is fixed, the few processors each have greater power, and the mean waiting time is low. This accounts for the very small nature of the non-static component of the failure probability. As the speeds of the individual processors are decreased, but their numbers increased commensurately, the static failure probability drops, but the probability of missing the hard deadline increases. In the area of the curve where the slope of the probability of failure is negative, the benefits accrued from adding redundancy outweigh the negative impact of the lowered individual speeds that result; elsewhere the reverse is the case. Notice that the curve corresponding to a deadline of 0.01 has no region of negative slope. This means that 2200 is a number-power product that is too small with respect to that hard deadline.

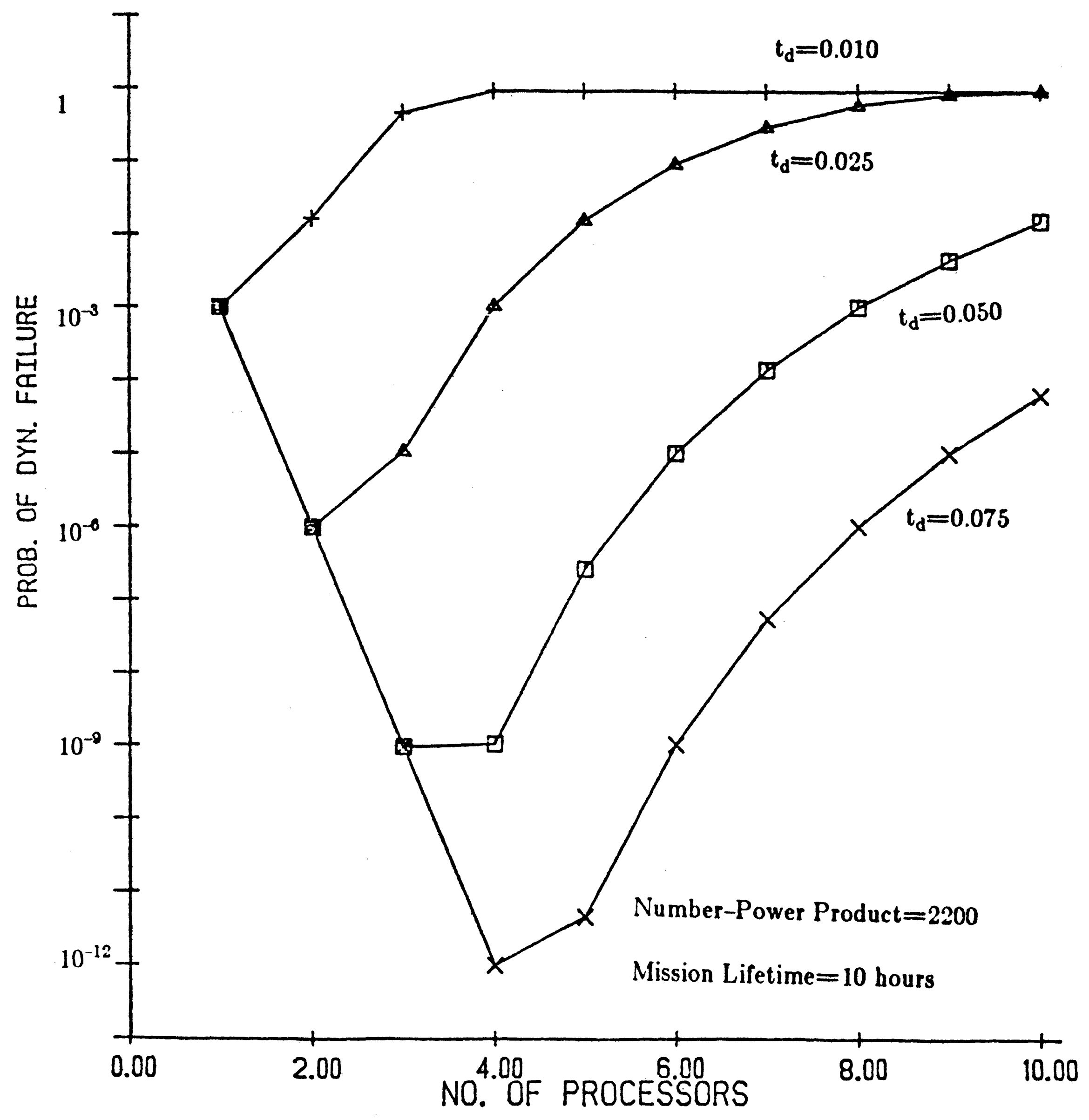


Figure 14. Dependence of Probability of Dynamic Failure on Processor Number and Hard Deadlines.

In Figure 15, the dependence of the probability of dynamic failure on the number-power product for a mission lifetime of 10 hours is considered. Each point on the curves represents the configuration yielding the lowest possible failure probability for the number-power product represented. The label of each point on this plot is the number of processors in the configuration for which this lowest failure probability is achieved. As the product increases, the optimal configuration tends to contain more processors: this also is due to the lowering of the non-static component of the dynamic failure probability when the product is increased.

Naturally, the curves are monotonically non-increasing. They serve to show the marginal gain in maximum achievable reliability that is to be had on increasing the number-power product at each point for the class of systems under consideration. Notice the "elbows" in the plot. These occur when the minimum failure probability configuration changes, and are the result of a tradeoff between the static and non-static components of the failure probability. The p_{dyn} drops exponentially with an increase in the product as long as the static component is a small fraction of p_{dyn}. When the non-static component drops to sufficiently below the static component value, the optimal configuration changes, and the static component once again becomes negligible compared to the non-static component. This race continues indefinitely and is portrayed in Figure 16. The discrete nature of the processors causes the elbows: if the number of processors were a continuous quantity, they would not appear.

The probability of dynamic failure is used as a pass-fail test for control computers. Plots such as Figure 15 can be used in this connection. As an example, let the mission lifetime be 10 hours, and the specified probability of dynamic failure

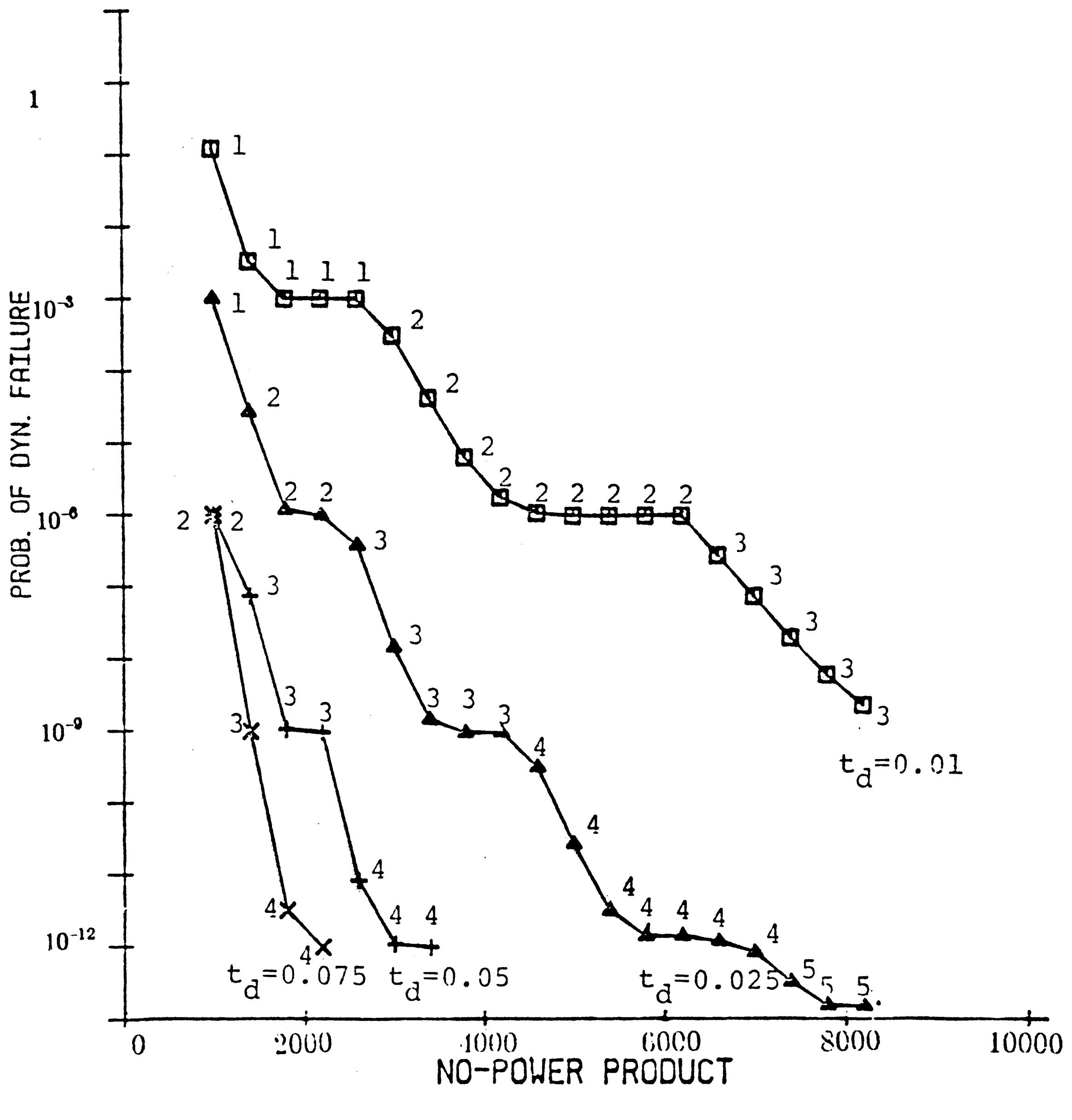


Figure 15. Minimum Achievable Probability of Dynamic Failure.

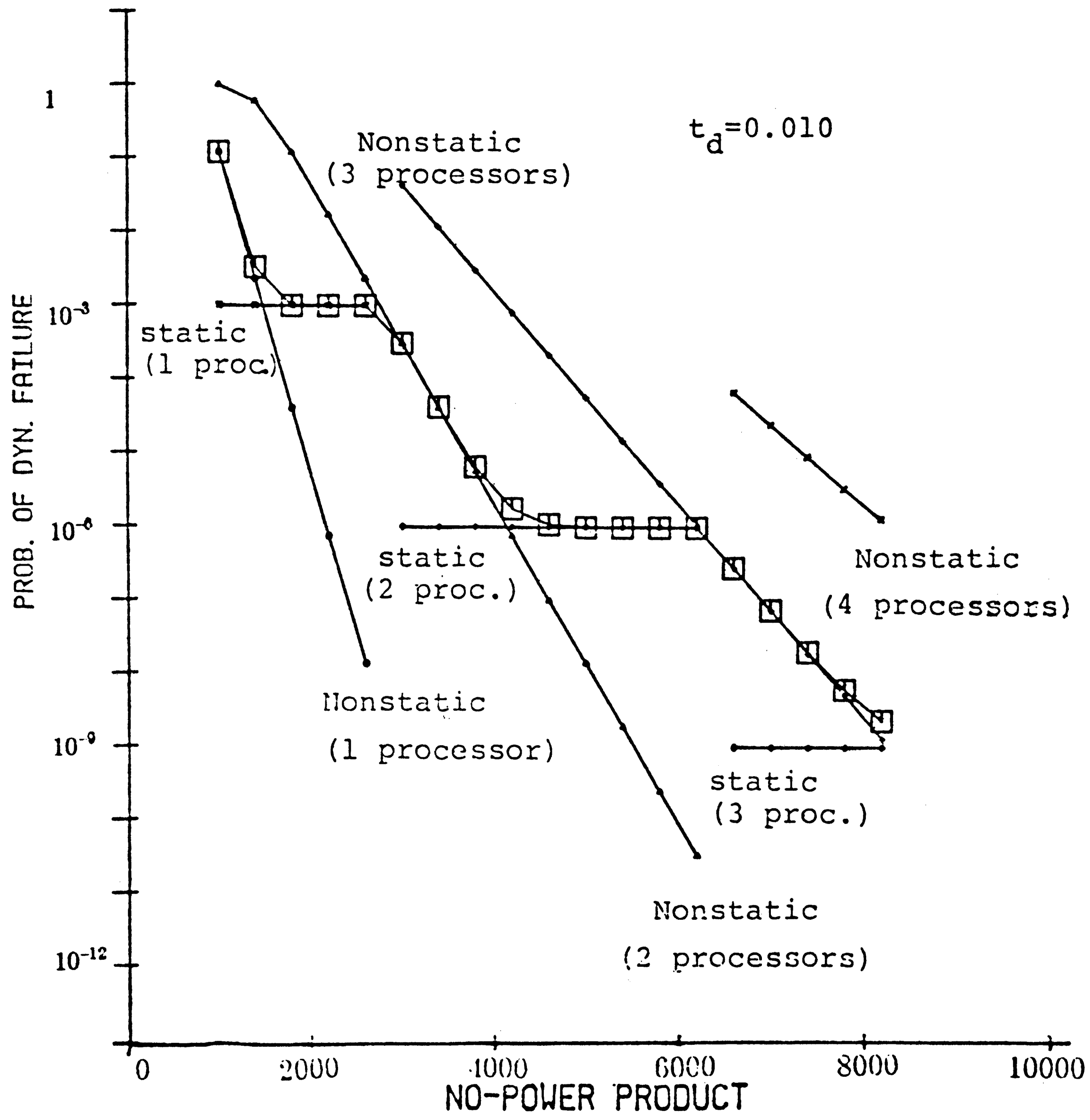


Figure 16. Race Between Static and Non-Static Components of Probability of Dynamic Failure.

equal to 10^{-7} over that period. Let the system parameters be those of the model in this section. Then, corresponding to each of the four deadlines considered, we can obtain graphically from Figure 15, the minimum number-power product that is required to satisfy the p_{dyn} specifications. These products are listed in Table 5. Any system that has a smaller number-power product must be rejected, no matter what its other credentials may be.

When this stage of the evaluation is complete, one has a set of acceptable configurations. Only after this point does the mean cost come into consideration. The mean costs associated with each of the points in Figure 15 is graphed in Figure 17, where the finite cost function, g, has been taken as equal to the response time for demonstrative purpose. The curves take the form of a sawtooth wave, with each upward transition occurring when the optimal configuration increases by one. Clearly, the greater the power of each processor, the smaller is the mean cost.

In Figure 18 (A, B, and C), we show the effects on p_{dyn} of changing mission lifetime for various values of the hard deadline, t_d. In the light of the preceding discussion, these plots should be largely self-explanatory. It is worth pointing out, however, that as the lifetime increases, the optimal configuration contains a larger number of processors. The trough (around the optimal point) becomes shallower as one increases the mission lifetime, until finally, it disappears to be replaced by a shallow trough one unit to the right. As the lifetime increases still further, the new trough deepens, then begins to become shallow. Whether or not this cycle continues depends upon the hard deadline: it will continue so long as the number-power product is sufficiently large to cope with the hard deadline at the lifetimes used; the plot will rise monotonically to a failure probability of one if this is not the case (cf.

Hard Deadline	Number-Power Product
0.010	7165
0.025	2126
0.050	1496
0.075	1180

Required $p_{dyn} = 10^{-7}$
Mission Lifetime = 10 hours

Table 5. Minimum Number-Power Product for Various Hard Deadlines.

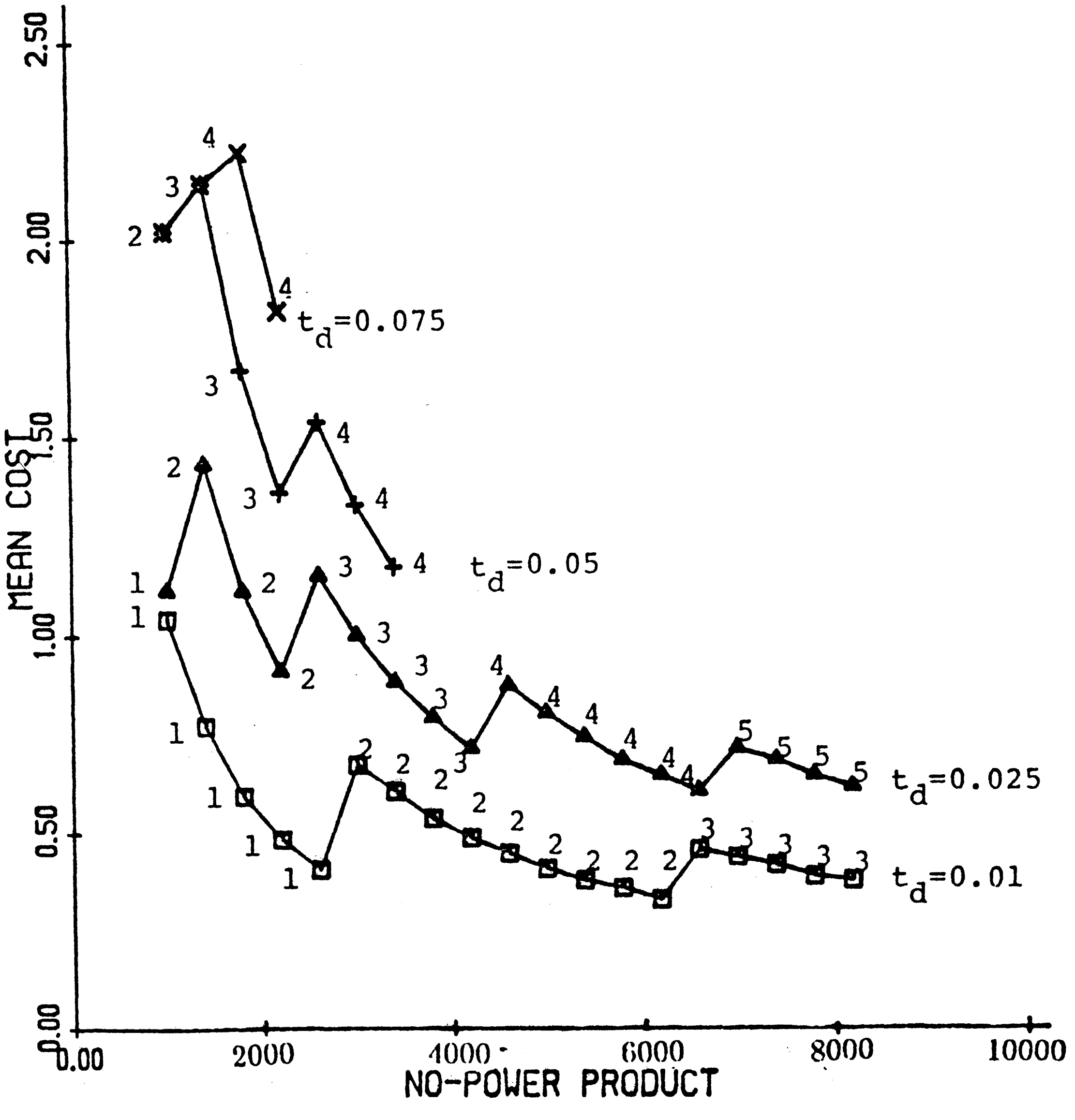


Figure 17. Mean Cost for Configurations of Figure 15.

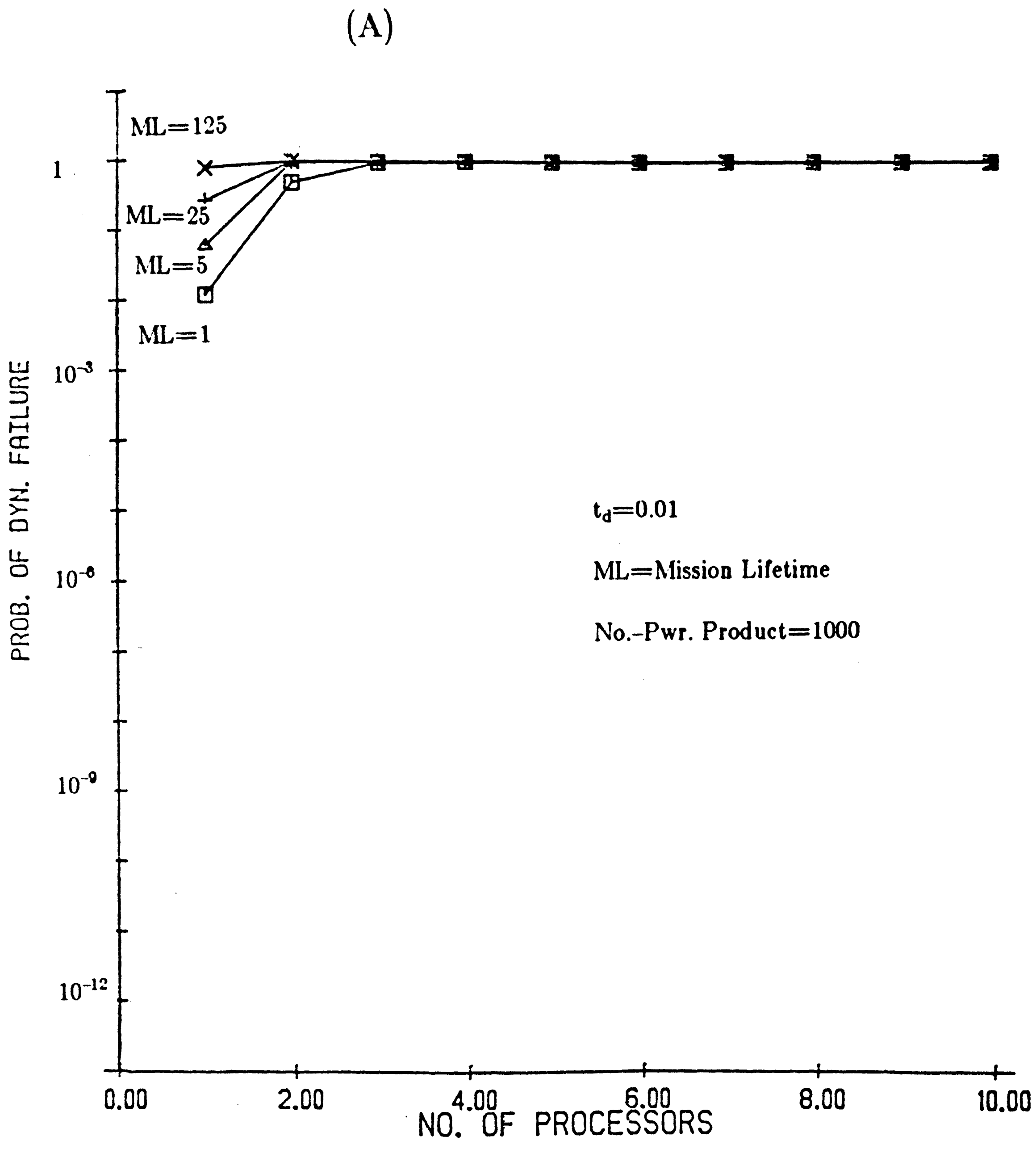


Figure 18(a). Probability of Dynamic Failure for a Constant Number-Power Product.

(B)

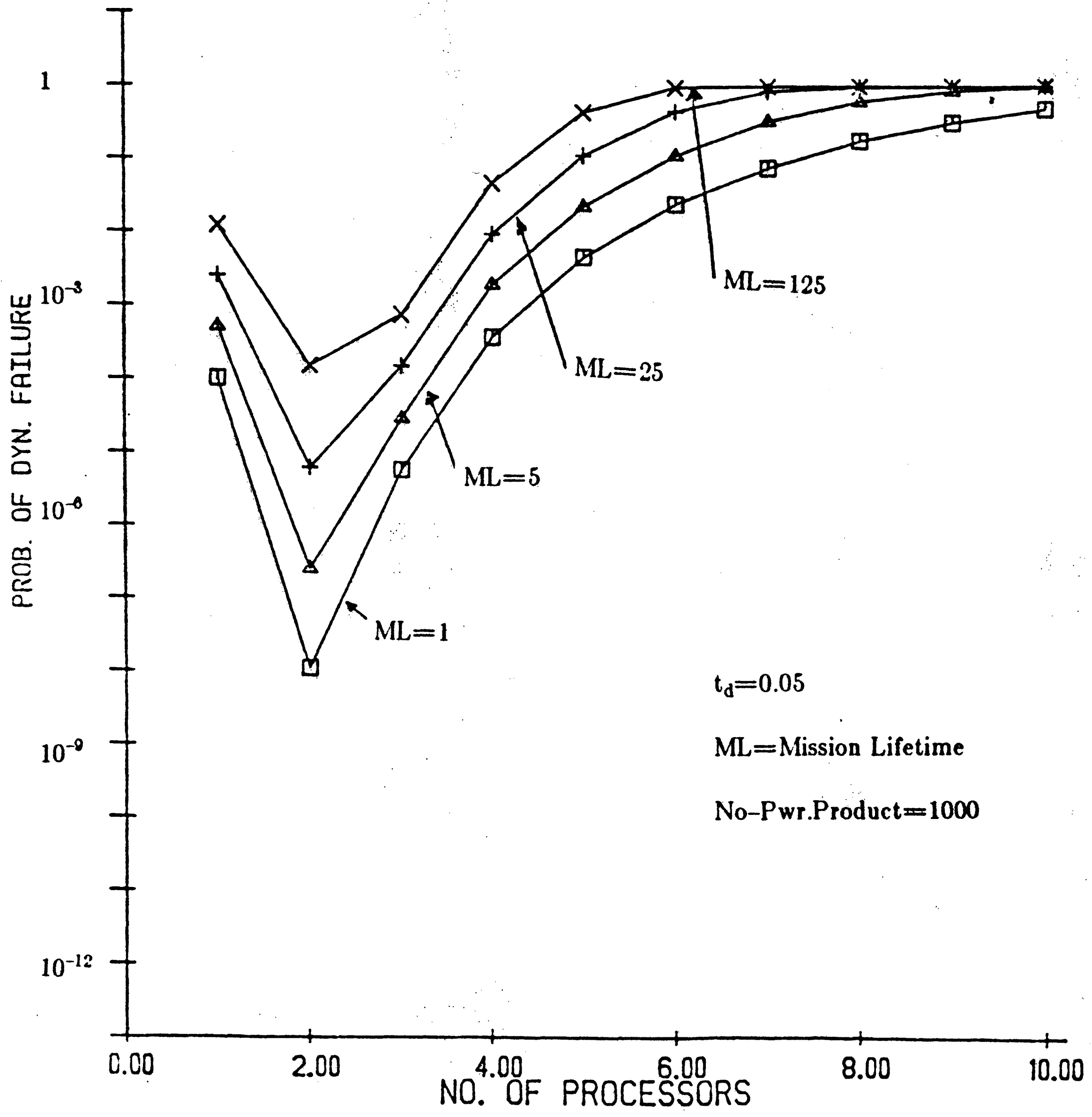
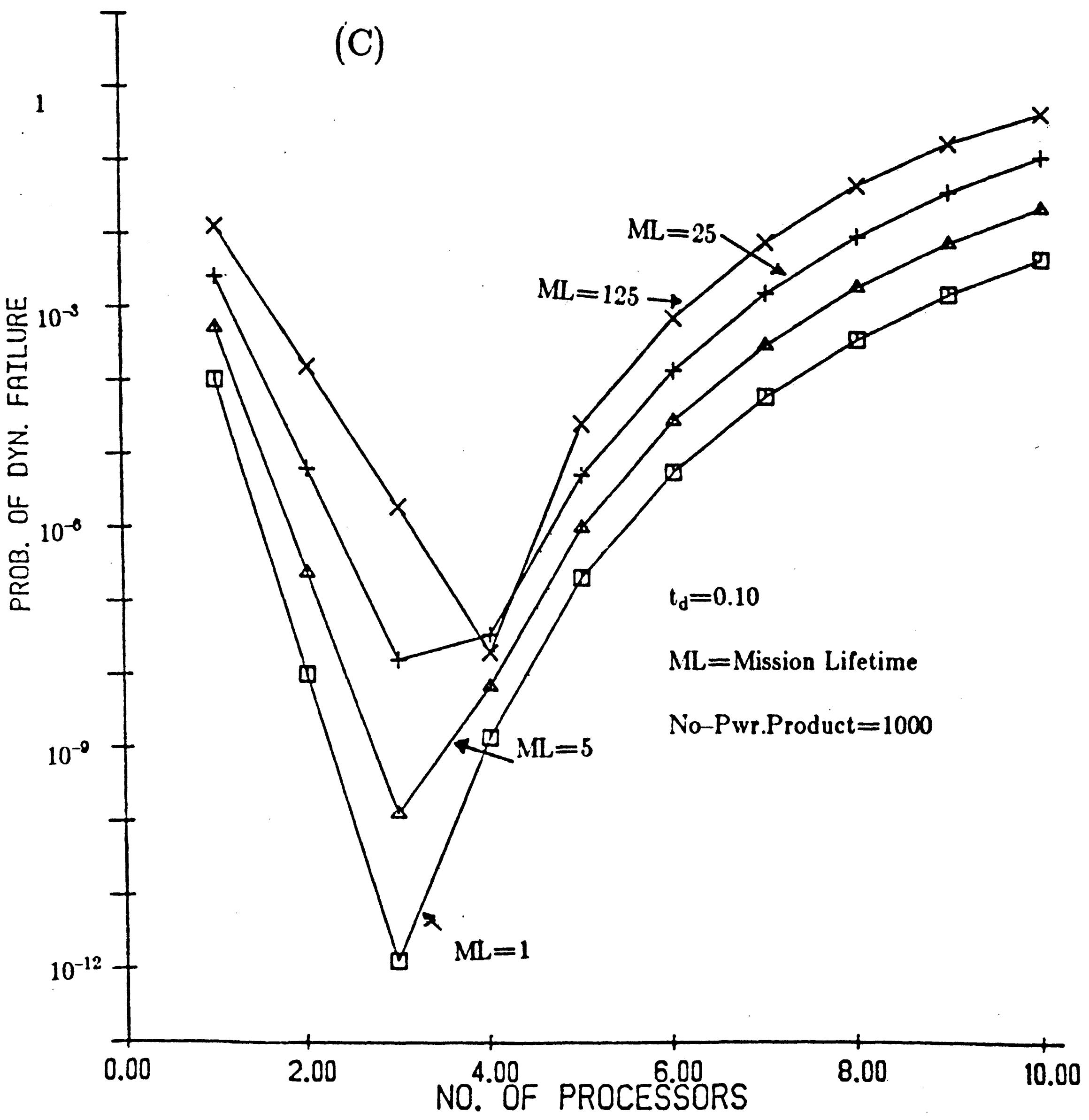

1
10^{-3}
10^{-6}
10^{-9}
10^{-12}
PROB. OF DYN. FAILURE
0.00
2.00
4.00
6.00
8.00
10.00
NO. OF PROCESSORS
ML=125
ML=25
ML=5
ML=1
t_d=0.05
ML=Mission Lifetime
No-Pwr.Product=1000

Figure 18(b).



(C)
1
10^{-3}
10^{-6}
10^{-9}
10^{-12}
PROB. OF DYN. FAILURE
0.00
2.00
4.00
6.00
8.00
10.00
NO. OF PROCESSORS
ML=25
ML=125
ML=5
ML=1
t_d=0.10
ML=Mission Lifetime
No-Pwr.Product=1000

Figure 18(c).

Figure 14).

In Figure 19 (A, B, and C), we show the associated Mean Costs per unit time of operation using the same cost function as was used in Figure 17. In all three curves, we may note the anomaly mentioned in Section 2: as the lifetime increases, and as the number of processors increases, there is a region over which the mean costs per hour actually drop. It is most pronounced in Figure 19A, where the probability of dynamic failure is close to unity under almost all configurations. This anomaly, of course, is due to the fact that the mean costs are computed on a response-time distribution that is conditioned on the system's not failing. Thus, on comparing Figures 19A, 19B, and 19C, we see that the system operating in the longest hard deadline, and therefore having greater reliability exhibits a ***higher*** mean cost per hour in some configurations than its identical counterparts that operate under more difficult conditions. If this causes undue irritation, the anomaly can be made to vanish by redefining the finite cost function for a task i to be:

$$f_i(t) = \begin{cases} g_i(t) & \text{if } t \le t_d \\ g_i(t_d) & \text{if } t > t_d \end{cases} \tag{33}$$

introducing the following functions:

$$\alpha_i(t) = \sum_{j=1}^{q_i(t)} g_i(\Xi(V_{ij}, r_{ij}, n_{ij}, t)) \tag{34}$$

$$\beta(t) = \sum_{j=1}^{r} \Gamma_i(t) \tag{35}$$

and defining the mean and variance costs to be:

$$\textit{Mean Approximate Cost}\ (MAC) \equiv \int_0^{\infty} E\{\beta(t)\}\, dL(t) \tag{36}$$

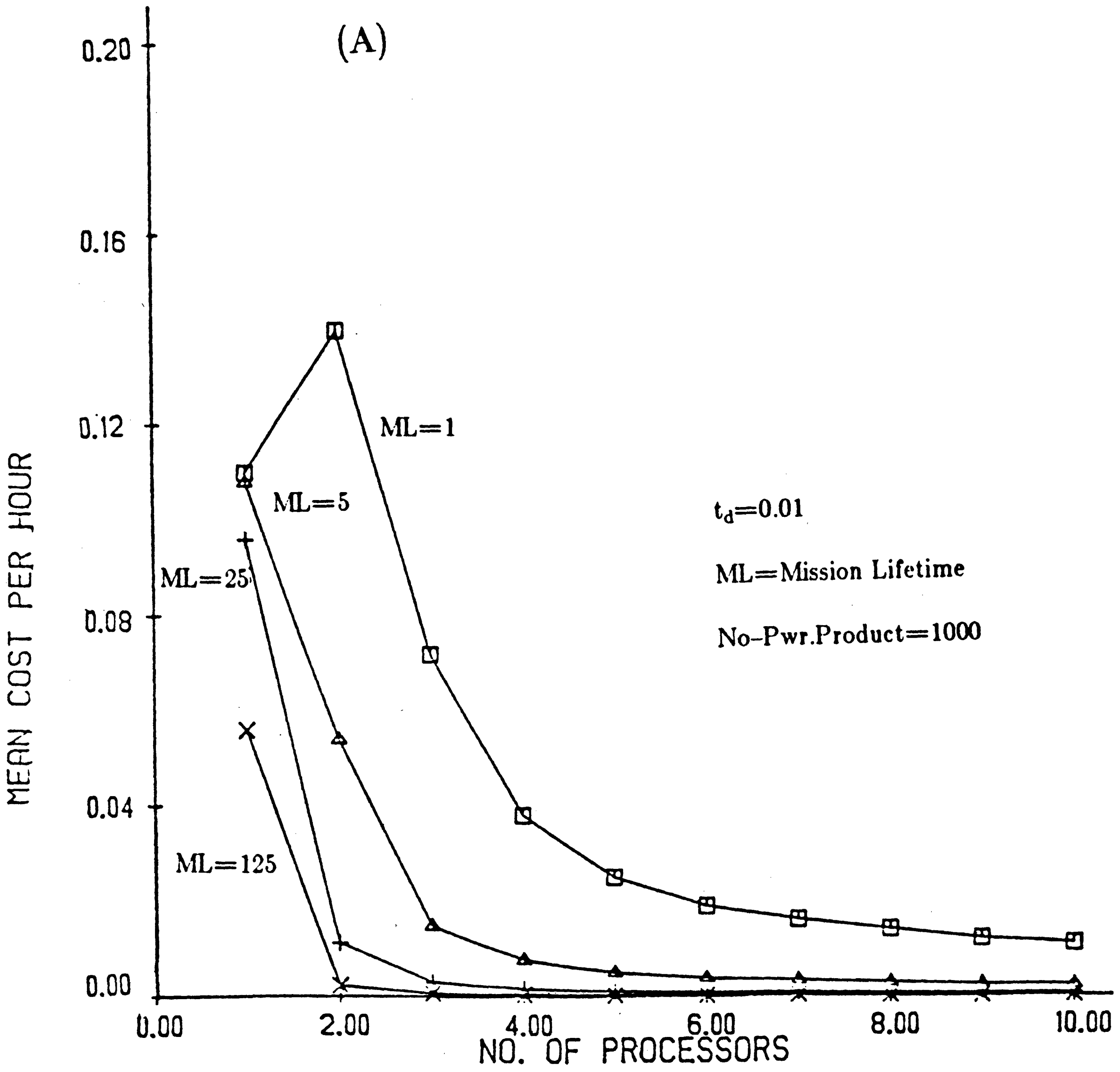
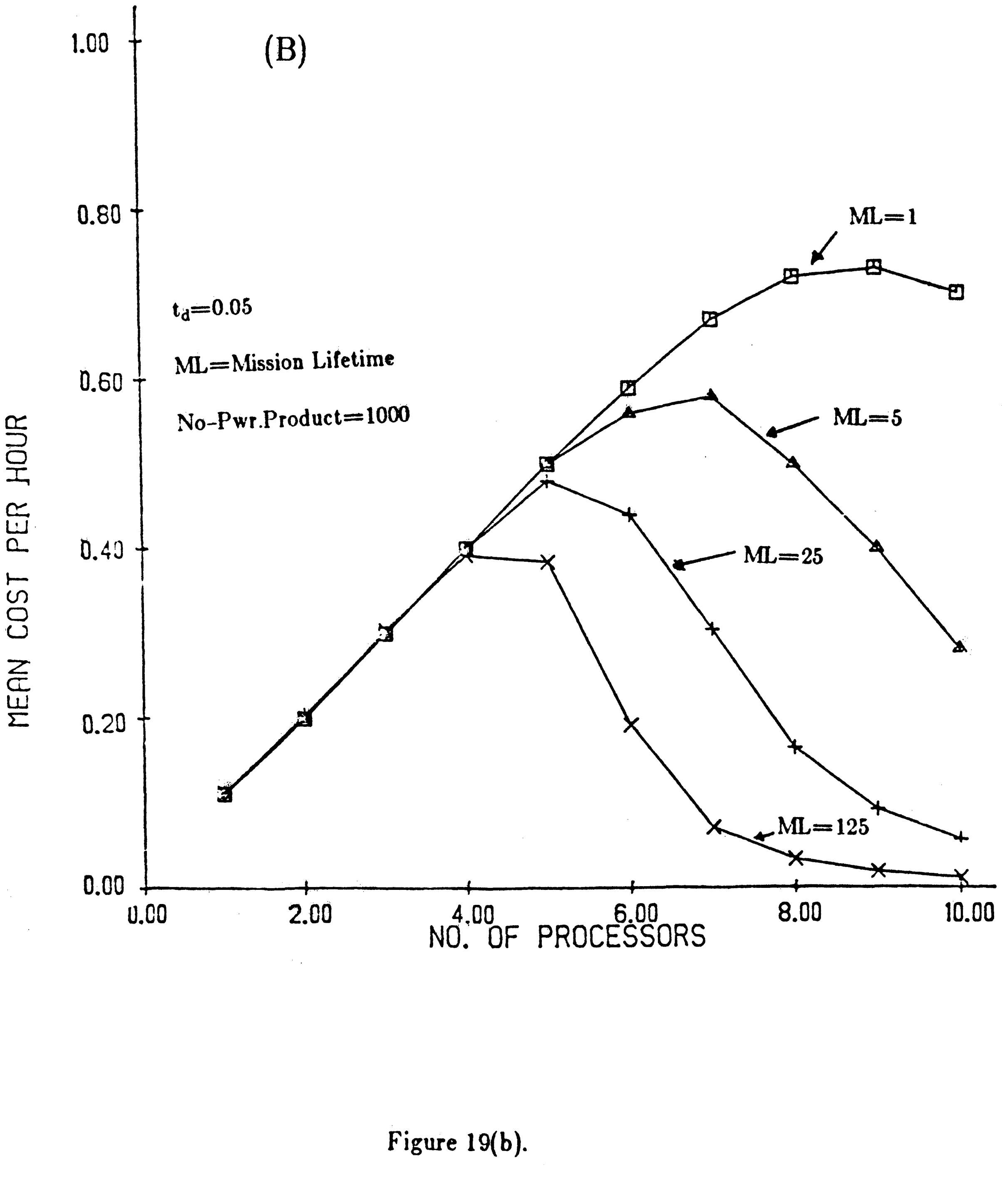


Figure 19(a). Mean Costs for a Constant Number-Power Product.



(B)
t_d=0.05
ML=Mission Lifetime
No-Pwr.Product=1000
ML=1
ML=5
ML=25
ML=125
MEAN COST PER HOUR
NO. OF PROCESSORS
1.00
0.80
0.60
0.40
0.20
0.00
0.00
2.00
4.00
6.00
8.00
10.00

Figure 19(b).

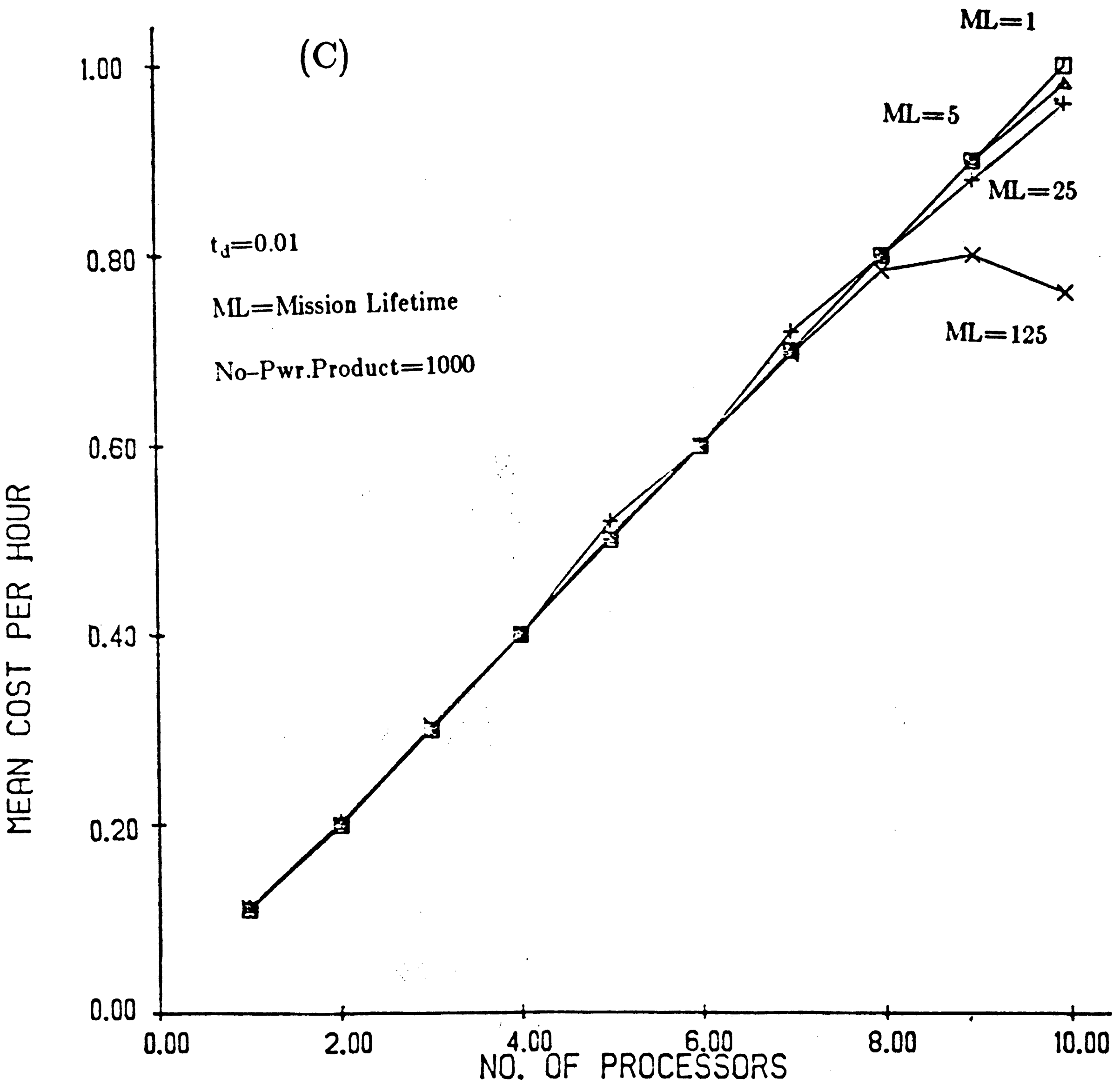

(C)
ML=1
ML=5
ML=25
ML=125
t_d=0.01
ML=Mission Lifetime
No-Pwr.Product=1000
MEAN COST PER HOUR
NO. OF PROCESSORS
1.00
0.80
0.60
0.40
0.20
0.00
0.00
2.00
4.00
6.00
8.00
10.00

Figure 19(c).

$$Variance\ Approximate\ Cost\,(VAC) \equiv \int_0^\infty Var\{\beta(t)\}\,dL(t) \tag{37}$$

It is easy to see that $MAC > Mean\ Cost$ always, and that the approximate costs approach the accurate costs when the probability of dynamic failure is small. Indeed, the anomaly does not appear until the probability of dynamic failure is significant. Since the applications under consideration are all critical processes, p_{dyn} is always small for the accepted configurations, and the configurations that exhibit this anomaly will be rejected by the p_{dyn} pass-fail test, and their mean costs need never be computed.

C. Extension

The number-power tradeoff can easily be extended to make it very useful in the process of design. The reader will have noticed that in the cost functions with which we measure the goodness of controller performance, no account is taken of controller hardware cost. All that the cost functions express is the control overhead incurred in actually *running* the process. Indeed, we may regard the mean costs as average *operating* overheads. It is not easy directly to incorporate the hardware cost into the cost functions themselves. Instead, one may consider the set of hardware configurations available for a particular hardware cost outlay. Then, *constant-cost* plots can be drawn, showing the range of performance (in terms of probability of dynamic failure and average operating costs) that is available for any particular hardware cost outlay. From similar curves, one may arrive at the minimum finite average operating cost associated with a particular hardware cost given that specifications for the probability of dynamic failure are met.

This approach can easily be illustrated with the number-power tradeoff considered here. When the hardware cost of a processor is proportional to its processing speed, the curves in Figure 13 become dynamic failure curves for a particular hardware cost. Curves such as Figure 18 can be used to identify the configurations that meet requirements for the probability of dynamic failure for given mission lifetimes, and the sensitivity of p_{dyn} to changes in mission lifetime.

Also, one can study any other tradeoffs that may exist between the hardware cost or the number-power product and the minimum mean cost per lifetime associated with such a cost or product.

The computer studied here is simple; however, it can be extended in some useful directions relatively easily. It is easy to take care of the case when the hardware cost or the finite cost function is a more complicated function of the processing speed. More complicated multiprocessors require a more involved analysis, but the basic ideas should now be clear.

CHAPTER 6

SYNCHRONIZATION AND FAULT-MASKING

A. Synchronization

Figure 20 is a schematic showing the handling of data as it enters the system through the sensors, and leaves it (in a figurative sense) at the actuators. Synchronization and fault-masking are integral parts of any fault-tolerant distributed system.

When a multiplicity of processors executes code in parallel, care must be taken to keep them reasonably in step. Therefore, the issue of synchronization is focal to all methods of forward error recovery. There are two basic methods of synchronization:

(1). Each processor has an ultra-precise clock. When the computer is switched on, the clocks are synchronized. If the clocks are sufficiently precise, the processors will continue to run in lock-step for an appreciable period. Unfortunately, such highly precise clocks are extremely expensive to build, and unsuited to incorporation in computer circuits. (For a description of highly precise clocks, see [19]). Clocks that are generally used in computer circuits drift too rapidly for this method to be employed in practice. We shall not

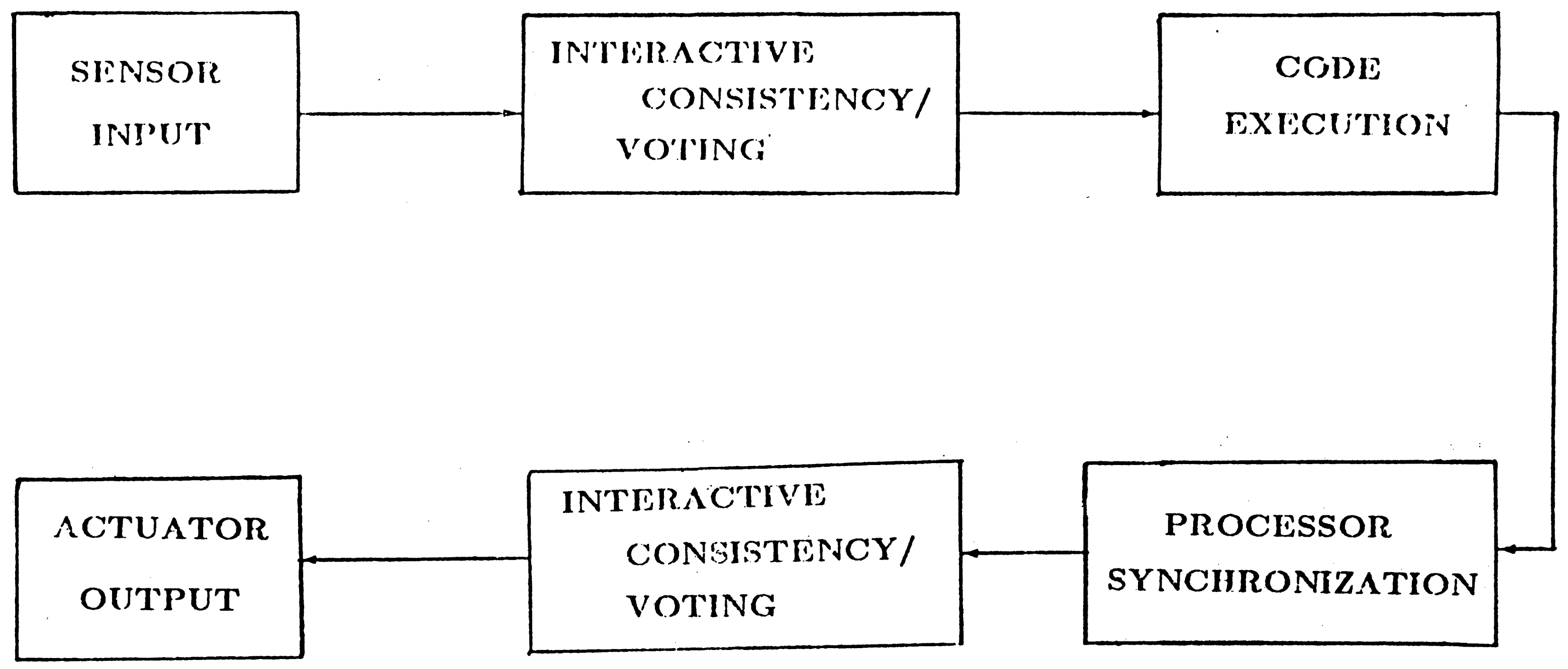

SENSOR INPUT
INTERACTIVE CONSISTENCY/ VOTING
CODE EXECUTION
PROCESSOR SYNCHRONIZATION
INTERACTIVE CONSISTENCY/ VOTING
ACTUATOR OUTPUT

Figure 20. Functional Block Diagram of a Real-Time System.

consider this method any further.

(2). The synchronization is carried out mutually. There is no single component whose functioning is critical to the security of the whole system. One may choose to synchronize the processor clocks, or the processors themselves at pre-defined boundaries of software execution. In the first case, one has a system operating more or less in lock-step, such as the FTMP system [1]. Both methods of synchronization are based on the same basic concepts; the only difference is the frequency with which synchronization is carried out.[1] The notion of *virtual time sources* now arises naturally. These are not necessarily clocks in the traditional sense; they mark the points at which an individual processor performs synchronization. It is convenient to view them as *virtual clocks,* whose transitions represent either clock "ticks" or execution of a stretch of code up to a pre-specified boundary. In the sequel, unless it is otherwise stated, the term "clock" is used to mean "virtual clock".

When synchronization is mutual, no "absolute" underlying time-source exists, only a set of time-sources whose relative behavior must be kept in step. The synchronizer (which may or may not be a physical part of the processor and which may be implemented either in hardware or in software) must therefore in each case have a perception of the state of the other time-sources. This perception may or may not be identical to that of the other synchronizers: if faulty modules necessarily behave consistently with respect to all synchronizers, it is identical; otherwise it need not be so.

[1] An important corollary of this is that the maximum clock drift rates that can be tolerated decrease with a decrease in the frequency of synchronization.

The synchronization process contributes to the system overhead in two ways. Firstly, there is the overhead imposed by the synchronization task itself. Secondly, the task-to-task communication overhead is proportional to the degree of synchronization achieved.

If hardware synchronization with phase-locked clocks is employed, the synchronization overhead can be reduced to vanishing point. If software synchronization is used, the overhead is significant. Both approaches to synchronization will be considered in succeeding sections. First, however, we will consider the second component.

Because of severe timing constraints, real-time systems do not generally use sophisticated mechanisms for task-to-task communication. Typically, data are transmitted from one task to another via timing rules agreed in advance. As a result, the receiving task has to wait for a time equal to the sum of the maximum transmission time and the maximum possible clock skew before it can read the data. Where synchronization is carried out in software and depends on the transmission of timing data on regular data channels, this transmission delay feeds back to increase the synchronization delay itself. We will consider this matter in detail in the sequel.

In what follows, we present a detailed discussion on each of hardware and software implementations of synchronization.

Hardware Synchronization

In this section, we consider synchronization by phase locking. Phase-locked clocks were first used to ensure that the processors of FTMP [1] operated in lock step. We consider a total of N clocks to be synchronized in the face of up to m

faulty clocks. The clocks are at the nodes of a completely connected graph. The basic theory behind their operation is simple. In Figure 21, we provide a schematic diagram of an individual clock. Each clock consists of a receiver which monitors the clock pulses of the N-1 other clocks in the arrangement, and these are used to generate a reference signal. By comparing this reference with its own pulse, the receiving clock computes an estimate of its own phase error. This estimated phase error is then put into an appropriate filter, and the output of the filter controls the clock oscillator's frequency. By thus controlling the frequency of the individual clocks, they can be kept in phase-lock and therefore synchronized for as long as the initial phase error is below a prescribed bound, i.e. for as long as the clocks started reasonably in step and their drifts are sufficiently low. A discussion of clock stability is provided in [21].

The arrangement for N=4, m=1 is, to our knowledge, the only phase-locked clock constructed and fully analyzed [20]. Unfortunately, when one attempts to increase m without care, synchronization can be lost due to the presence of malicious faults. In this section, we show how to design phase-locked clocks to tolerate a given arbitrary number of malicious failures. Our work is a generalization of the original design [20] which can tolerate at most one failed clock.

The following notation and definitions are used in this section.

Definition 1: If the overall system of clocks is properly synchronized, all individual non-faulty clocks must agree closely with each other. A well-synchronized system thus has *global clock cycles*. Global clock cycle i is the interval between the i*-th* tick of the fastest non-faulty clock (i.e. the non-faulty clock that has its i*-th* tick before that of all the other non-faulty clocks) and the (i+1)*-th* tick of the fastest

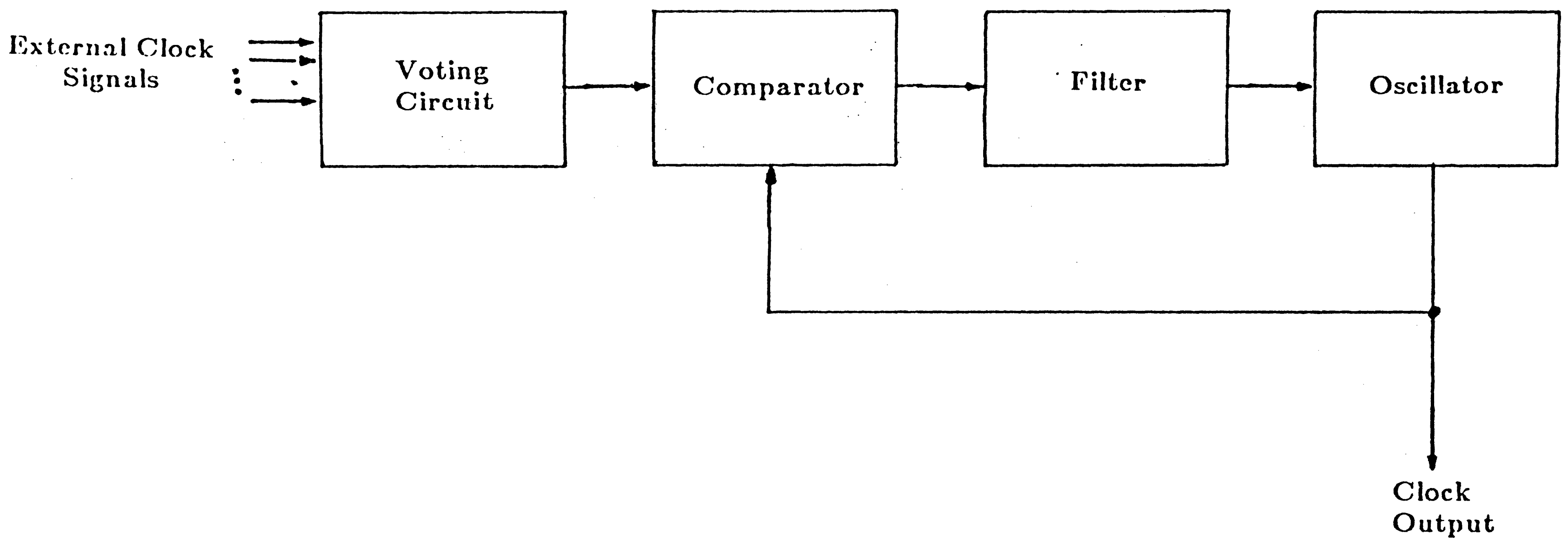


Figure 21. Component of a Fault-Tolerant Phase-Locked Clock.

non-faulty clock. For brevity, we shall denote global clock cycle *i* by *gcci*.

Definition 2: Each of the clocks "sees" through its receiving circuitry, the ticks of the other clocks. These ticks, together with the receiving clock's own tick, can be totally ordered in any *gcci* by the relation "prior or equal to". Such an ordered set, called a *scenario,* for clock α in *gcci* is denoted by S_{α}^{i}. We shall frequently drop the superscript for convenience: where this is done, it will be understood that we are talking about some *gcci*.

If a non-faulty clock *c* does not receive a tick from clock *d* within a given timeout period in any global clock cycle, the tick for *d* is arbitrarily assumed by *c* to be at the end of that timeout period. The scenario of every non-faulty clock therefore has exactly N elements.

Definition 3: If clock *a* has clock *b* as its reference in some *gcci*, it is said to *trigger* on *b* in that *gcci*.

Definition 4: Given the various triggers, we can draw a directed graph with the clocks as the vertices, and the directed arcs reflecting the relationship "triggers" in some *gcci*. Such a graph is called the ***trigger graph***. For example, in Figure 22, *a* triggers *b* and *c*, and is itself triggered by *d*, while *d* is triggered by *b*. A ***clique*** of clocks is a component of the trigger graph. In Figure 22, there are two cliques: $\{a,b,c,d\}$ and $\{e,f,g\}$.

Notation: G and NG are the set of clocks and non-faulty clocks, respectively, in the system. There are N clocks in all, and up to m failures must be sustained.

Definition 6: A *partition* of G is defined as a set $P=\{G_1,G_2\}$, where G_1 and G_2 are

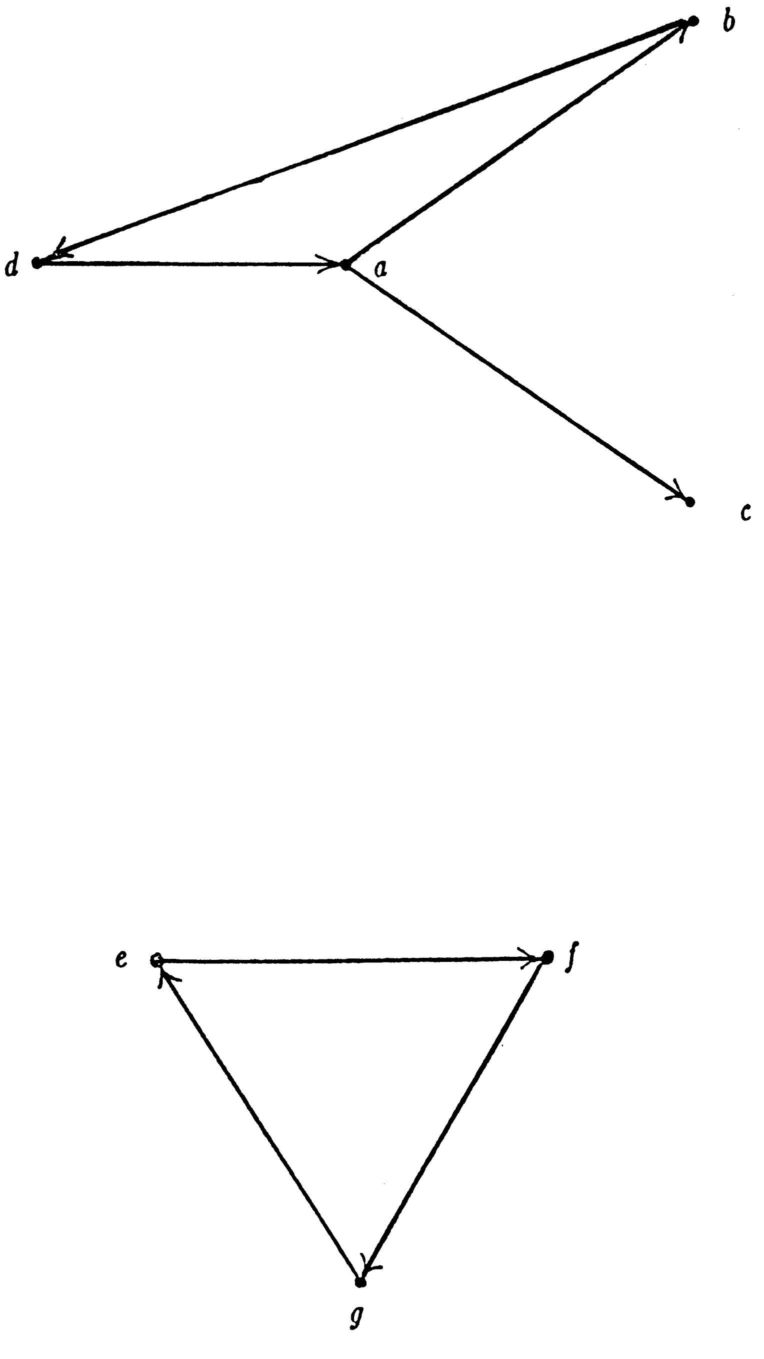


Figure 22. Trigger Graph: An Example

subsets of G with the following properties:

(i) $G = G_1 \bigcup G_2$

(ii) $G_1 \bigcap G_2 \bigcap NG = \phi$,

(iii) $G_i \bigcap NG \neq \phi$, i=1,2.

From (i), each clock must belong to at least one of G_1 and G_2. From (ii), only faulty clocks may belong to both G_1 and G_2. From (iii), there must be at least one non-faulty clock in each of G_1 and G_2.

Definition 7: A clock a is said to be *faster* than a clock b in scenario S if a precedes b in S. In a partition $P=\{G_1,G_2\}$, G_1 is said to be faster than G_2 if every non-faulty clock in G_1 is faster than every non-faulty clock in G_2.

Notation: Given a partition $P=\{G_1,G_2\}$, NG_1 and NG_2 are the non-faulty clocks in G_1 and G_2, respectively. By definition 6, neither NG_1 nor NG_2 can be empty and $NG_1 \bigcap NG_2 = \phi$.

Definition 8: Cliques A and B (of clocks) are said to be non-overlapping if the non-faulty clocks of A are either all faster than those of B, or vice versa.

Notation: Denote the position of a clock c in its own scenario S_c^i in *gcci* by p_c^i. Again, we shall frequently drop the superscript for convenience. The reference signal (i.e. the trigger) is a function of N and of p_c. It is denoted by $f_{p_c}(N)$. By this, we mean that clock c triggers on the $f_{p_c}(N)$*-th* signal in S_c, not counting itself.

For the system to operate satisfactorily, all the non-faulty clocks must have their ticks close together. Also, they should tell good time, i.e. the length of every global clock cycle should be about the length of an ideal (or absolute time) clock's inter-tick interval. These conditions dictate the following two *conditions of correctness* C1 and C2.

Definition 9: Each of the following *conditions of correctness* must be satisfied in *gcci* if the system is to be correctly operating in every *gcci*.

C1. For all partitions $P=\{G_1,G_2\}$ of the set of clocks G, in which the non-faulty clocks in G_1 are all faster than those in G_2, each of the following (K1 and K2) must apply:

K1. If, in *gcci*, all clocks in NG_1 trigger on clocks in G_1, then there is at least one clock in NG_2 that triggers on a clock in G_1. Furthermore, if no clock in NG_2 triggers on a clock in NG_1, at least one clock $k \in NG_2$ must trigger on a faulty clock $h \in G_1$ such that in the scenario S_k, there is at least one clock $r \in NG_1$ that is slower than the clock h.

K2. If, in *gcci*, all clocks in NG_2 trigger on clocks in G_2, then there is at least one clock in NG_1 that triggers on a clock in G_2. Furthermore, if no clock in NG_1 triggers on a clock in NG_2, at least one clock $k \in NG_1$ must trigger on a faulty clock $h \in G_2$ such that in S_k, there is at least one clock $r \in NG_2$ that is faster than h.

C2. If a non-faulty clock x triggers on a faulty clock y, then there must exist non-faulty clocks z_1 and z_2 such that z_1 is faster than or equal to y, and y is faster than or equal to z_2. Either z_1 or z_2 may be x itself.

Intuitively, we may regard C1 as preventing the formation of non-overlapping cliques -- which would obviously destroy synchrony -- and C2 as ensuring that the system keeps good time, i.e. that each global clock cycle is close to being the clock cycle of an ideal clock.

Finally, we assume that the transmission of clock signals through the system takes negligible time. This ensures that all non-faulty clocks are seen by all clocks in the same mutual order.

The phase-locked clock system for $N=4$, $m=1$ is simple enough to be proved correct by an exhaustive enumeration of all eventualities. It is, to our knowledge, the only phase-locked clock actually constructed [20].

Here, the reference used is the second incoming pulse (in temporal order), i.e. the median pulse. Such a clock is proof against the malice of a single faulty clock. To give the reader a feeling for why this is so, and to enhance his intuition about malicious failure, we provide below a simple explanation.

Call the four clocks *a*, *b*, *c*, and *d*. Let *d* be the maliciously faulty clock. Because *d* is malicious, it may provide different timing signals (i.e. lie) to different receivers. Since the non-faulty clocks by definition send their ticks at the same moment (or do not lie) to all the other receiving clocks, the mutual ordering of the non-faulty clocks within every scenario is the same for all non-faulty clocks. That is to say, if clock *b* sees clock *a* faster than clock *c* in some *gcci* (i.e. clock *a* sends its *i-th* tick to *b* before clock *c* does so), then *a* will appear faster than *c* to both the other non-faulty clocks in the system, i.e. to *a* and *c* in that *gcci*. *d*, however, may appear in different positions in the scenarios of the non-faulty clocks since it is malicious. One way of proving that a four-clock arrangement works despite *d*'s being malicious, is to enumerate all possible actions of *d* and show that the system still continues to satisfy the conditions of correctness.

Assume without loss of generality that a is prior or equal to b which in turn is prior or equal to c in some *gcci*. Consider a sample set of scenarios for our four-clock example. The triggering clock is denoted in bold-face type.

$S_a = a \leq b \leq \mathbf{c} \leq d$

$S_b = a \leq \mathbf{d} \leq b \leq c$

$S_c = a \leq \mathbf{b} \leq d \leq c$

The scenario S_d is irrelevant, since d is faulty.

Notice first that the position of the faulty clock d changes relative to the others, while the mutual ordering of the non-faulty clocks remains unchanged, as indeed it should.

It is easy to see that both conditions of correctness will be satisfied, and that the clock will operate correctly if the above scenario holds. It is not difficult to write down all the $4^3=64$ possible scenarios (with the ordering of the non-faulty clocks fixed as above) that are made possible by the arbitrary positioning of d, and to convince oneself that, for all possible scenarios, C1 and C2 are satisfied.

Unfortunately, if we try to allow for $m=2,3,...$, by expanding the system arbitrarily without sufficient care, the conditions of correctness can be violated. In fact, it is even possible for a system to contain an arbitrarily large number of clocks, and still to be vulnerable to just two malicious failures.

To see this, consider the following example. Let us choose, for each clock y in the system, $f_{p_y}(N)$ as the median clock signal in the scenario, not counting clock y. If N is odd (and there is thus an even number of "other" clocks), choose the slower of the two middle clocks. Then, $f_{p_i}(N)$ is only a function of N. We therefore drop the subscript for this example. Choosing the median signal is certainly good intuition.

Let there be only two faulty clocks, x_1 and x_2, and n=N-2 non-faulty clocks a_1, ..., a_n.

Case 1: $N \geq 7$. Consider some *gcci*. Assume that a_k is faster than a_l in *gcci* if $k<l$. Now, let x_1 and x_2 present themselves as the fastest two clocks to a_1, ..., a_p, and as the slowest two clocks to the other non-faulty clocks, i.e. a_{p+1},...a_n, where $p = \lceil n/2 \rceil = f(N)-1$. Then, the set of scenarios can be represented as in Figure 23.

Recalling that a clock triggers on the $f(N)$-*th* tick in its scenario ***not counting itself,*** we can draw the trigger graph as in Figure 24. It follows that $\{a_1,..., a_p\}$ and $\{a_{p+1}, ..., a_n\}$ will be two non-overlapping cliques, no matter how large n may be. It is easy to work out the case for N=7 to convince oneself of this fact.

Case 2: $N \leq 7$. This is trivial, and showing that the system is incapable of sustaining even two maliciously faulty clocks is left to the reader.

This has been a cautionary tale of the unbridled use of intuition in designing phase-locked clocks. Assured now that a more careful approach is needed, we turn in the following section to showing how to expand phase-locked clocks.

Our job is to (i) find the lower bound, N, on the size of a system of clocks that must sustain up to m maliciously faulty clocks, and (ii) find the functions $f_x(N)$ for x=1,...,N.

We begin with the following two lemmas that smooth our way to the main result.

$$\begin{array}{llll}
S_{a_1}: & x_1\ x_2\ a_1\ a_2 & \cdots & a_n \\
S_{a_2}: & x_1\ x_2\ a_1\ a_2 & \cdots & a_n \\
S_{a_3}: & x_1\ x_2\ a_1\ a_2 & \cdots & a_n \\
 & & \vdots & \\
S_{a_r}: & x_1\ x_2\ a_1\ a_2 & \cdots & a_n \\
 & & & \\
S_{a_{r+1}}: & a_1\ a_2 & \cdots & a_n\ x_1\ x_2 \\
S_{a_{r+2}}: & a_1\ a_2 & \cdots & a_n\ x_1\ x_2 \\
S_{a_{r+3}}: & a_1\ a_2 & \cdots & a_n\ x_1\ x_2 \\
 & & \vdots & \\
S_{a_n}: & a_1\ a_2 & \cdots & a_n\ x_1\ x_2
\end{array}$$

Figure 23. Scenarios in Example

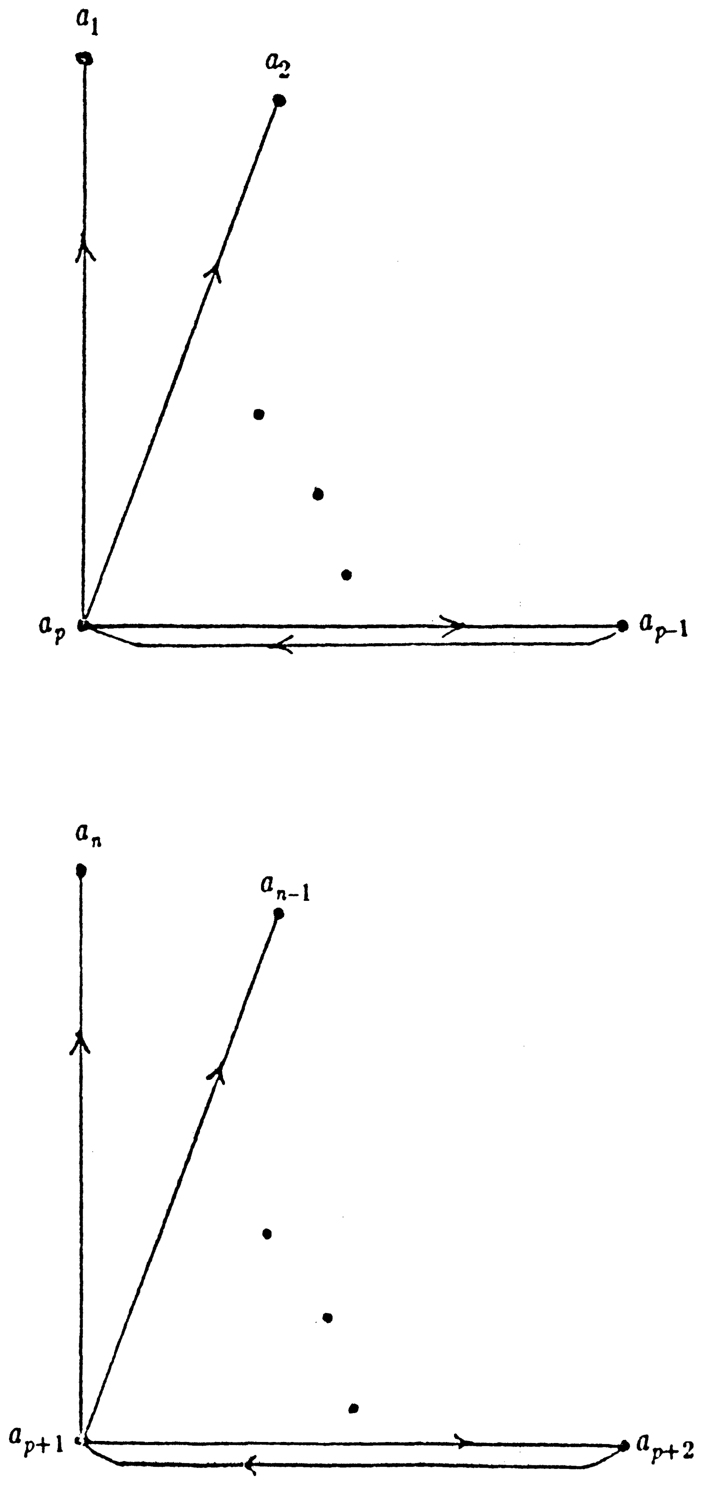


Figure 24. Trigger Graph for Scenarios in Figure 23

Lemma 1: Condition C2 is satisfied for all partitions $P=\{G_1,G_2\}$ if and only if there exist functions $f_x(N)$ for $x=1,\ldots,N$, such that

$$\min\{m,\ x-1\} < f_x(N) < \max\{N-m,\ x\} \tag{38}$$

Proof: Let k be a non-faulty clock such that $p_k = x$. We must show that Eq. (38) holds for all x for which p_k is defined iff condition C2 holds.

Suppose that there exist functions $f_x(N)$ for $x=1,\ldots,N$ satisfying Eq. (38). This implies $\min\{m,\ x-1\}+1 < \max\{N-m,\ x\}$ for all $x\in\{1,2,\ldots,N\}$, leading to $N>2m+1$. Hence, it is sufficient to consider the following three cases:

(i) $x\leq m$:

Clearly, $\max\{N-m,\ x\} = N-m$, $\min\{m,\ x-1\} = x-1$ and therefore $x-1 < f_x(N) < N-m$. If the reference clock is non-faulty, we have nothing to prove. If it is faulty, then since there are at most m faulty clocks, there must be at least one non-faulty clock slower than the reference clock. Also, from the left half of the inequality, $f_x > x-1$, and since clock k is non-faulty, there is a non-faulty clock (i.e. k itself) faster than the reference clock. So, C2 is satisfied.

(ii) $N-m\geq x>m$:

$\min\{m,x-1\}=m$, $\max\{N-m,x\}=N-m$ and therefore $m+1 \leq f_x(N) \leq N-m-1$. Since at most m faulty clocks exist, if the reference clock in S_k were faulty, it must appear in S_k as slower than at least one non-faulty clock (the right half of the inequality), and faster than at least one non-faulty clock (the left half of the inequality), and C2 is satisfied.

(iii) $N \geq x > N-m$:

> $\min\{m,x-1\}=m$, $\max\{N-m,x\}=x$ and $m+1 \leq f_x(N) \leq x-1$. As with the previous cases, there must appear in S_k at least one non-faulty clock that is faster than the reference clock, if the reference clock is faulty. Also, since k is non-faulty, and appears in the x-th (i.e. p_k-th) position, there is at least one non-faulty clock, in particular clock k, that is slower than the reference clock in S_k, thus satisfying C2.

Conversely, suppose $f_x(N) \leq \min\{m,x-1\}$. Then, C2 is violated when faulty clocks appear in positions $1,\ldots,f_x(N)$ of S_k. Similarly, if $f_x(N) \geq \max\{N-m,x\}$, C2 is violated when faulty clocks appear in positions $f_x(N)+1,\ldots,N$ of S_k.[2] **Q.E.D.**

Lemma 2: If all clocks in NG_1 trigger only on clocks in G_1 (where the notation is the same as in definition 9), then the following are equivalent:

(i) $q_1 \geq \min\limits_{k \in NG_2} f_{p_k}(N)$ where q_1 is the number of non-faulty clocks in G_1.

(ii) K1 is satisfied.

Proof:

(i) implies (ii): If (i) holds, then it is easy to see that no matter how the up to m faulty clocks in G arrange themselves, K1 is satisfied.

(ii) implies (i): Suppose, to the contrary, that $q_1 < \min\limits_{k \in NG_2} f_{p_k}(N)$. Consider the nonempty set $L = \{y : y \in NG_2 \text{ and } f_{p_y}(N) = \min\limits_{k \in NG_2} f_{p_k}(N)\}$. Assume that there are $i \leq m$ faulty clocks in G_1. Since the faulty clocks may present themselves in any

[2] Once again, the addition of 1 occurs because a clock does not count itself when counting to $f_x(N)$.

position in any scenario, consider the case where they present themselves in the scenario of every $y \in L$ in the $q_1+1,\ldots,q_1+i$ positions. Then, there is no non-faulty clock in G_1 that is slower than the reference clock of any clock in NG_2, a contradiction. **Q.E.D.**

The two theorems below yield the main result of this section.

Theorem 1: To ensure that, despite up to m malicious failures, the conditions of correctness are satisfied, the system must have $N \geq 3m+1$ clocks.

Proof: We will only consider here the case of partitions $P = \{G_1, G_2\}$ in which all clocks in NG_1 trigger on clocks in G_1. The other case (i.e. K2) can similarly be dealt with.

Let there be q_1 and q_2 clocks respectively in NG_1 and NG_2. Let $M = \{y : y \in NG_1 \text{ and } f_{p_y}(N) = \max_{k \in NG_1} f_{p_k}(N)\}$. Let i be the number of faulty clocks that belong to G_1. Then, the assumption that all non-faulty clocks in G_1 trigger on clocks in G_1 is equivalent to saying that one of the following Eqs. (39) and (40) must apply:

$$q_1+i \geq \max_{k \in NG_1} f_{p_k}(N) + 1 = f_{p_y}(N) + 1 \tag{39}$$

which applies if there exists at least one p_y, $y \in M$, such that $p_y < f_{p_y}(N)$. The addition of 1 follows from the fact that clock y does not count itself when counting to $f_{p_y}(N)$. If $p_y \geq f_{p_y}(N)$ for all $y \in M$, the following Eq. (40) applies:

$$q_1+i \geq \max_{k \in NG_1} f_{p_k}(N) \tag{40}$$

First consider the case where Eq. (39) applies. The condition that C1 (more specifically, K1) holds implies, from Lemma 2, that

$$q_1 \geq \min_{k \in NG_2} f_{p_k}(N) \tag{41}$$

Since this must be true for all partitions of G, we have for all $q_1 \in \{1,\dots,N-i-1\}$:

$$q_1 \geq \max_{k \in NG_1} f_{p_k}(N) - i + 1$$

if $q_1 \geq \min_{k \in NG_2} f_{p_k}(N)$. Hence, K1 can be written as:

$$\text{For all } q_1 \in \{1,\dots,N-i-1\}, \quad \left\{ q_1 \geq \max_{k \in NG_1} f_{p_k}(N) - i+1 \;\supset\; q_1 \geq \min_{k \in NG_2} f_{p_k}(N) \right\}$$

In particular, this is true for $q_1 = \max_{k \in NG_1} f_{p_k}(N) - i+1$. Thus,

$$\max_{k \in NG_1} f_{p_k}(N) - i+1 \geq \min_{k \in NG_2} f_{p_k}(N)$$

or

$$\max_{k \in NG_1} f_{p_k}(N) - \min_{k \in NG_2} f_{p_k}(N) \geq i-1 \tag{42}$$

Recall that this is true if Eq. (39) applies. Similarly, if Eq. (40) applies, we have from an identical argument,

$$\max_{k \in NG_1} f_{p_k}(N) - \min_{k \in NG_2} f_{p_k}(N) \geq i \tag{43}$$

Eqs. (39)-(43) must hold for all possible i. Since there are at most m faulty clocks, we must have:

$$\max_{k \in NG_1} f_{p_k}(N) - \min_{k \in NG_2} f_{p_k}(N) \geq m-1 \tag{42'}$$

if Eq. (39) applies, and

$$\max_{k\in NG_1} f_{p_k}(N) - \min_{k\in NG_2} f_{p_k}(N) \geq m \qquad \textbf{(43')}$$

if Eq. (40) applies.

We first consider the case where Eq. (39) applies. We claim that it implies that $N>3m$.

To see why, let y be the slowest clock in M and z the slowest clock in L (with L defined as in Lemma 2). Then, due to Lemma 1 and Eq. (42') the following inequality must hold:

$$\max\{N-m,\, p_y\} > \max_{k\in NG_1} f_{p_k}(N) \geq m-1+\min_{k\in NG_2} f_{p_k}(N) > m-1 + \min\{m,\, p_z-1\} \qquad (44)$$

Then up to m faulty clocks in the system can arrange themselves in any order. In particular, they can so order themselves in S_y that $p_y \leq N-m$, and so order themselves in S_z that $p_z > m$. Since Eq. (44) must hold *always*, no matter what the faulty clocks do, we must have:

$$N-m > \max_{k\in NG_1} f_{p_k}(N) \geq m-1+\min_{k\in NG_2} f_{p_k}(N) \geq (m-1) + (m+1) \qquad (45)$$

from which we arrive at the equation

$$N>3m \qquad (46)$$

Recall that this applies whenever Eq. (39) holds. If, instead, Eq. (40) applies, we can similarly show that

$$N>3m+1 \qquad (47)$$

Since we seek the smallest N to satisfy the conditions of correctness, we have done if we can show that there exist functions $f_x(N)$ such that Eq. (39) always applies (and therefore Eq. (40) never applies), and for which Eq. (45) is satisfied. But, we can always construct $f_x(N)$ to (i) be monotonically non-increasing functions of x and (ii) satisfy Eq. (45): an example of such a construction is provided in the statement of Theorem 2 below. Hence Eq. (39) always applies, and $N \geq 3m+1$, is the necessary condition.

The case when all clocks in NG_2 trigger on clocks in G_2 can be similarly treated. **Q.E.D.**

Theorem 2: If $N \geq 3m+1$ and $f_x(N) = \begin{cases} 2m & \text{if } x < N-m \\ m+1 & \text{if } x \geq N-m \end{cases}$ then the conditions of correctness are satisfied.

Proof: $f_x(N)$ as defined here satisfies Lemmas 1 and 2 and is monotonically non-increasing in x. Clearly, C2 holds. Also, it is easy to see that <u>if</u> $N > 3m$ and Eq. (39) implies Eq. (41), <u>then</u> case K1 in Definition 8 will hold. We therefore only have to show that the definition of $f_x(N)$ as given above satisfies Eq. (41) if Eq. (39) is satisfied. This can easily be verified by a direct substitution.

Case K2 can be similarly seen to hold. **Q.E.D.**

It should be noted that the set of functions $f_x(N)$ is not always unique. From the proofs of Theorems 1 and 2, the following inequalities are sufficient:

(i) $m+1 \leq f_x(N) \leq N-m-1$ for all $x = 1,\ldots,N$.

(ii) $f_x(N) \geq m-1+f_{N-m}(N)$ for all $x \leq m+1$,

(iii) $f_{N-m}(N) \leq f_x(N) \leq f_{m+1}(N)$ for $N-m > x > m+1$,

(iv) $f_i(N) \geq f_j(N)$ iff $i \leq j$.

The intervals $x \geq N-m$ and $x \leq m+1$ arise from the up to m faulty clocks in the system. All that we can tell about the fastest non-faulty clock g in the system (this clock must have the maximum value of $f_x(N)$) in clock g's scenario is that it is in the first $m+1$ clocks in that scenario. Similarly, all that we can tell about the position of the slowest non-faulty clock s in the system (which must have the minimum value of $f_x(N)$) is that it occupies a place in the last $m+1$ clocks. This leads at once to the intervals $x \geq N-m$ and $x \leq m+1$.

It is interesting to note that if conditions C1 and C2 are both satisfied, and the functions $f_x(N)$ are monotonically non-increasing in x, then a stronger condition than C2 automatically holds.

Corollary: If the conditions of correctness are satisfied, with the $f_x(N)$ being defined as monotonically non-increasing functions of x, then the following condition C3 holds.

C3. Every non-faulty clock necessarily triggers on either a non-faulty clock, or a faulty clock that is sandwiched between the *other* non-faulty clocks.

Proof: Now, C3 follows immediately from C2 for all but the fastest and slowest non-faulty clocks.

Consider the fastest non-faulty clock. In the course of proving Theorem 1, it was established that $N-m > f_x(N) > m$ for all $x=1,\ldots,N$, and that $\max_{k \in G} f_{p_k}(N) - \min_{k \in G} f_{p_k}(N) \geq m-1$, leading to $2m$ as the smallest value for $\max_{k \in NG} f_{p_k}(N)$ where $NG \subset G$ is the set of non-faulty clocks. From the monotonic nature of the $f_{p_k}(N)$, the trigger for the fastest non-faulty clock must lie in the interval $2m+1, \ldots, N-m$. But, since $N \geq 3m+1$, any faulty clock in this interval must be sandwiched between non-faulty clocks.

The proof for the slowest non-faulty clock is similar. **Q.E.D.**

Remark 1: Synchronization Overhead

In the case of a phase-locked clock, there is some time overhead due to the oscillations that are possible as a result of malicious behavior. However, these are minimal when good crystal clocks are used, and so it is reasonable to treat the overhead of hardware synchronization as negligible. Also, the clock skew is very small/negligible in a well-designed phase-locked clock.

Remark 2: An Alternative Design

The only other hardware arrangement that we are aware of for keeping synchronization in the face of malicious behavior is the multi-stage synchronizer arrangement proposed by Davies and Wakerly [22]. The idea is shown in Figure 25. It consists of m stages of N synchronizers each. The system works on the principle that, with this redundancy, there must be at least one level of synchronizers that assures proper synchronization in the presence of malicious faults. An informal proof is provided in [22].

This arrangement results in a proliferation of hardware. As may readily be verified, the total number of devices (processors and synchronizers) in the cluster is $2m^2+3m+1$. The total number of I/O ports required is given by $8m^3+16m^2+10m+2$. The potential enormity of the above numbers should be driven home by the consideration that in order to maximize returns from redundancy, the individual modules must be isolated from one another as much as possible. This dictates that power supplies must also be replicated in large numbers, and that the benefits of large-scale integration cannot be brought to bear on the issue: individual synchroniz-

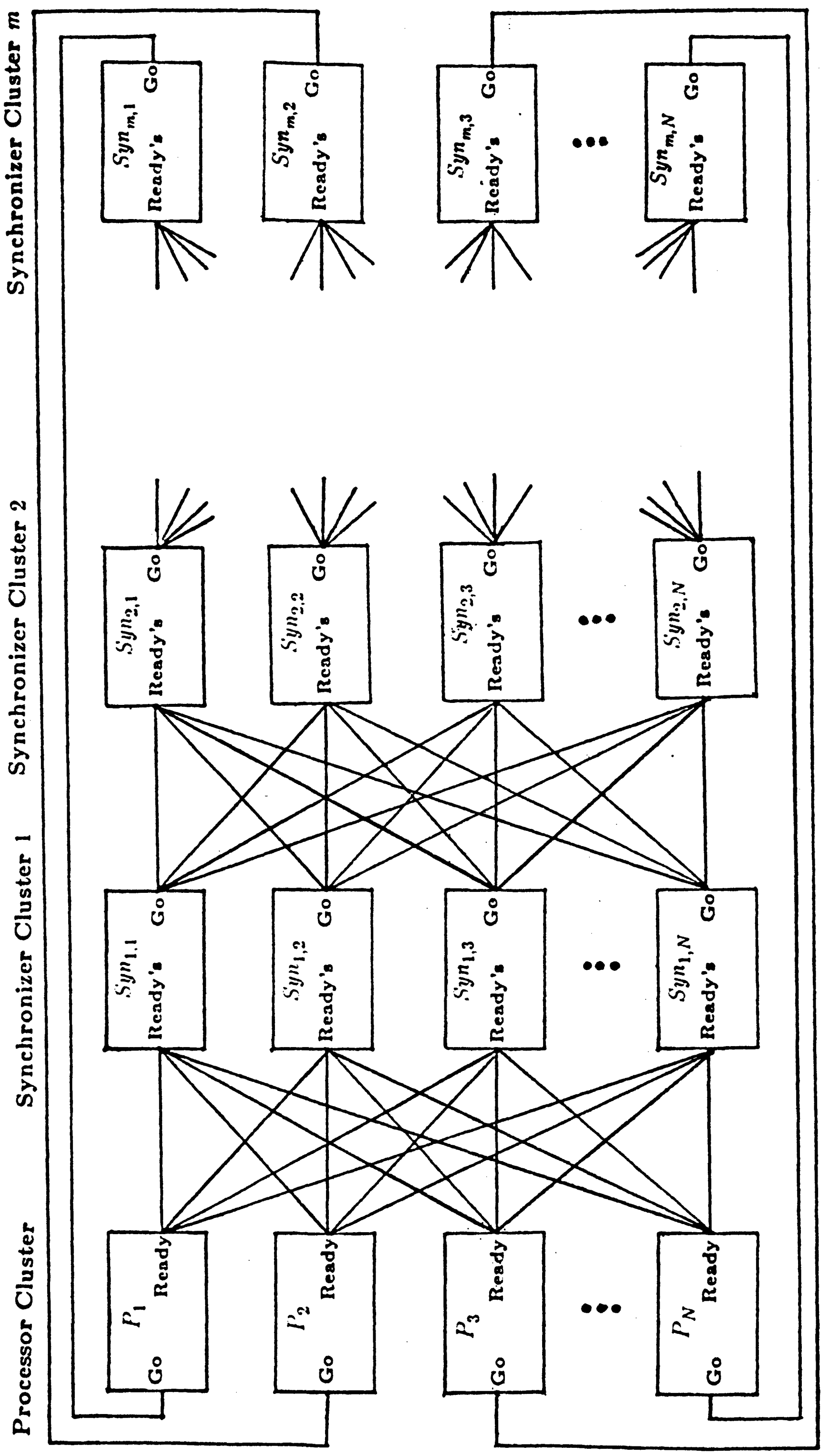


Figure 25. Davies and Wakerly's Multistage Synchronizer

ers must be on separate devices -- even, perhaps, on separate cards. Otherwise, correlated and common-cause failures could wipe out reliability gains made by device redundancy.

Compared with the gargantuan nature of the redundancy required by the Davies and Wakerly approach, the $N=3m+1$ requirement of phase-locked clocks represents an extremely elegant hardware solution to the problem of synchronization in the presence of malicious faults.

Software Synchronization

The use of decentralized algorithms for synchronization offers an alternative to the hardware methods described above. Such algorithms enable a system consisting of many processors with their own clocks to operate in close synchrony. The degree of synchronization obtained by these algorithms depends primarily on the performance of the communications system, the precision of the clocks, and the frequency of resynchronization. The task-to-task communications system's one-way message time is at least $B+\delta$ where B is the maximum transmission time and δ is the maximum clock skew. The most time-efficient of the software algorithms that we know of is the *interactive convergence* algorithm [23].

In the interactive convergence algorithm, each processor in the system determines its skew relative to every other processor in the system. If any relative skew is greater than a predetermined threshold, it is set to zero. An average of all the relative skews is calculated and used to correct its clock.

The following theorem (a trivial adaptation of one proved in [23]) characterizes the maximum clock skew of the system in terms of the following system parameters.

ϵ - maximum error in reading another processor's clock

ρ - maximum drift rate between any two clocks in the system

N - number of clocks in the system.

m - maximum number of faulty clocks accommodated.

R - resynchronization period.

$S(N)$ - execution time of the resynchronization task.

δ_0 - maximum clock skew at start-up.

Theorem 3 (adapted from [23]): If the following conditions hold:

$$3m<N$$

$$\delta \geq [1-\frac{3m}{N}-2\rho(1-\frac{m}{N})]^{-1}[2\epsilon\{1+\rho(1-\frac{m}{N})\} + \rho\{R + 2\frac{N-m}{N} S(N)\}]$$

$$\delta \geq \delta_0 + \rho R$$

$$\max(\delta, S(N)) < R$$

$$\rho\delta << \epsilon,$$

then, the non-faulty clocks remain in synchrony, i.e. the maximum skew is δ.

The synchronization algorithm is run periodically, the major component of the execution time usually being the time required to read every other processor's clock in the system. In the SIFT system, each processor's clock value is broadcast during a window of time allocated to it. There are N such windows, one for each processor in the system. All other processors wait during this window to receive the broadcast data value.

In order to accommodate the worst-case situation, each window must be at least $B+\delta$ long. The interactive convergence algorithm takes an execution time equal to $S(N)=N(B+\delta)+K$, where K is the time needed to compute and carry out the clock correction.

It should be noted that this execution time of the synchronization task affects the synchronization process itself. Indeed, since this is a function of N, there is a maximum cluster size that can be synchronized in this way. To see this, substitute the above expression for $S(N)$ in the formula for δ, and obtain:

$$\delta \geq N[N-3m-2\rho(N^2+N-mN-m)]^{-1}\ [2\epsilon+\rho\{R+2(N-m)(B+\frac{K}{N})\}] \tag{40}$$

From this, one can (a) compute the minimum execution time of the synchronization task as a function of the cluster size, (b) obtain the quality of synchronization (the smaller the δ, the better the synchronization), and (c) determine the largest possible cluster that can be thus synchronized: this is the largest N for which $S(N) < R$.

The values for the SIFT system are given by B=18.2 micro-seconds, and execution time for the synchronization task is 1.760 milli-seconds. Numerical results on the synchronization overhead using these values are plotted in Figure 26. The maximum cluster size permissible for synchronization is tabulated in Table 6.

Although the expression $S(N) = N(\delta+B)+K$ was presented as emanating from the SIFT system, it is easy to see that in *any* system where communication is by broadcast, and clock transmission slots are pre-determined, this expression will hold. It should also be reiterated that such communication protocols are the most commonly used protocols in real-time systems. In any case, it is obvious that whatever the protocol used, $S(N)$ is very unlikely to be less than a first order function of N.

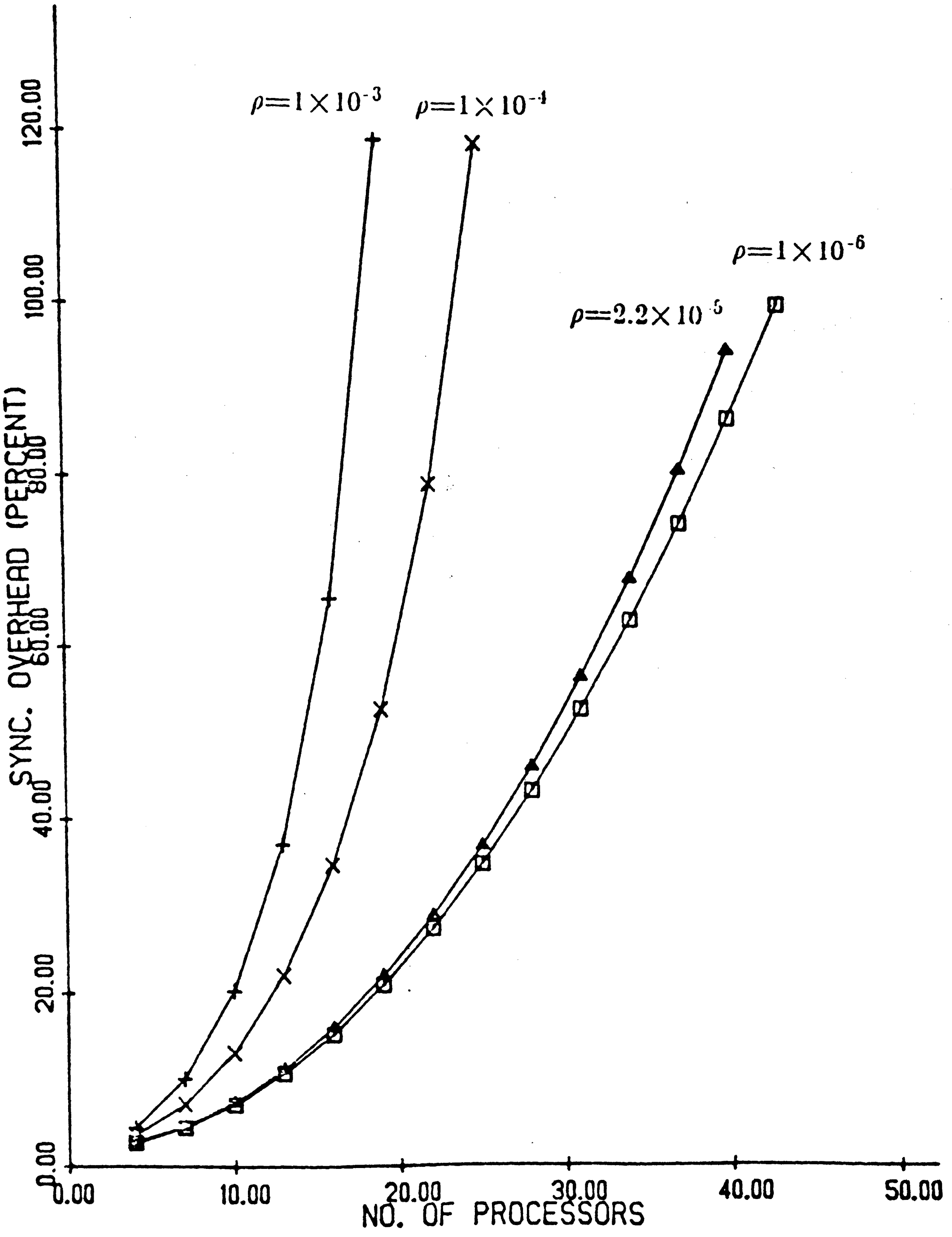


Figure 26. Software Synchronization Overhead

Drift Rate	Maximum Cluster Size
1×10^{-6}	43
5×10^{-6}	40
1×10^{-5}	40
2.2×10^{-5} **	40
5×10^{-5}	37
1×10^{-4}	34
5×10^{-4}	22
1×10^{-3}	16

** SIFT Value

Table 6. Maximum Cluster Size Permissible for Software Synchronization

Even if, in a hypothetical case, $S(N)$ were negligible (which, of course can never happen but nevertheless represents an extreme case), δ will continue to be a function of N, and there will be a point for which $\delta > R$, at which synchrony will break down.

B. Voting and Byzantine Generals Algorithm

Voting

Once the delay involved in synchronization is taken account of, there is very little additional delay if the voting is carried out in hardware. With software voting, however, the additional overhead can be significant.

Voting in software is carried out by individual processors placing data to be voted on in pre-specified "mailboxes" or "pigeonholes". The voter searches the mailboxes for valid data, fetches them, and then votes on them. The execution time in the retrieval step is directly proportional to the number of processors in the cluster, N. The execution time required to vote N values and diagnose up to m faults is at least $(N-1)C_1 + C_2$ but less than $[(N-1) + 2m - 2]\ C_1 + C_2 = [\frac{5(N-1)}{3} -2]\ C_1 + C_2$ where $C_1,\ C_2$ are some constants [24].

Experimental data exist for 3-MR and 5-MR in SIFT. These data can easily be introduced into the linear model obtained above. If s is the number of data values voted, and $V_N(s)$ the time taken for an N-way vote on s data values, the following expression was found to hold for SIFT:

$$V_N(s) = 58.5\ s\ N + 91.5s+38 \quad \textit{micro-seconds}. \tag{41}$$

This is a large overhead: in SIFT, for example, voting is performed at the beginning of a 3.2 milli-second subframe. If s=6, N=5, then 73% of the subframe is consumed by the voting algorithm [25].

Byzantine Generals Algorithms

The Byzantine Generals, or interactive consistency, algorithm must be used when it is necessary to isolate the sources of errors as well as to mask the errors themselves. It finds use when reconfiguration upon failure is to be attempted and the executive is distributed. The algorithm takes into account the fact that faulty processors may be malicious, in other words, that they need not fail only in "safe" directions. To be absolutely certain that faulty processors can be properly identified for isolation, it is necessary to allow for every possible misbehavior: thus the case when a faulty processor is malicious, i.e. that it actively and intelligently attempts to hide its malfunction, must also be handled. Such algorithms are typically used to reach agreement between processors in a cluster on incoming sensor data, and in certain clock synchronization algorithms. For further details, see [26-28].

The input of data is accomplished by every processor reading the external sources independently or by one processor reading the external sources and then distributing the obtained value to the rest of the processors. In the first case, each processor would very probably get a different value -- even if they were in perfect synchrony -- due to the inherent instability in reading analog data. Hence, a subsequent exchange of values read along with a mid-value selection is required to get a consistent value. However, this process suffers from sensitivity to malicious faulty processors and *interactive consistency* (or Byzantine Generals) algorithms are essential where fault isolation and reconfiguration are required.

The interactive consistency algorithm consists of the following steps:

(1) The source value is distributed to the N processors.

(2) The received values are exchanged m times to handle up to m faulty processors.

(3) A consistent value is obtained by use of a recursive algorithm. When $m=1$, this reduces to a majority calculation.

The overhead for these interactive consistency algorithms can be considerable. N must be at least $3m+1$. The number of messages required to obtain interactive consistency is of the order of N^{m-1}. To give an idea of the actual numbers incurred in practice, some experimental results from the SIFT computer [25] are used.

In SIFT, with five-way voting, only one fault can be located. The simple flight-control applications currently running in SIFT use 63 external sensor values, each of which goes through the interactive consistency algorithm. From the data collected, execution times for steps (1) and (2) of the algorithm can be estimated, and a lower bound determined for step (3). The following data were measured: step (1) : 3.05 *ms*, step (2) : 2.22 *ms*, and step (3) : 6.57 *ms* (total 11.84 *ms*). For larger m, the step (1) execution time should not change significantly, while the step (3) calculation would require at least 6.57 *ms* (very likely much more). The step (2) process consists of only message exchanges and thus varies directly with the number of messages which are sent. The following formula represents an approximate execution time for step (2) as a function of m: 2.22 N^{m-1} *ms*). We may add the timing values for steps (1) and steps (3) above to this this expression to obtain a lower bound for the overhead of the Byzantine Generals algorithm in SIFT. Since the interactive consistency tasks must be executed at the data sample rate, a large portion of the available CPU time is consumed: see Table 7.

These results indicate the extremely high overhead imposed in an attempt to achieve interactive consistency. It should be pointed out that there have lately been some more efficient implementations of the Byzantine Generals algorithm [29] than have been implemented on SIFT. However, even such implementations exhibit high overheads as the number of faulty modules to be accommodated increases.

C. Reconfigurable and Non-reconfigurable Systems

To locate faults after they have been detected by voting, diagnostic tests must be run. To ensure agreement amongst all non-faulty processors about the results of the tests, the interactive consistency algorithm must be executed.

Unfortunately, as we have seen, this algorithm is extremely time-consuming to run. Reconfigurable systems must therefore contend with a large overhead as compared to non-reconfigurable systems.

However, reconfigurable systems have the advantage of dynamic redundancy management. When widespread failures occur, it is possible to retire some clusters in order to keep others at full strength. Also, by periodically purging itself of faulty components, a reconfigurable system can survive in the face of more failures than can a non-reconfigurable system. For example, if one started operation with a 7-cluster, the reconfigurable system would not fail unless either (a) all but two processors fail, or (b) more than m (m=2 for a 7-cluster, and 1 for a 4-cluster) processors fail between successive tests, while the corresponding non-reconfigurable system would fail if more than 3 processors failed. This does not automatically mean that a

Data Sample Period	m=1	m=2	m=3
100 ms	11.8 %	25.1 %	>380 %
50 ms	23.7 %	50.2 %	>760%
33 ms	35.9 %	76.2 %	>1140%
25 ms	47.4 %	>100 %	>1520%

m = number of faulty processors accommodated

Table 7. Overhead of Byzantine Generals Algorithm: Lower Bound for SIFT

reconfigurable system is necessarily better than a non-reconfigurable one, since as we shall see, timing requirements impose severe constraints on the size of reconfigurable clusters.

We shall contrast the reliability of reconfigurable and non-reconfigurable systems with the following example. Assume that there is a single critical task in the system that requires 1.6 milli-seconds to run, and that this task is dispatched every 50 milli-seconds, and that the system must be ready to begin executing the task the moment it is released. There is a total of N processors available. Processors fail according to an exponential law with specified MTBF. The mission lifetime (duration between successive service stages) is also specified.

In the following sections, we consider non-reconfigurable and reconfigurable systems separately. In both cases, we assume that synchronization is by means of phase-locked clocks. Since these can be made arbitrarily reliable and are common to both reconfigurable and non-reconfigurable systems, we do not consider the probability of clock failure in what follows. Numerical results in the section on reconfigurable systems are based on the lower bounds obtained from SIFT. Processor failures are assumed to occur independently, forming a Poisson process with mean interarrival time 5×10^5 seconds.

Non-Reconfigurable System

Under the above assumptions, the probability of dynamic failure is simply equal to the probability of static failure, i.e. the probability that fewer than $\lceil N/2 \rceil$ processors fail over the mission lifetime. This probability is graphed in Figure 27.

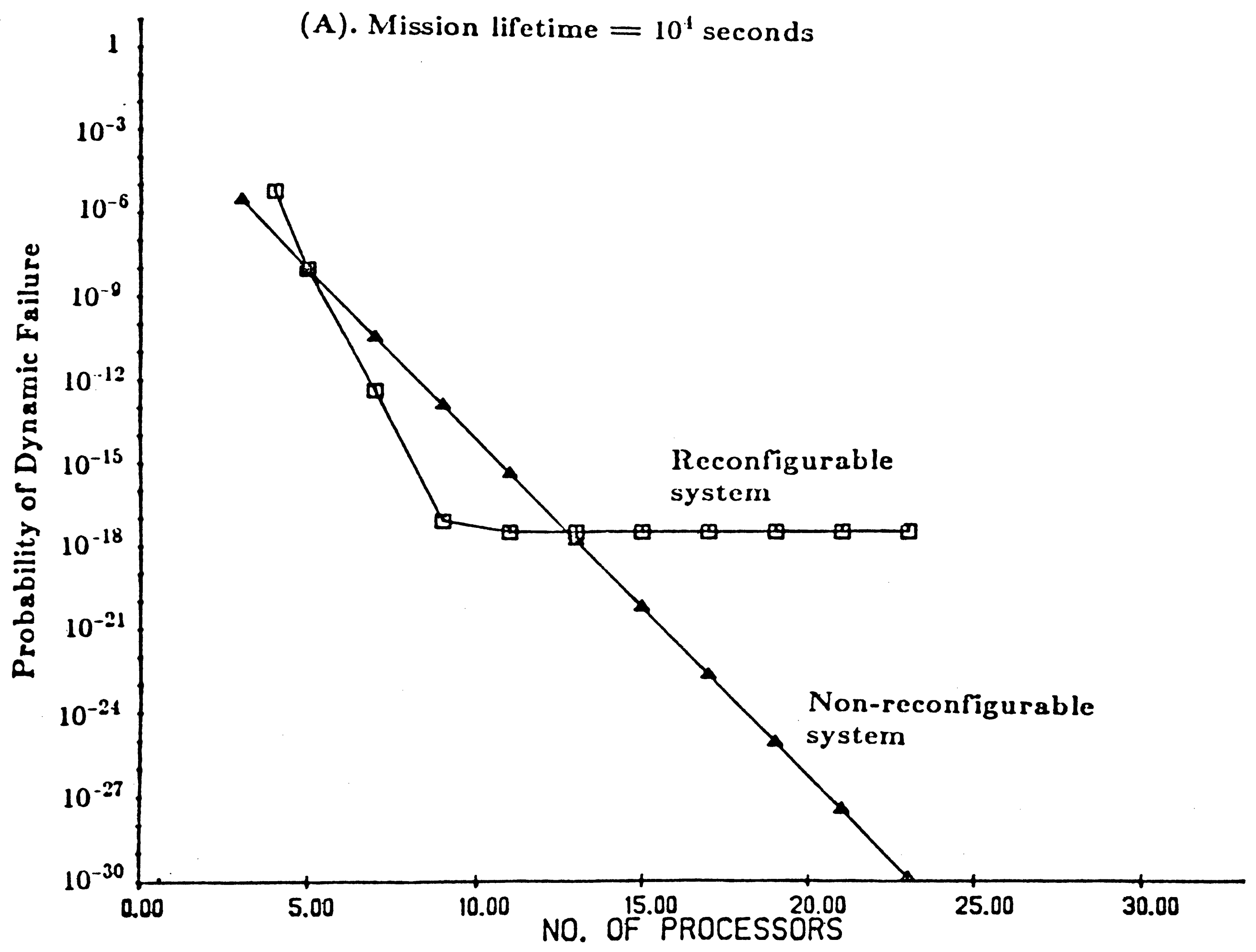


Figure 27. Comparison of Reconfigurable and Non-Reconfigurable Systems

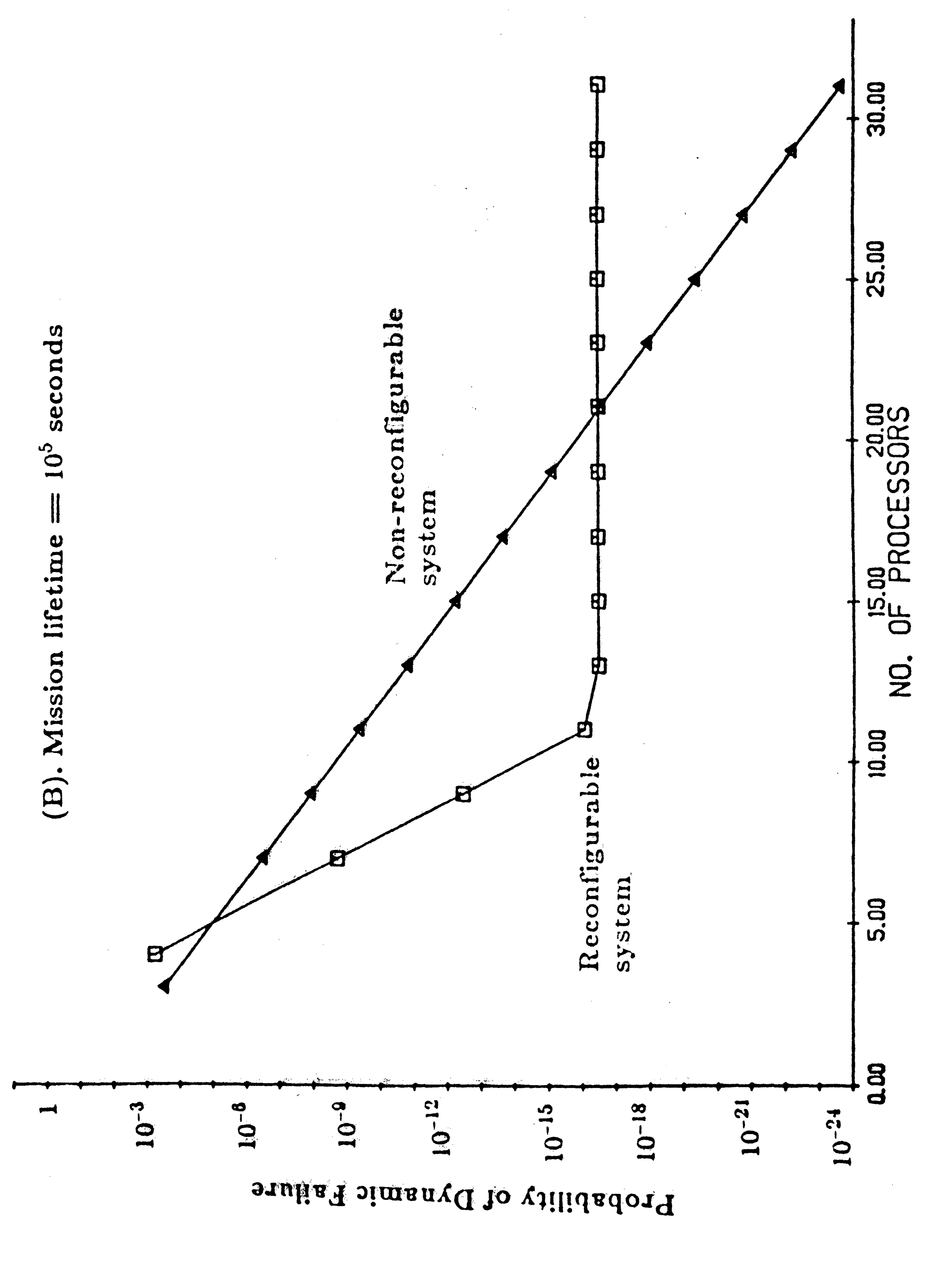

(B). Mission lifetime = 10^5 seconds
Non-reconfigurable system
Reconfigurable system
Probability of Dynamic Failure
1
10^{-3}
10^{-6}
10^{-9}
10^{-12}
10^{-15}
10^{-18}
10^{-21}
10^{-24}
0.00
5.00
10.00
15.00
20.00
25.00
30.00
NO. OF PROCESSORS

Figure 27(b)

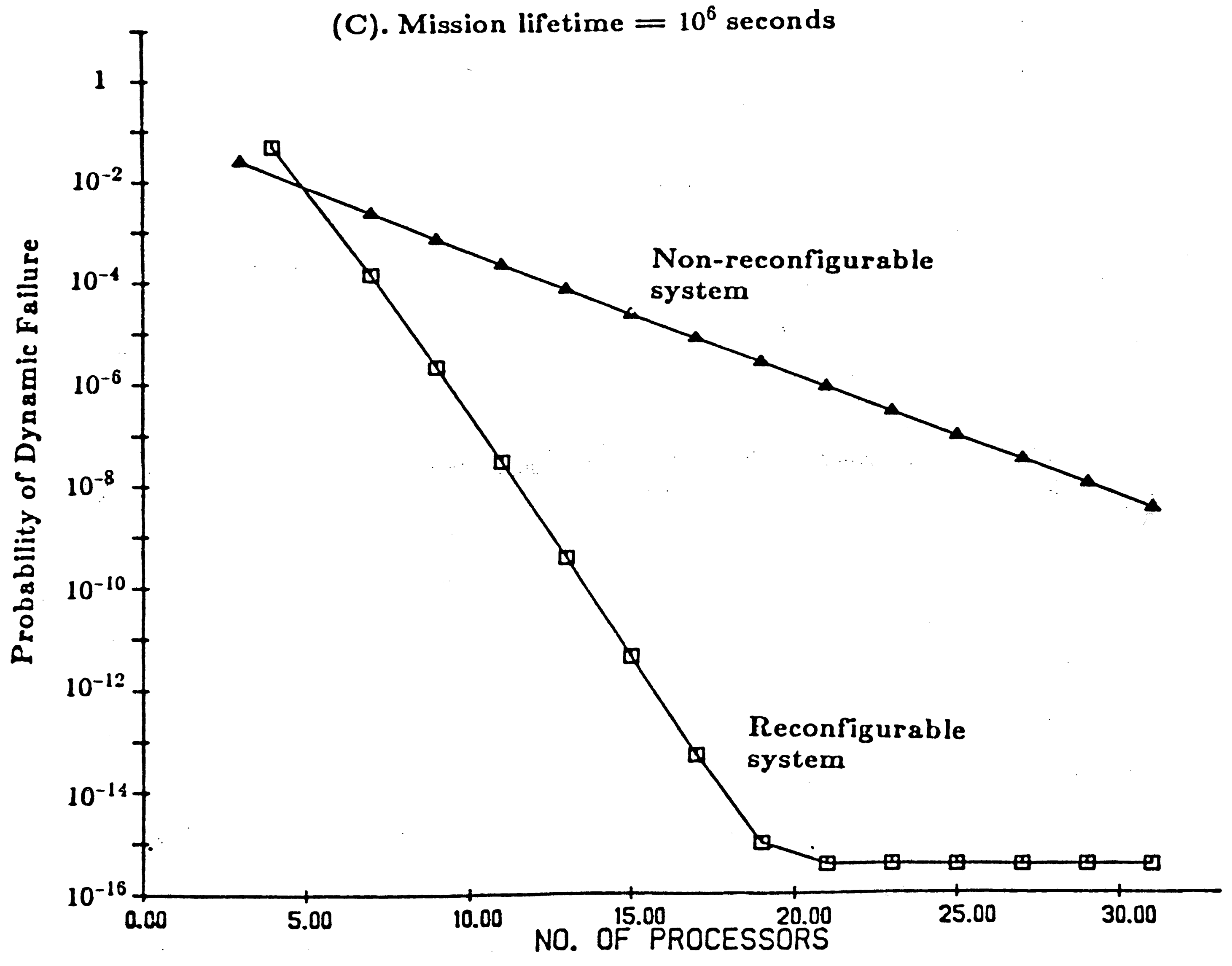

(C). Mission lifetime = 10^6 seconds
Probability of Dynamic Failure
1
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
0.00
5.00
10.00
15.00
20.00
25.00
30.00
NO. OF PROCESSORS
Non-reconfigurable system
Reconfigurable system

Figure 27(c)

Reconfigurable System

We assume here that the interactive consistency algorithm will only be invoked when a vote detects processor failure. The problem of replicating simplex data to amongst multiple processors is not treated here: it is assumed that the slight variations in analogue sensor data obtained without the Byzantine algorithm are acceptable. The purpose of the interactive consistency algorithm here is to obtain agreement on diagnostic tests. The assumptions are that the tests have 100% coverage, and for convenience, that the diagnostics take 3 ms. Clearly, the diagnostic period is insensitive to the value of m. It is not difficult to alter the analysis to allow for a relaxation of these assumptions. Doing so may alter the numerical values presented, but will not change the qualitative nature of these results.

The execution time is bounded below by $2.22N^{m-1}+9.6$ milli-seconds. Since the task execution time is 1.6 milli-seconds, an unreplenishable reserve of 50–1.6=48.4 milli-seconds of time is available. If the overhead is smaller than this, the probability of dynamic failure is equal to the probability of hardware failure: if not, it is equal to unity.

As may be seen from a simple calculation, for clusters with $m>2$, the overhead exceeds the reserve of time, so that the maximum allowed size of the cluster is $N=7$, $m=2$. For this reason, if we start with more than 7 processors, the additional processors will have to be on stand-by for inclusion upon a failure within the cluster. Failure occurs if either during a single execution more than m failures occur in the cluster, or the processor pool is exhausted, i.e. if there is an insufficient number of processors left to make up a cluster.

The reconfiguration policy is simple. The system begins operation with either a 7-MR or a 4-MR cluster (depending on the value of N). As processors fail, they are replaced if spares are available. If the stock of spares is exhausted, further failures are handled by the 7-cluster reconfiguring into a 4-MR cluster. If $N<7$, only a single failure can be tolerated. It is thus a combination of hybrid and adaptive voting.

Numerical results for the probability of failure of reconfigurable systems are plotted in Figure 27 for a ready comparison with their non-reconfigurable counterparts.

It is apparent from Figure 27 that while increasing the number of available processors in a non-reconfigurable system reduces its probability of failure, there is a lower bound to the probability of failure for reconfigurable systems. This bound is caused by the fact that the cluster size is limited to 7, since the overhead exceeds 100% for larger clusters. There is therefore a point after which the probability of more than *m* processor failures over a single execution (aggregated over the mission lifetime) becomes the dominant component in the probability of failure. As one might expect, the reconfigurable system performs better than the non-reconfigurable system when the mission lifetime is larger. This has been at the horrible price, in this case, of a 50.3% overhead.

Clearly, the results in Figure 27 are problem-specific, indeed, they are critically dependent on the length of the inter-dispatch interval, the unreplenishable reserve of time that is available, and the time taken to execute the Byzantine algorithm. Also, while the reconfigurable system may appear to be the better performer in Figure 27, this is largely due to the large reserve of time available. Suppose that the task, instead of taking a maximum of 1.6 milli-seconds to perform, took 35 milli-seconds. Then, the reserve of time is 50–35=15 milli-seconds, and the largest reconfigurable cluster that could fit in this reserve is N=4, m=1. In Figure 28, we display results for such a task. Naturally, the reconfigurable system comes out much more poorly here.

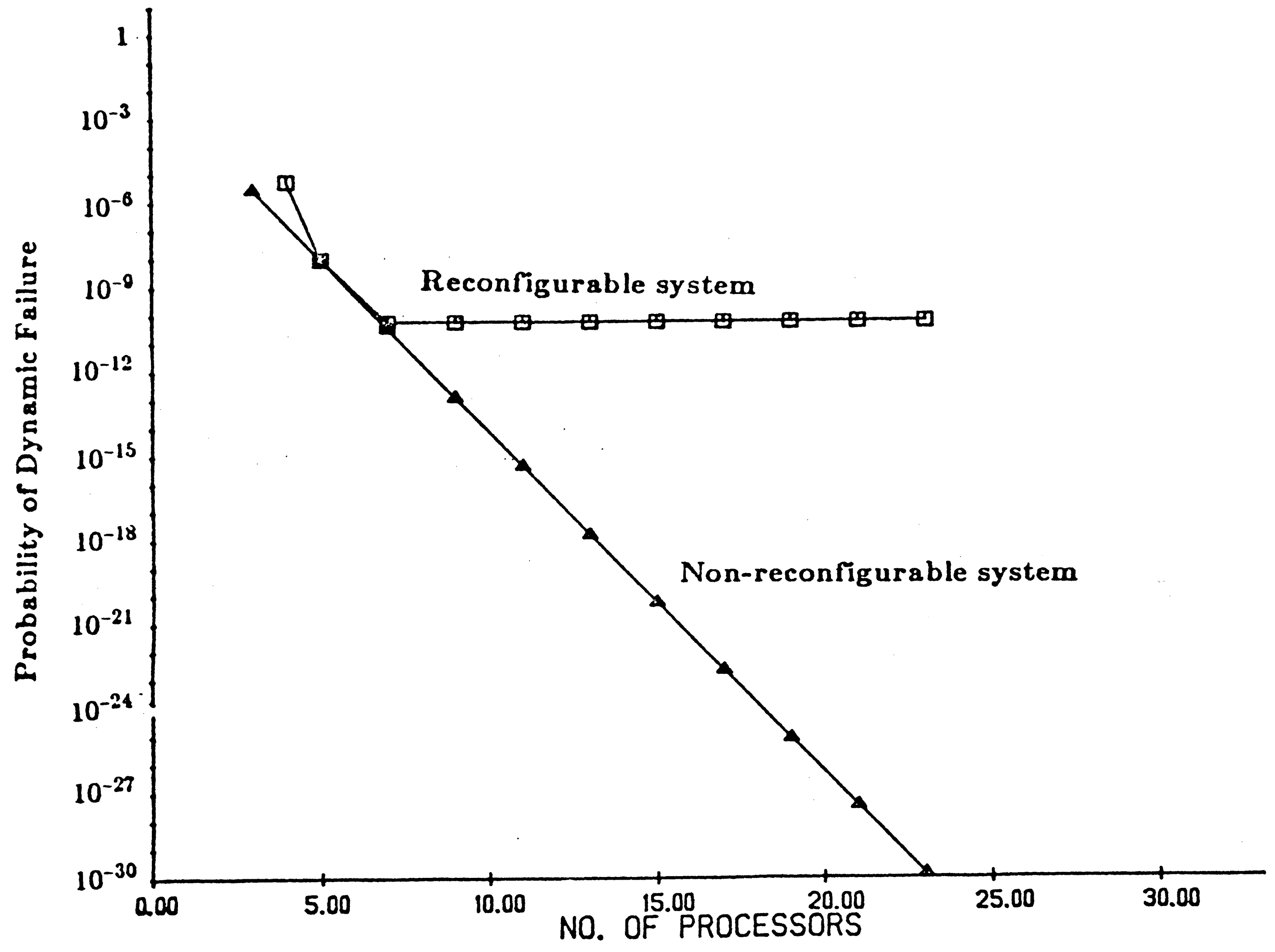


Figure 28. **Comparison of Reconfigurable and Non-Reconfigurable Systems**

CHAPTER 7

SCHEDULING TASKS WITH A QUICK RECOVERY FROM FAILURE

Part of the procedure for recovery from a processor failure must consist of ensuring that hard deadlines of critical tasks continue to be met to a very high probability. This is usually done by replicating tasks. One may choose (i) to have all replicates, or *clones*, of tasks always running in parallel, or (ii) to have some of them (called "ghosts" in the sequel) initially inactive, and then to activate them in case a processor carrying an active (or "primary") colleague fails. When the mission lifetime is long, or the failure probability requirements are stringent, the first approach causes a considerable incremental processor loading (due to the large number of replicates that are then required), and a worsening of response times. Even if such a worsening does not increase the probability of missing a hard deadline, it will tend to degrade the quality of control provided to the real-time system: It might therefore be profitable to consider the second approach, which has the advantage of requiring fewer replicates to run in parallel.

Implementing this second approach involves optimally (more realistically, quasi-optimally) inserting the ghosts into the schedule. The ghosts cannot be regarded as primary clones, or the schedule reduces to that in the first approach. In many cases, it is unwise to plan to schedule them on-line when a processor failure is detected: that might not be possible under the imposed timing constraints. In such instances, one must embed a set of contingency schedules into the original or primary schedule and invoke it whenever a processor failure is detected. In this chapter, we first show how to find the contingency schedules. Then, a dynamic programming approach is used to construct out of these contingency schedules, the primary schedule, which is defined as the schedule followed if there is no processor failure. The contingency and the primary schedules consists only of primary clones. A ghost schedule is also derived, and it is then a simple matter to amalgamate the ghost and contingency schedules on-line when one or more ghosts have to be activated due to processor failure.

While a more detailed problem statement is provided in the next section, it might be useful to preview here what we have done in this chapter. In general, the job of scheduling tasks on processors consists of optimizing some objective function, while ensuring that given constraints are met. In this case, the constraints are on the hard deadlines being satisfied, and the objective function is the expected sum of the cost functions of the finishing times of the executed tasks. We assume that an algorithm exists that accepts as input a list of tasks with their release times, run times, deadlines, and cost functions, and produces as output a schedule that minimizes (more realistically, quasi-minimizes) the above objective function. Such a schedule will not be fault-tolerant. We show how to use such an algorithm to create a fault-tolerant schedule, consisting of a quasi-optimal primary schedule which is executed as long as no processors fail, and a set of contingency schedules that are invoked when one or more processors fail, making it necessary to activate the ghosts of the primary tasks that were originally supposed to have run on the now-failed processors. The object is to keep the number of replicates that must run in parallel at the desired level. The primary and contingency schedules between them ensure that reliability specifications are met, i.e. that even in the face of a given number of processor failures, the probability of missing a hard deadline is kept acceptably low.

Papers in the literature [38,39] that have addressed the problem of fault-tolerant scheduling for real-time controllers have taken a different approach to handling the effects of processor failure. The scheduling algorithm used in [38] is a best-fit procedure that is claimed to be quick enough to compute a new post-failure schedule in real time. There is, however, no guarantee that a certain given number of processor failures can be sustained: indeed, the method assumes that as failures occur, tasks that cannot be scheduled under this approach are simply shed. The algorithm in [39] has the same features, being an improved version of the one in [38].

Section A contains a list of assumptions, and a more detailed statement of the problem. Section B contains the main result of the chapter, Section C a numerical example, and Section D a brief indication of this algorithm's practical usefulness.

A. Assumptions and Problem Statement

The computer is assumed to be a multiprocessor following the definition of Enslow [40], that is to say, the processors of this computer each have their own private memories. The schedule to be derived is *locally-preemptive,* i.e., tasks resident on the same processor are allowed to preempt each other, 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. It also removes the need to allow for queueing delays as part of the schedule.

By a *feasible schedule,* we mean a schedule in which all hard deadlines are met.

As in [38] and [39], the tasks are treated as having no precedence constraints. This does not rule out tasks forming a pipe, as in Figure 29, if the tasks can effectively be decoupled by bounding above the deadline of a parent task in the task-precedence tree by the minimum of the release time of all its child tasks. This is not always possible, since for example, the release time of a child task may solely depend on the moment that the parent task finishes executing. In such cases, the approach presented here (as also those in [38] and [39]) breaks down, and a more powerful algorithm must be sought.

Each version has a given release time, a deadline, and represents a certain volume of computation demand. Other assumptions are as follows.

A1. The mission lifetime, 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 failure are nowadays in the 1,000- to 10,000- hour range. Given a ten-processor system, with processor MTBF's of 10,000 hours, the mean time between successive processor failures varies from about 1,000 to 10,000 hours, depending on how many processors are left functional, and assuming Poisson fault occurrences. We shall later consider the consequences of relaxing A1. A2 is unrealistic: the real-time environment is

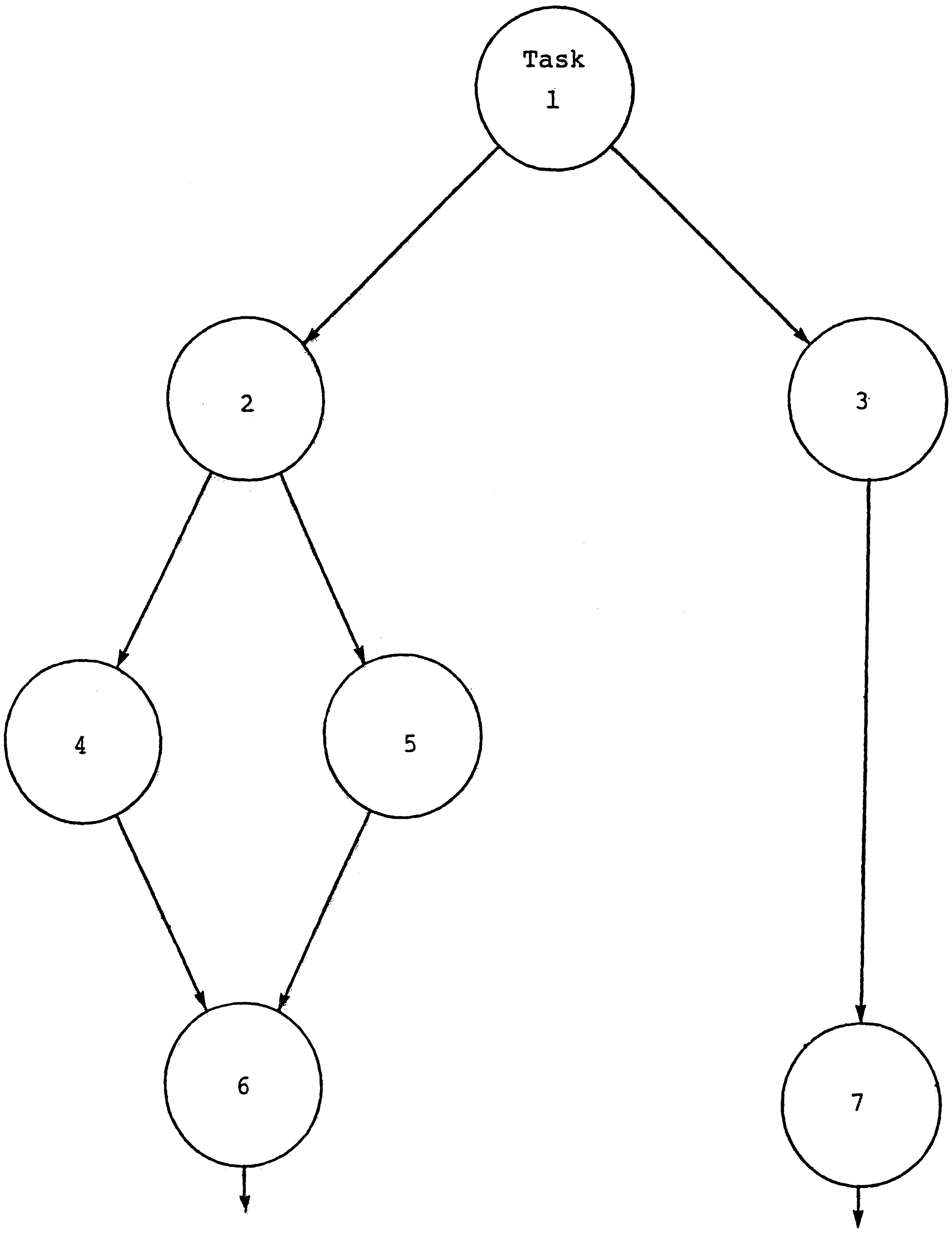


Figure 29. A Typical Pipe

stochastic, and this influences the task characteristics. However, in practice, worst-case numbers may be used to ensure that performance bounds can be met, thereby making A2 acceptable. Without A2, the problem would turn into one of stochastic scheduling which is known to be extremely difficult, if not impossible. Without sufficient processors, we clearly cannot proceed with a scheduling operation, which justifies A3. It is worth pointing out here that determining the minimum number of processors required is an NP-complete problem.

As we saw in Chapter 3, a completion time of C_j of a particular version j can be linked to a *cost,* or overhead, of $f_j(C_j)$, where f_j is the cost function. Each schedule defines the task completion times, and therefore has a cost associated with it, computed by adding together all the contributions from individual tasks. This is the objective function that must be minimized for determining an optimal schedule, subject to meeting the specified probability of dynamic failure.

We now consider rescheduling upon processor failure. Rescheduling must not be expected to rescue a given version of a task: that must be done, if at all, by other means. It is at the *end* of a version execution that the system responds to a failure by rescheduling the affected tasks.

To understand why, we must consider the fault-detection mechanism. The traditional approach to fault tolerance using massive redundancy is to partition the multiprocessor logically into clusters of processors with respect to the individual tasks, and have the tasks run in parallel on each processor in the cluster. At the end of their execution, voting is carried out on the results. If unanimity is achieved, then it is assumed that no failure has occurred, the event that all members of the cluster arrive at the same wrong result being considered practically impossible. If the majority of processors agrees on a result, that is considered the "true" result, and all processors in the cluster that do not agree with the majority result (or put out no result at all up to a given timeout moment) are regarded as suspect. Of course, if there is no majority result, the cluster as such fails, hard deadlines are missed, and catastrophic or otherwise unacceptable failure is the result.

When unanimity is not achieved, fault-location algorithms must be invoked to locate the faulty processors. One way to do this is to run diagnostic tests on all the processors and then to agree on the results by using, say, the interactive consistency algorithm In any case, the detection and location of a faulty processor occurs shortly after a given execution on that processor is complete. There is nothing that the rescheduling algorithm can do if the cluster as a whole fails, i.e., if the voter can find no majority result. The probability of that happening must be kept acceptably low by using sufficient active redundancy. Also, since everything (except failures) in the system is deterministic by assumption A2, the voting times are fixed, and so faults can only be located at certain given moments.

When a processor failure is located, the primary clones allocated to the failed processor and not already started, cannot be executed. One ghost of each of these clones must therefore be activated. Clearly, the system will know of the need to activate a certain ghost only when it knows of the processor failure that gave rise to this need. We define the moment by which the system knows of the need to activate a ghost as that ghost's *notification time*. The notification time for ghost j is denoted by ν_j, with the ghosts being numbered 1,2, ... For convenience, we define $\nu_0=0$. We assume that ν_j is less than or equal to ghost j's release time. *Notification interval j on processor i* is defined as the interval between the j-th and

$(j+1)$-th notification times of ghosts on that processor.

Failure can occur due to one of two reasons:

F1. Failure of one or more redundant clusters during one execution cycle so that at least one task misses a hard deadline sometime during the mission. As we said above, there is nothing that the rescheduling algorithm can be expected to do about this.

F2. F1 does not happen, but so many failures occur over the entire mission lifetime that the system is unable to ensure that each version begins executing on a full-strength cluster, i.e. one might have to make do with fewer than the prescribed number of clones running in parallel (so that the probability of F1 occurring is raised above that considered tolerable).

F2 is a simplification born of pessimism: not every weakened cluster is guaranteed to fail. From assumption A2, F1 and F2 can be used to determine the maximum number, N_{sust}, of processor failures that should be sustained. It is easy to see

$$p_{dyn} \leq Prob\{F1\} + Prob\{F2\} \tag{55}$$

where the $\leq$ relationship arises from the fact that F2 is pessimistic. N_{sust} can then easily computed from (1), and the given reliability specification, p_{dyn}^{spec}. This is illustrated in the following example.

Example: Let there be two clusters of three processors each, together with some standbys. Assume that processor failures occur independently as a Poisson process with rate μ. Let the period between fault-detection tests be constant at α, and the mission lifetime be L. Define $\beta \equiv \lceil L/\alpha \rceil$. Clearly,

$$Prob\{F1\} \approx 2\beta p^2(\alpha)\,(3-2p(\alpha)) \tag{56}$$

where $p(r)=1-e^{-\mu r}$. The approximation in (1) holds so long as the probability of $F1$ is much less than one. In such a case, the probability of F2 is given by:

$$Prob\{F2\} = 1 - \sum_{i=0}^{M} \binom{N}{i} p^i(L)[1-p(L)]^{N-i} \tag{57}$$

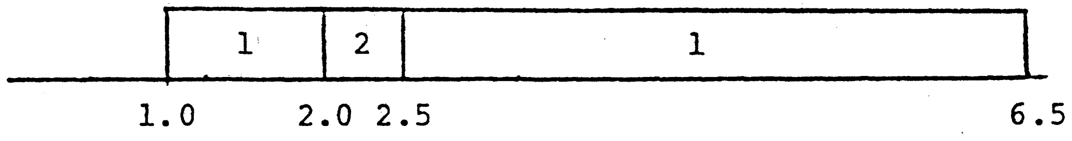
when M is the number of failures that can be sustained. Then, N_{sust} is the greatest M in (3) for which

$$Prob\{F2\} \geq p_{dyn}^{spec} - Prob\{F1\} \tag{58}$$

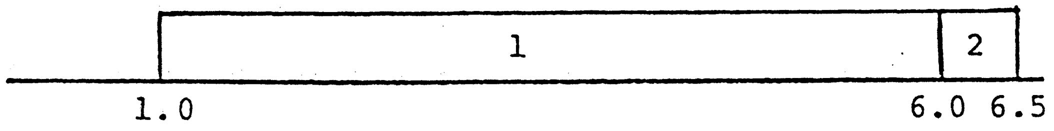
When $p_{dyn}^{spec}=10^{-9}$, $MTBF$=10,000 hours, the mission lifetime, L=10 hours, and fault-location checks are carried out every 0.1 second, Prob{F1} is about 1.67×10^{-11}, (totally negligible when compared to $p_{dyn}^{spec}=10^{-9}$), and $N_{sust}=2$.

As noted before, in order to sustain the allowed-for number of processor failures, ghost clones of the various critical tasks are scheduled. Now, ghost clones are not activated if there is no processor failure, and we have assumed (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 is allocated primary clones 1 and 2, and ghost clone 3. Let their release times be $rel_1=0$, $rel_2=1$, and $rel_3=3$, their deadlines be $t_{d1}=10$, $t_{d2}=12$, and $t_{d3}=7$, and their cost functions be $f_1(t)=t-1$, $f_2(t)=(t-2)^2$, and $f_3(t)=0$ 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 $r_3=4$. Without the ghost, the optimal primary schedule is clearly as shown in Figure 30a. 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 Figure 30b, which is much more expensive than the one in Figure 30a.

Since, in the overwhelming majority of cases, the primary schedules are the only ones that will be executed, their efficiency 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 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 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 Chapter 3. Given that the set of real numbers is everywhere dense, this inflicts no practical inaccuracy on our



(a) Optimal Primary Schedule Without Influence of Ghost 3



(b) Optimal Primary Schedule Showing Influence of Ghost 3

Figure 30. Effect of Ghosts on Primary Schedules: An Example

calculations.

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

We can now state our problem. Assume we are given an algorithm P that finds either an optimum or quasi-optimum feasible schedule by minimizing the costs associated with individual schedules when all clones are assumed to be primary.[1] Owing to the NP-completeness of the scheduling problem, P is most likely to be a heuristic procedure. A good example is a best-fit procedure, along the lines of the algorithm in [38] (although some changes might be needed to accommodate the different optimization criterion here). As in [38], we make the assumption that we can split P into two sub-algorithms: P_1 which 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. Our task is to develop an algorithm Q that can be used in conjunction with P_1 and P_2 to obtain an optimum schedule when enough ghosts are incorporated into the task set to sustain up to N_{sust} processor failures, and also to obtain contingency schedules that can be invoked when the system is notified that a given ghost is to be activated.

B. Main Result

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

[1] To avoid tedium in the sequel, by "optimal" we shall mean either optimal or quasi-optimal. When we wish to exclude quasi-optimality, we shall use the term "truly optimal".

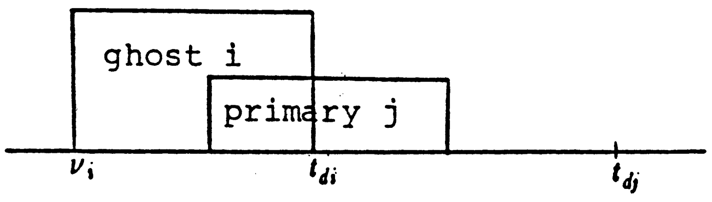
C1. There are N_{sust} ghost clones for each version.

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.

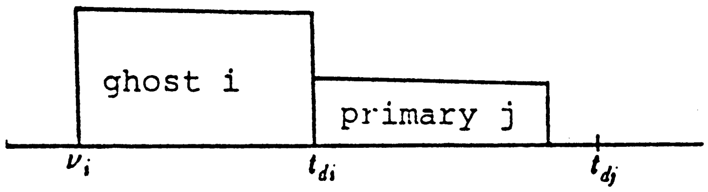
C3. Two clones of the same version, whether primary or ghostly, cannot be scheduled on the same processor.

An illustration of C2 is provided in Figure 31. 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 Figure 31a, 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. Figure 31b shows an overlap that does not meet condition C2: in the event that the ghost is activated, then either ghost i or primary j will miss its deadline.

It is easy to see that C1, C2, and C3 are necessary and sufficient conditions to ensure that reliability specifications are achieved.

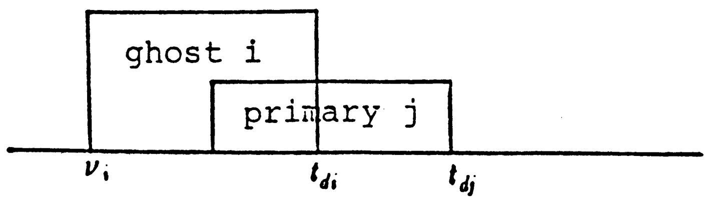


Overlapped schedule



Ghost i Activated

(a) Permissible Overlap



(b) Impermissible Overlap

Figure 31. Illustrating Condition C2

We can now turn to obtaining algorithm Q. We begin by stating some additional definitions and notation pertinent to algorithm 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 clones released prior to t are less than t and (ii) no clone is released at t. Clearly, a hole depends only on the *allocation* of clones and not on their specific positioning in a schedule. In other words, all feasible schedules for a given allocation 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 anything allocated are holes. By *primary holes,* we mean the holes that would exist in a processor's schedule if the ghosts allocated to that processor did not exist.

Algorithm P_2 has, as input, the deadlines of the clones, and the remaining run times. *Running* $P_2(A)$ means that P_2 is invoked with the set of clones A allocated to the processor. By *initializing* P_2 at some time r we mean that P_2 generates a schedule starting at r, using the inputs mentioned above. We denote the set of ghosts that have been allocated by P_1 to processor i by θ_i. By *ghost*$_i(j)$ we mean the j-th ghost (in order of notification time) allocated to processor i.

Algorithm Q adjusts or *transforms* the hard deadlines at step 3b. A deadline transformation is said to be *feasible* if, when it is possible to generate a feasible schedule of clones with pre-transformed deadlines, it is also possible to do so after the clones' deadlines have been transformed. Finally, we define P_2^* to be a modification of 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, 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 P_2 is when it would not alter the finishing time of a primary clone. See Figure 32. In that event, all that is required to do is to "nest" the ghost within the primary. This means that, for any given time t, the non-primary-hole space occupied by the ghosts prior to t is the minimum.

Algorithm 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 P_1 do steps 1 to 6 for every processor i:

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

Step 2. Run $P_2^*(\pi_i \bigcup \theta_i)$. Return control to 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 P_2 at $\nu_{ghost(j)}$.

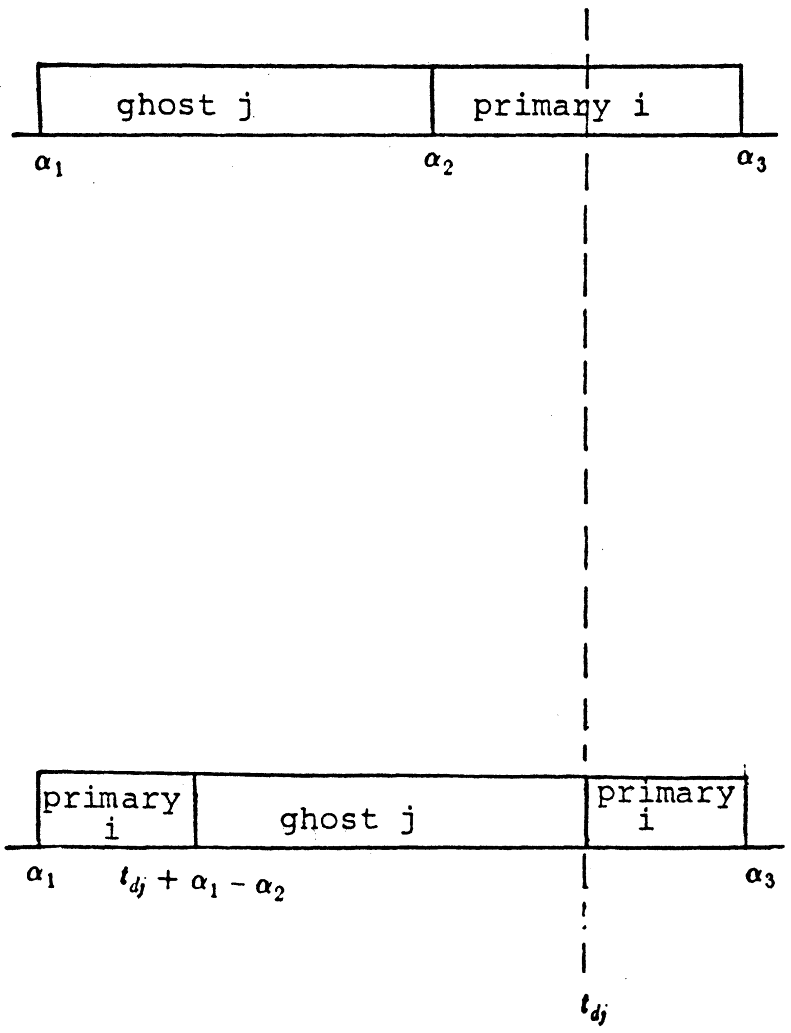


Figure 32. Nesting a Ghost Inside a Primary

(b) Transform the hard deadlines of the primary clones $k \in \pi_i$ to be $t_{dk}^{(0)} - util_{\theta_i}^{G}(0, t_{dk}^{(0)})$, where $t_{dk}^{(0)}$ is the original hard deadline of clone k and $util_{\theta_i}^{G}(a,b)$ is the non-primary-hole space occupied by the ghosts in θ_i, according to the ghost schedule G in the interval [a,b].

(c) Run $P_2(\pi_i)$ with these new deadlines, and let S_{ij} be the schedule obtained.

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

(e) Set $\theta_i := \theta_i - ghost(j)$

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

Step 4. If all primary clones $k \in \pi_i$ have not been completed by $\nu_{ghost(j)}$, set $t_{dk} := t_{dk}^{(0)}$, initialize P_2 at $\nu_{ghost(j)}$, and run $P_2(\pi_i)$, recording the output in S_{ij}.

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

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. Q puts out a matrix of contingency schedules S_{ij}, with the optimal primary schedules (optimality is proved 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 $\nu_{ghost_i(j)}$ if (a) $ghost_i(j)$ is to be activated, and (b) all ghosts l with $\nu_l<\nu_j$ and allocated to processor i will not be activated.

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–C3 for the ghosts in θ_i. This is schedule S_{i0}. The portion of S_{i0} in the interval $[\nu_0,\nu_{ghost_i(1)}]$ 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 $[\nu_0,\nu_{ghost_i(1)}]$. Then, at $\nu_{ghost_i(1)}$, $ghost_i(1)$ is discarded from θ_i, and the above procedure is repeated to obtain S_{i1}. The portion of S_{i1} in the interval $[\nu_{ghost_i(1)},\nu_{ghost_i(2)}]$ is the corresponding portion of S_i^*. Proceeding in this manner, Q pieces together the optimum primary schedule, which is the one that will be run if there is no ghost invocation. If $ghost_i(n)$ 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 $\nu_{ghost_i(n)}$) 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.

Figure 33 provides an illustration of this. In this example, there is no hole in the schedule prior to t=40. 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 feasible schedule for a processor i, then all deadline transformations in Q pertinent to processor i for that allocation are feasible.

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 $util_{\theta_i}^{G}$ (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. **Q.E.D.**

Theorem 2: If P is truly optimal, S_i^* is truly optimal.

Proof: Let schedule S_i^* found by Q not be truly 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

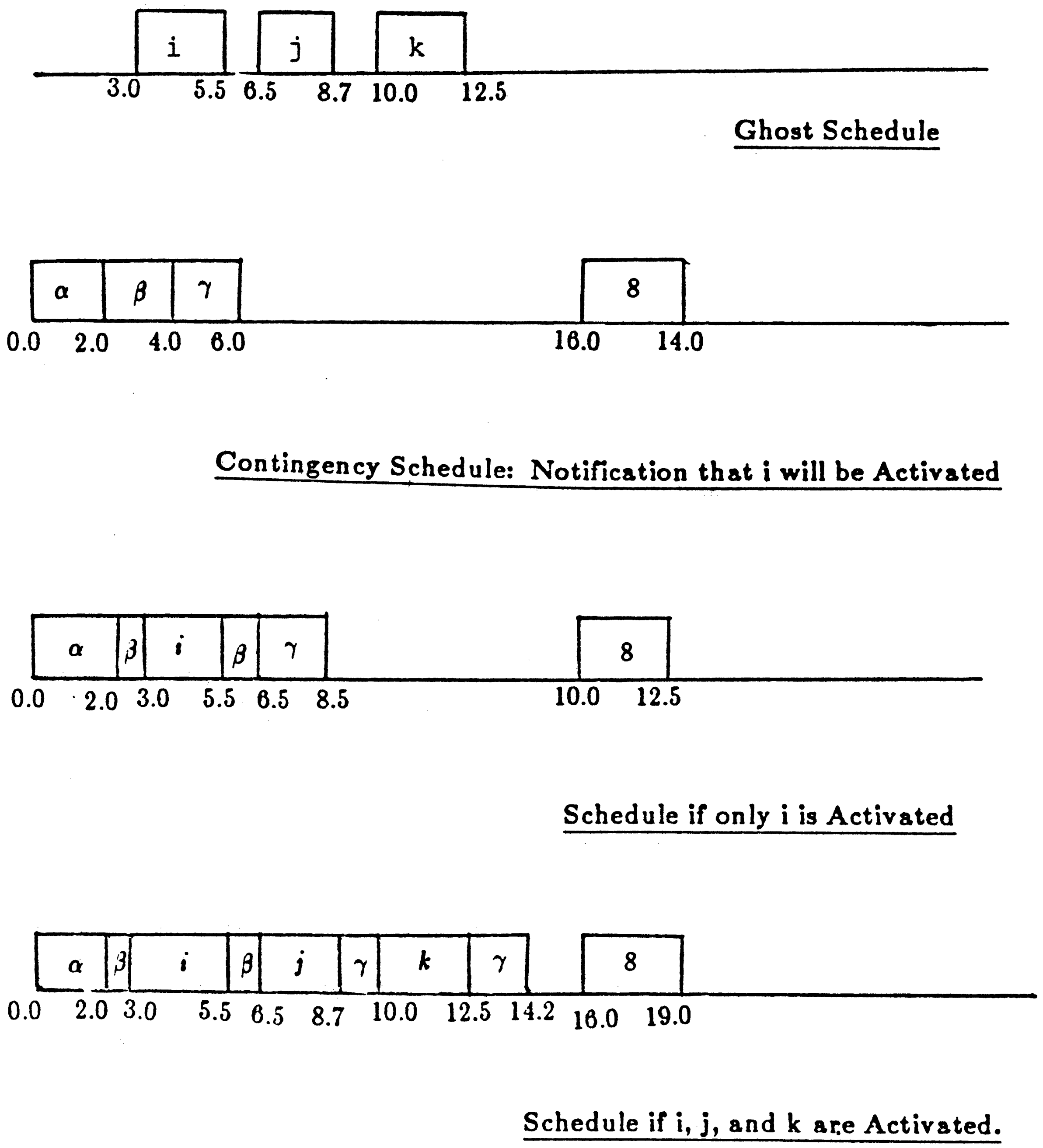


Figure 33. Amalgamating Ghost and Contingency Schedules

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. $\delta\in[\nu_{ghost_i(h)},\nu_{ghost_i(h+1)})$. Denote by $util_i(a,b)$ the amount of non-hole space occupied on processor i by ghosts $B=\{c: \nu_c>\nu_{ghost_i(h)}$ and c *is allocated to* $i\}$ in the interval [a,b]. Clearly, $\epsilon\in(t_{dj}^{(0)}-util_i(0,t_{dj}^{(0)}),\ t_{dj}^{(0)})$, where $t_{dj}^{(0)}$ denotes the original (i.e. pretransformed) deadline of clone j. If $util_i(0,\epsilon)=util_i(0,t_{dj}^{(0)})$, 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, $util_i(0,\epsilon)<util_i(0,t_{dj}^{(0)})$, i.e. $util_i(\epsilon,t_{dj}^{(0)})>0$. If all ghosts in B scheduled prior to ϵ are activated, then clone j's finishing time will be increased by at least $util_i(0,\epsilon)$ + $util_i(\epsilon,util_i(0,\epsilon)+\epsilon)$. Now, $util_i(\epsilon,util_i(0,\epsilon)+\epsilon)=\sigma_1>0$ since otherwise $util_i(util_i(0,\epsilon)+\epsilon,t_{dj}^{(0)})$ > $t_{dj}^{(0)}-\epsilon-util_i(0,\epsilon)$, a contradiction. So, the finishing time for clone j will be incremented again, this time by σ_1. Similarly, we can show that $util_i(util_i(0,\epsilon)+\epsilon,util_i(0,\epsilon)+\epsilon+\sigma_1)=\sigma_2>0$, and so on. An infinite sequence $\{\sigma_i\}$ will be generated.

Now, if we can show that $\sum_{k=1}^{\infty}\sigma_k=util_i(0,t_{dj}^{(0)})-util_i(0,\epsilon)$, we have proved that S' is infeasible. Suppose, to the contrary, that $\sum_{k=1}^{\infty}\sigma_k=m<util_i(0,t_{dj}^{(0)})-util_j(0,\epsilon)$. Then, $util_i(\epsilon+m+util_i(0,\epsilon),t_{dj}^{(0)})=util_i(0,t_{dj}^{(0)})-util_i(0,\epsilon)-m$. Then, clearly, we have $util_i(\epsilon+m+util_i(0,\epsilon),t_{dj}^{(0)})\leq t_{dj}^{(0)}-\epsilon-m-util_i(0,\epsilon)$, i.e. $\epsilon\leq t_{dj}^{(0)}-util_i(0,t_{dj}^{(0)})$, a contradiction. So S' does not exist. **Q.E.D.**

If P is optimal, then Q will also be optimal, since $util_i(0,t_{dk})$ depends only on P_2^*, not on P_2.

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. For S_{in} to be optimal, we would have to piece it together as we did S_i^*. This would give rise to schedules $S_{in\ n+1}$, $S_{in\ n+2}$, etc. One would then have a contingency schedule for each possible combination of ghost invocation. (This is further explored in the example in the next section.) The set of such schedules would be very large. However, under assumption A1, failures are rare, and so the probability of any of the contingency schedules being invoked is very small. This means that the expected cost (in terms of the sum of cost functions of the clone finishing times) of the above set of schedules (i.e. S_i^* and the contingency schedules) is only slightly greater than the expected cost of the schedule that consisted of S_i^* as primary schedule, and the whole set of contingency schedules, pieced together as explained above. Indeed, it is because of assumption A1 that we have been able to use the cost of S_i^* as the optimal cost associated with the given allocation of clones to processor i.

This brings one 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 mission lifetime is very long. This is true, for example, of 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.

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 Q, since it depends on the given algorithm P.

C. Example

We illustrate here the formation of an optimal schedule for a given allocation (specified by algorithm P_1). We provide schedules for one of the processors only: the others are similarly derived.

Assume that P_1 has allocated primary clones of tasks 1,2,3, and 4, and ghost clones of tasks 5,6,7, and 8 to this processor. The cost function for primary clones of task i is given by:

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

The hard deadlines for the tasks are $t_{d1}=130$, $t_{d2}=110$, $t_{d3}=90$, $t_{d4}=60$, $t_{d5}=22$, $t_{d6}=39$, $t_{d7}=55$, and $t_{d8}=105$. The run time, rt, of all these tasks is 15 time units. The release times are: $rel_i=i$ for i=1,2,3,4, $rel_5=0$, $rel_6=24$, $rel_7=40$, and $rel_8=90$. 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 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 i: $V(i)=\min\{V(j)\}$ for all $j\in R$.

The holes are the intervals $[0,1)$ and $(130,\infty)$. The contingency schedules are shown in Figure 34. 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 P_1.

In Figure 35, we show the actual schedules followed upon ghost activation. The case treated is one in which contingency schedule S_{10} 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 non-hole space occupied by ghost 5. If ghosts 5 and 6 are activated, then primary 4 is interrupted twice: once by 5 and then by 6.

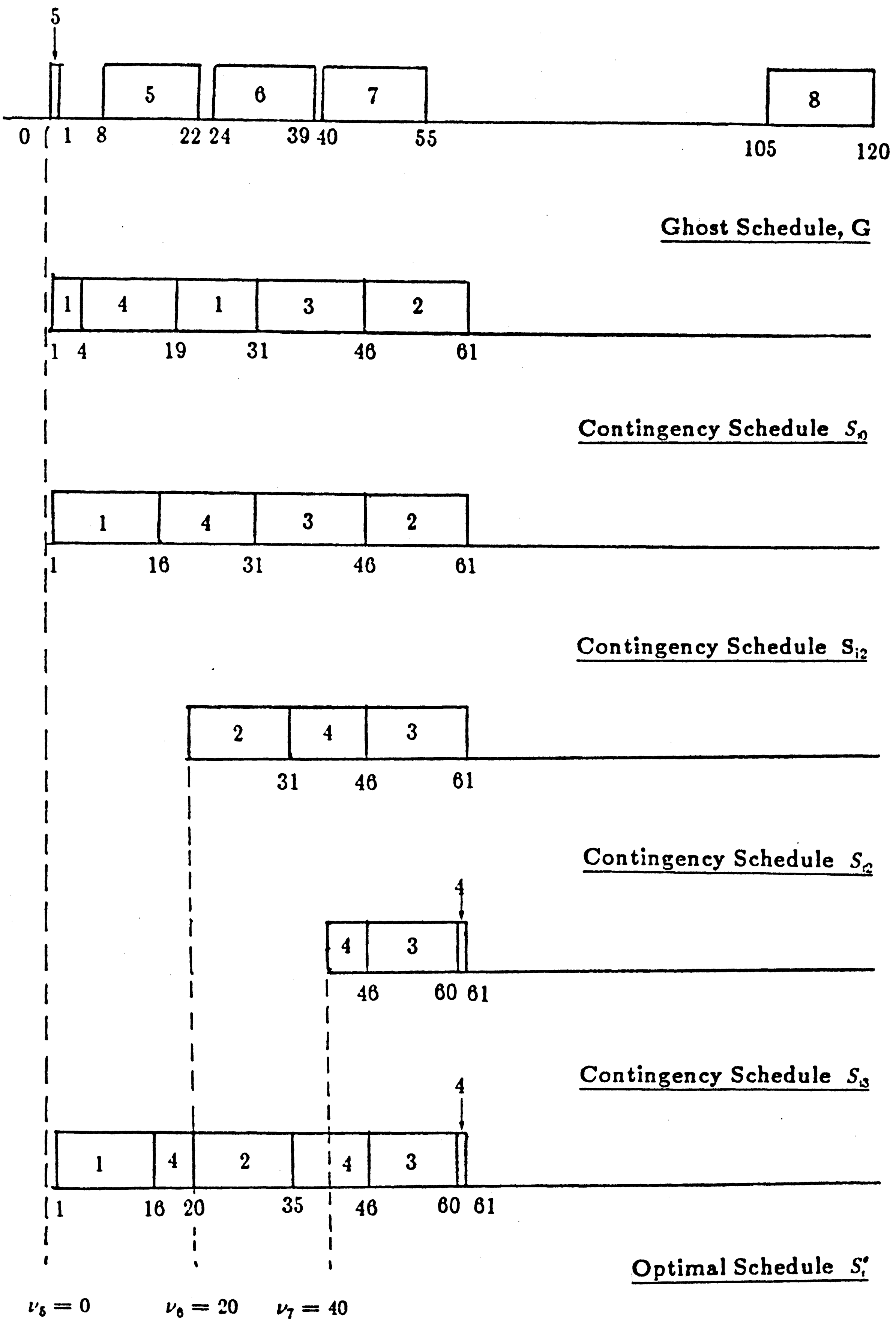
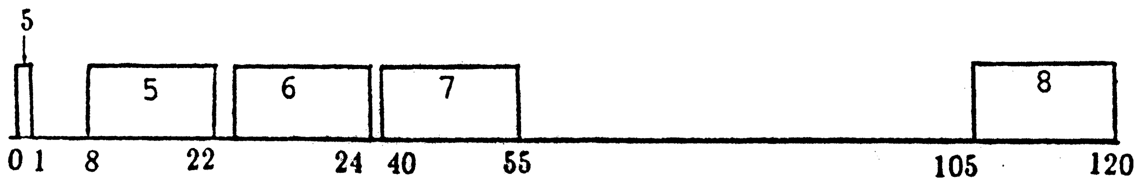
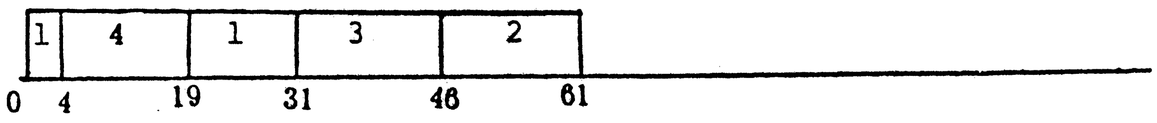


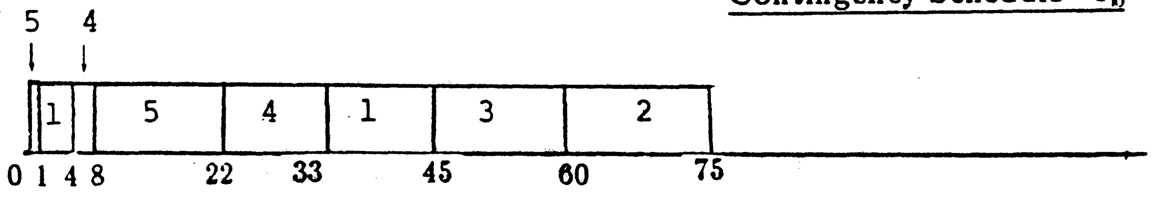
Figure 34. Obtaining S_i^*



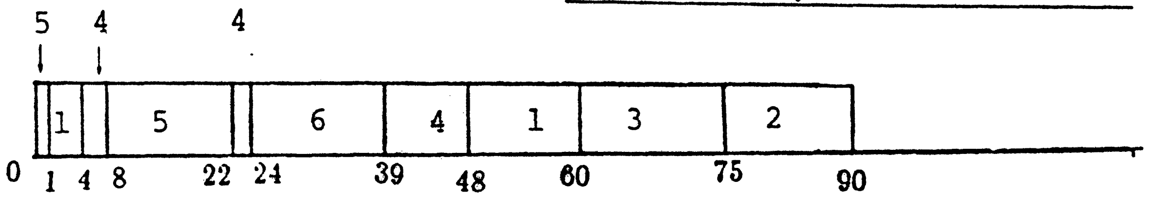
Ghost Schedule, G



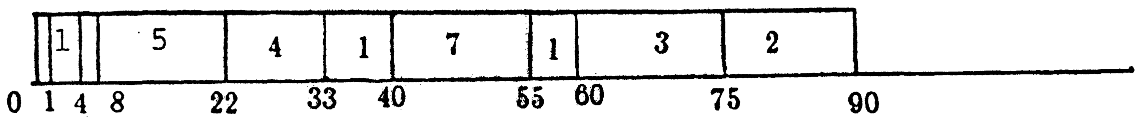
Contingency Schedule S_0



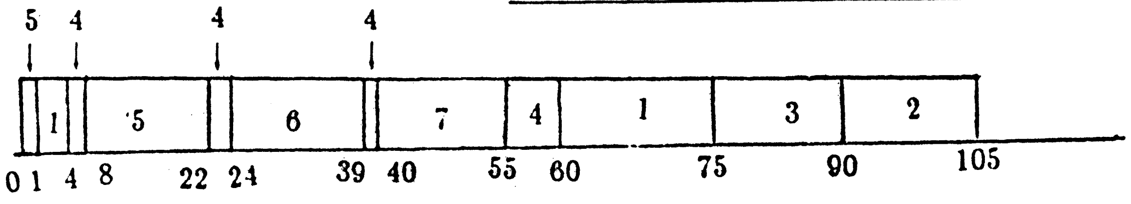
Schedule if Only Ghost 5 is Activated



Schedule if Ghosts 5 and 6 are Activated



Schedule if Ghosts 5 and 7 are Activated



Schedule if Ghosts 5, 6, and 7 are Activated

Figure 35. Schedules when Ghosts are Activated

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}.

We can now easily see that S_{i0} is not in itself optimal, in that it does not always lead to an optimal (optimal here is defined as "at least as good as the best schedule possible by using algorithm P_2^*") schedule if the ghosts are activated. In S_{i0}, primary 3 is set for execution before primary 2, although, if they exchanged positions, the resulting schedule would have lower cost. This is because we want to ensure that primary 3 meets its deadline in the event of all the contingencies that are notified after after $\nu_5=0$, specifically the contingency that ghosts 5, 6, and 7 are all activated. If we were willing to afford the extra memory space required to store more contingency schedules, we could piece together S_{i0} as we did S_i^*, and store, in addition to it, the second-order contingency schedules $S_{i0\,0}$, $S_{i0\,1}$, and $S_{i0\,2}$. These second-order contingencies are shown in Figure 36, together with the "new" S_{i0}. (Clearly, $S_{i0\,0}$ is just the same as the old S_{i0}.) Of course, these changes do not affect S_i^* as found by piecing together the original contingency schedules, since the new contingency schedules S_{im} for m=1,2,... , differ from their old counterparts, if at all,

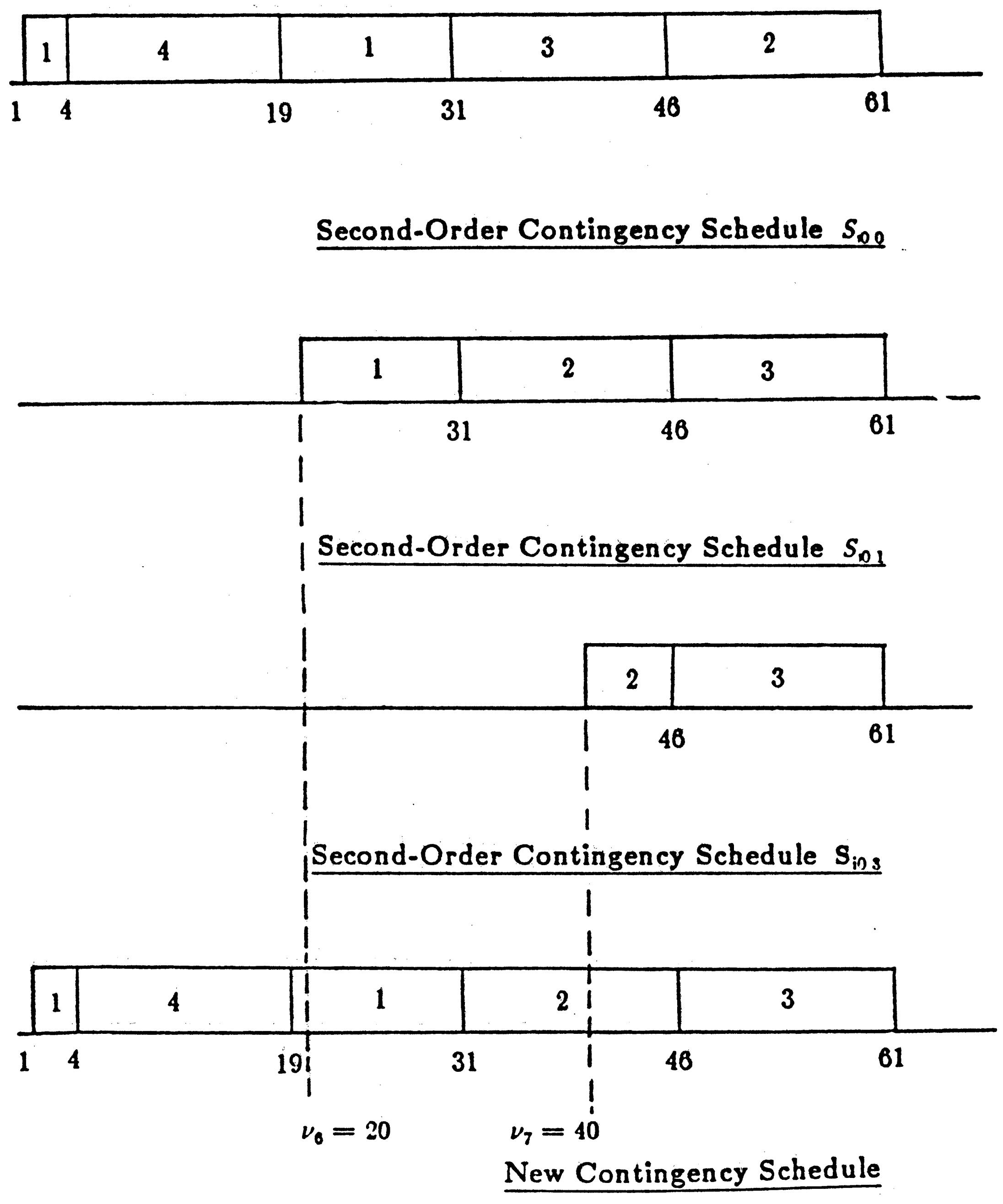

1
4
1
3
2
1
4
19
31
46
61
Second-Order Contingency Schedule $S_{i0\,0}$
1
2
3
31
46
61
Second-Order Contingency Schedule $S_{i0\,1}$
2
3
46
61
Second-Order Contingency Schedule $S_{i0\,3}$
1
4
1
2
3
1
4
19
31
46
61
$\nu_6 = 20$
$\nu_7 = 40$
New Contingency Schedule

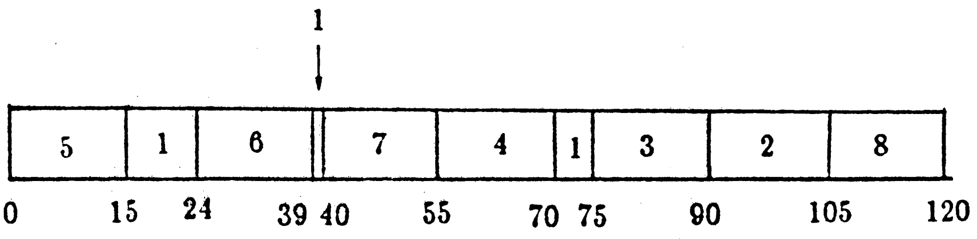
Figure 36. Piecing Together a Contingency Schedule

only for values of $t > \nu_{ghost(m+1)}$ (defining $\nu_{ghost(n)} = \infty$ if there are fewer than n ghosts allocated to processor i), and this piece of S_{im} is not included in S_i^*.

D. Discussion

This chapter has introduced a dynamic programming algorithm to generate fault-tolerant schedules. This algorithm is expected to be used in embedded computers where extremely high reliabilities are required. Examples are aircraft- and spacecraft-controllers, life-support systems, etc.

The algorithm is useful because it minimizes the number of primary clones. This makes the schedule S_i^* efficient for each processor i, while still providing fall-back schedules to be invoked in the (rare) instances of processor failure. 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 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 Figure 37, 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 Figure 35.



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

Figure 37. All Clones Treated as Primary

PART IV

DISCUSSION

CHAPTER 8

DISCUSSION

In this dissertation, we have introduced new performance measures for assessing more appropriately the performance of multiprocessors in the control of critical real-time processes. In addition, we have, among other things:

(i) Provided detailed examples of the derivation of these measures.

(ii) Studied problems of synchronizing real-time systems in the presence of malicious failure, showing the impracticality of software methods, and presenting means for the indefinite expansion of phase-locked clocks.

(iii) Presented a dynamic programming algorithm to generate fault-tolerant schedules.

The work presented here was impelled by the need for a more orderly approach to the design and evaluation of multiprocessors used in real-time applications. The proliferation in the literature of (often-times ill-considered) multiprocessor architectures is partially a consequence of the lack of an effective framework in which designers can function.

Coupled with this design activity is a growing awareness of the limitations of the theory associated with distributed processing. Queueing theory and scheduling theory are in their infancy. Even moderately complex queueuing and scheduling problems are proving to be -- for the present at least -- intractable. The need is therefore growing for carefully considered approximate models that combine the result of experience and intuition with the accuracy and power of the analytical method. Before such models can be confidently used, however, a great deal of groundwork has to be completed.

Central to this groundwork is the definition of appropriate characterizations of computer performance. As we have pointed out elsewhere [42], "by the simple act of choosing a particular criterion, the researcher imposes a bias on the results that follow. Performance measures by definition specify the commodity that is of importance ... They have a subtle influence on the way systems are viewed. They are languages through which we seek to convey system performance. We know from experience with natural languages that these affect not only the way in which ideas are expressed, but also the very ideas themselves. Performance measures are no exception to this. They manipulate our view of system behavior, contorting it to fit a pre-conceived mould. For this reason, the choice of a performance measure determines the practical usefulness of the results that are then derived."

We have therefore concentrated the first part of this effort on the derivation of appropriate performance measures. The rest of the dissertation may be considered a natural follow-up on these measures, and the problems treated there are among the most troublesome in fault-tolerant real-time system design.

Extensions of this work are not difficult to think of. These include the following.

(i) Determining ways to find the cost functions in real-life systems more efficiently than is currently possible. One way might be to derive approximate closed-form expressions for the functions.

(ii) Extending our performance measures to general-purpose systems. The chief difference between these measures and the traditional ones is that the former are task-oriented (i.e. user-oriented), and the latter are system-oriented. Part of this work would include further investigations of user-perceived computer performance. In particular, much work remains to be done on user-perceived response delay: it is not clear that mean response time is an accurate measure of this. Perhaps some measure based on the second moment will be more appropriate. Some preliminary work in this area has been done by Geist and Trivedi [41].

(iii) Studying the optimal placement of checkpoints. We have argued elsewhere [42] that the currently-used measures of mean response time and availability are inappropriate, and proposed new measures that are more sensitive to the benefits that accrue from checkpointing. This is interesting and relevant since it is likely -- as we pointed out above -- that mean response time is not a correct measure of perceived response delay.

(iv) Measuring the cost of status information in computer systems with distributed control. There is beginning to be increasing interest in such systems. When one employs distributed control, the perception of the state of the system can vary from controller to controller. Naturally, the closer each perception is to actuality, the better the control provided. However, distributing status information amongst the controllers is expensive since it takes time. If we used time-based performance measures such as these (they would obviously have to be modified if we were dealing with general-purpose computers), the cost of distributing such status information could then be determined. One can then transform the problem into one of taking decisions in the face of costly information: a problem widely studied in statistical decision theory.

The work that we have presented here was motivated by the need to study control computers as a distinct -- and important -- branch of computer engineering. Control computers have for too long been treated as an unimportant appendage of general-purpose computers. The problems specific to control computers are too difficult and important to be subjected to benign neglect.

APPENDIX

EXPRESSIONS FOR FINITE COST FUNCTION

Finite cost functions for Example 2 in Chapter 2 can be expressed by an **if-then-else** construct as follows:

```
if $x_{1i}x_{2i}>0$ then
    if $|x_{2i}|>k$ then
        $g(x_i,\xi) = \frac{1}{2}[t_1(x_i,k,\xi) + t_1(x_i,-k,\xi)]$
    else
        $g(x_i,\xi) = \frac{1}{2}[t_1(x_i, \mathrm{sgn}(x_{2i})k, \xi) + t_2(x_i, -\mathrm{sgn}(x_{2i})k, \xi)]$
else
    if $|x_{2i}|>k$ then
        $g(x_i,\xi) = \frac{1}{2}[t_2(x_i,k,\xi) + t_2(x_i,-k,\xi)]$
    else
        $g(x_i,\xi) = \frac{1}{2}[t_1(x_i, -\mathrm{sgn}(x_{2i})k, \xi) + t_2(x_i, \mathrm{sgn}(x_{2i})k, \xi)]$
end if;
```

where $a = H/m$, $y(x_{2i},k)=x_{2i}+k$, $x_i=(x_{1i},x_{2i})^T$,

$$\tau(x_i,k;\xi) = \frac{2a|x_{1i}| - y^2(x_{2i},k) - 2a|y(x_{2i},k)|\xi}{4ay(x_{2i},k)}$$

$$\tau(x_i,k,\xi) = \frac{2a\,|\,x_{1i}\,|\,-y^2(x_{2i},k)-2a\,|\,y(x_{2i},k)\,|\,\xi}{4ay(x_{2i},k)}$$

$$t_1(x_i,k,\xi) = \xi + \frac{|\,y(x_{2i},k)(1+\sqrt{2})\,|}{a}$$

$$t_2(x_i,k,\xi) = \begin{cases} \xi+\tau(x_i,k,\xi)+\dfrac{|\,x_{1i}\,|-|\,y(x_{2i},k)\,|\,(\xi+\tau(x_i,k,\xi))}{a}-\dfrac{\tau^2(x_i,k,\xi)}{2} & \text{if } \xi \leq \dfrac{2a\,|\,x_{1i}\,|\,-y^2(x_{2i},k)}{2a\,|\,y(x_{2i},k)\,|} \\ \xi+\dfrac{y^2(x_{2i},k)}{2a}+2\sqrt{\dfrac{y^2(x_{2i},k)}{a^2}-\dfrac{|\,x_{1i}\,|-|\,y(x_{2i},k)\,|\,\xi}{a}} & \text{otherwise} \end{cases}$$

REFERENCES

[1] A. L. Hopkins, et al., "FTMP -- A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE, Vol. 66, No. 10, pp. 1221-1239, October 1978.*

[2] 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, October 1978.*

[3] M. S. Fox, "An Organizational View of Distributed Systems," *IEEE Trans. Sys., Man, and Cyber.,* Vol. 11, No. 1, pp. 70-80, January 1981.

[4] J. Gertler and J. Sedlak, "Software for Process Control -- A Survey," *Automatica,* Vol. 2, No. 6, pp. 613-625, 1975.

[5] J. F. Meyer, D. G. Furchgott, and L. T. Wu, "Performability Evaluation of the SIFT Computer," *IEEE Trans. Comput.,* Vol. C-29, No. 6, pp. 501-509, June 1980.

[6] R. E. Barlow and F. Proschan, *Mathematical Theory of Reliability,* John Wiley, New York, 1962.

[7] M. D. Beaudry, "Performance-Related Reliability Measures for Computing Systems," *IEEE Trans. Comput.,* Vol. C-29, No. 6, pp. 501-509, June 1978.

[8] J. Losq, "Effects of Failure on Gracefully Degrading Systems," *Proc. FTCS-7,* pp. 29-34, March 1977

[9] R. Troy, "Dynamic Reconfiguration: An Algorithm and its Efficiency Evaluation," *Proc. FTCS-7,* pp. 44-49, March 1977.

[10] F. A. Gay and M. L. Ketelson, "Performance Evaluation of Gracefully Degrading Systems," *Proc. FTCS-9,* pp. 51-58, June 1979.

[11] J. M. De Souza, "A Unified Method for the Benefit Analysis of Fault-Tolerance," *Proc. FTCS-10,* pp. 187-192, October 1980.

[12] X. Castillo and D. P. Siewiorek, "A Performance Reliability Model for Computing Systems," *Proc. FTCS-10,* pp. 187-192, October 1980.

[13] T. C. K. Chou and J. A. Abraham, "Performance/Availability Model of Shared Resource Multiprocessors," *IEEE Trans. Reliability,* Vol. R-29, No. 1, pp. 70-76, April 1980.

[14] H. Mine and K. Hatayama, "Performance-Related Reliability Measures for Computing Systems," *Proc. FTCS-9,* pp. 59-62, June 1979.

[15] S. Osaki and T. Nishio, *Reliability Evaluation of Some Fault-Tolerant Computer Architectures,* Springer Verlag Lecture Notes in Computer Science, Berlin, 1980.

[16] R. Huslende, "A Combined Evaluation of Performance and Reliability for Degradable Systems," *Proc. ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems,* pp. 157-164, September 1981.

[17] J. F. Meyer, "On Evaluating the Performability of Degrading Computer Sustems," *IEEE Trans. Comput.,* Vol. C-29, No. 8, pp. 720-731, August 1980.

[18] J. F. Meyer, "Performability Modelling of Distributed Real-Time Systems," *Int'l Workshop Appl. Math. and Perf/Reliab. Model. Comp./Comm. Syst.,* pp. 345 - 356, September 1983.

[19] G. K. Manacher, "Production and Stabilization of Real-Time Task Schedules," *J. ACM,* Vol. 14, No. 3, July 1967, pp. 439-465.

[20] D. E. Kirk, *Optimal Control Theory,* Prentice Hall, Englewood Cliffs, NJ, 1970.

[21] F. J. Ellert and C. W. Merriam, "Synthesis of Feedback Controls Using Optimization Theory -- An Example," *IEEE Trans. Auto. Control,* Vol. AC-8, No. 4, April 1963, pp. 89-103.

[22] L. Essen, *The Measurement of Frequency and Time Interval,* H.M.S.O., London, 1973.

[23] T. B. Smith, "Fault-Tolerant Clocking System," *Proc. FTCS-11,* pp. 262-264, 1981.

[24] A. W. Holt and J. M. Myers, "An Approach to the Analysis of Clock Networks," R-1289, *C.S.Draper Laboratory Contract Report (Preliminary),* June 1978.

[25] D. Davies and J. F. Wakerly, "Synchronization and Matching in Redundant Systems," *IEEE Trans. Comput.,* Vol. C-27, No. 6, pp. 531-539, June 1978.

[26] J. Goldberg, *et al.,* "Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer," *NASA CR-172146,* June 1983.

[27] S. L. Hakimi and K. Nakajima, "Adaptive Diagnosis: A New Theory of t-Fault-Diagnosable Systems," *Twentieth Allerton Conf. Comm. Control,*

Comput., pp. 231-240, October 1982.

[28] D. Palumbo and R. W. Butler, "SIFT -- A Preliminary Evaluation," *Proc. Fifth Digital Avionics Symp.*, August 1983.

[29] M. Pease, *et al.*, "Reaching Agreement in the Presence of Faults," *J. ACM,* Vol. 27, No. 2, pp. 228-234, April 1980.

[30] L. Lamport, *et al.*, "The Byzantine Generals Problem," *ACM Trans. Prog. Lang. and Syst.*, Vol. 4, No. 3, pp. 382-401, July 1982.

[31] D. Dolev, "The Byzantine Generals Strike Again," *J. Algorithms,* Vol. 3, No. 1, pp. 14-30, January 1980.

[32] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design,* Digital Press, Bedford, MA, 1982.

[33] J. Losq, "A Highly Efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Trans. Comput.*, Vol. C-25, No. 6, pp. 569-578, June 1976.

[34] F. P. Mathur and A. Avizienis, "Reliability Analysis and Architecture of a Hybrid-Redundant Digital System," *Spring JCC, AFIPS Conf. Proc.*, Vol. 36, pp. 373-383, 1970.

[35] P. J. B. King and I. Mitrani, "The Effect of Breakdown on the Performance of Multiprocessor Systems," *Performance '81,* North-Holland, Amsterdam, 1981.

[36] N. A. Lynch, M. J. Fischer, and R. J. Fowler, "A Simple and Efficient Byzantine Generals Algorithm," *Proc. Reliab. Dist. Soft. and Database Syst.*, pp. 46-52, 1982.

[37] D. Gross and C. M. Harris, *Fundamentals of Queueing Theory,* John Wiley, New York, 1974.

[38] K. S. Trivedi and J. A. Bannister, "Task Allocation in Fault-Tolerant Distributed Systems," *Acta Informatica,* Vol. 20, Fasc 3, pp. 261-281, 1983.

[39] J. B. Mazzola and A. W. Neebe, "A Heuristic Procedure for Allocating Tasks in Fault-Tolerant Distributed Computer Systems," *Naval Research Logistics Quarterly,* Vol. 30, pp. 493-504, July 1983.

[40] P. H. Enslow, "What is a Distributed Data Processing System?" *Computer,* Vol. 11 No. 1, pp. 13-21, January 1978.

[41] R. M. Geist and K. S. Trivedi, "The Integration of User Perception in the Heterogeneous M/G/2 Queue," in A. K. Agrawala and S. K. Tripathi, eds. *Performance '83,* North-Holland, Amsterdam, 1983.

[42] C. M. Krishna, Y.-H. Lee, and K. G. Shin, "On Optimization Criteria for Checkpoint Placement," *Communications of the ACM,* to appear.