NASA Contractor Report 3010

A Fault-Tolerant Multiprocessor Architecture for Aircraft

Volume I

T. B. Smith, A. L. Hopkins, W. Taylor
R. A. Ausrotas, J. H. Lala,
L. D. Hanley, and J. H. Martin

CONTRACT NAS1-13782
JULY 1978

NASA

NASA Contractor Report 3010

A Fault-Tolerant Multiprocessor Architecture for Aircraft

Volume I

T. B. Smith, A. L. Hopkins, W. Taylor
R. A. Ausrotas, J. H. Lala,
L. D. Hanley, and J. H. Martin
The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts

Prepared for
Langley Research Center
under Contract NAS1-13782

National Aeronautics
and Space Administration

**Scientific and Technical
Information Office**

1978

TABLE OF CONTENTS FOR VOLUME I

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

This volume is a twelve-month interim report on a continuing study of a fault-tolerant multiprocessor architecture for aircraft, sponsored by the NASA Langley Research Center, Flight Instrumentation Division. The Technical Contract Monitor was Mr. Nicholas D. Murray. The work reported here was performed in the NASA/Army Department of the Draper Laboratory, with support from the Computer Science Division and the MIT Flight Transportation Laboratory. The project engineer was Dr. T. Basil Smith, III. The work period was May, 1975 to April, 1976.

A fault-tolerant multiprocessor architecture evolved at the Draper Laboratory under various sponsors from 1966 to the present time. This architecture, together with a comprehensive information system architecture, has important potential for future aircraft applications. This report is directed at the preliminary definition and assessment of a suitable multiprocessor architecture for such applications. The architecture is strongly driven by a requirement for extremely remote probability of system failure. Throughout this report, it is hypothesized that the computer's failure rate will be designed below 10^{-9} failures per hour in flights of up to ten hours duration, with a preferred goal of 10^{-10} failures per hour.

Summary of Conclusions

The following important conclusions have developed during and/or as a result of the work reported here.

The architectural principles of the multiprocessor have been demonstrated in an experimental configuration with a commercial aircraft simulation. The basic fault detection, diagnosis, and reconfiguration strategies are operational, and have undergone at least a superficial validation.

Future aircraft improvements in performance and economy are strongly contingent on the development of dependable information processing systems, and in some cases on integrated systems. The fault-tolerant multiprocessor appears to be a near-term candidate upon which to base a dependable integrated system. Airline practice, computer sizing, software, technology, and reliability issues have been investigated in this context and found to be compatible with the architecture.

A systematic approach to aircraft performance and survival is to use a distributed system with redundant sensors and effectors, dedicated local processing, highly dependable data communications, and a highly dependable central computer. The multiprocessor architecture is strongly compatible with such a system, although more than one multiprocessor may be required in order to achieve a high level of damage tolerance.

Reliability model predictions indicate that computer maintenance can usually be postponed for at least tens of hours following a module failure by the inclusion of one or more extra spare modules of each type. As long as certain minimum criteria are met before each takeoff, it should be possible to achieve a computer failure probability of the order of 10^{-9} in a ten-hour flight. This will depend critically on module MTBF's, which would have to be greater than is typical of current avionics practice. Methodologies do exist, however, for achieving the necessary MTBF's. Another key factor in achieving the low computer failure rate is the rate at which the computer can be reconfigured and subjected to self test. This architecture lends itself well, by virtue of its low degree of processor dedication.

Project Structure

The following tasks were treated in the work period.

TASK 1: Multiprocessor Architecture Evaluation
- A. Establish New Baseline (Chapters 1,2,4,6,9)
- B. Reliable Software (Chapter 6)
- C. Reliability Modeling (Chapter 5)
- D. Minimum Configuration (Chapter 10)
- E. Demonstration (Chapter 10)

TASK 2: Appraisal of Aircraft Operating Environment
- A. Cost and Effectiveness (Chapter 3)
- B. Computational Requirements (Chapter 3, 9)

C. System Integration (Chapter 3)

TASK 3: Technology Assessment (Chapters 7, 8).

Organization of This Volume

The first two chapters serve as a tutorial review of the system and multiprocessor architecture, as updated during the work period.

Chapter 3 presents a summary of the activity devoted to studying the aircraft application, primarily by the MIT Flight Transportation Laboratory.

Chapters 4 through 9 treat specific issues relevant to the architecture, including reliability, software, technologies, and sizing.

Chapter 10 describes a physical emulation of a multiprocessor of this type, and its use in an autopilot demonstration under fault injection conditions.

CHAPTER 1

SYSTEM CONTEXT

This chapter is concerned with the architectural considerations for a highly reliable and available multiprocessor computer for on-board integrated system management. A highly reliable computer is necessary, but not sufficient, for the implementation of life-critical control functions such as active controls, total fly-by-wire, and total system management. Therefore, before taking up the multiprocessor's architectural details, we will consider its significance in the context of the whole aircraft.

1.1 Sensors and Effectors

For purposes of information processing analysis, it is appropriate to regard the aircraft as a system of sensors and effectors, where information processing elements derive inputs from sensors and generate control signals to effectors. Effectors, which are displays and actuators, operate upon the man-machine environment via dynamics and human responses, producing effects that are measured by sensors. Figure 1.1 illustrates this model.

A significant point of departure from this model occurs at the next lower level of abstraction in contemporary systems, in that redundancy occurs in the form of independent and often dissimilar subsystems, as shown in Fig. 1.2. Subsystem failure is a routine occurrence, and in most cases has less than a catastrophic impact on flight safety. To implement systems in this manner, the flight crew is employed as a system integrator, with ultimate responsibility for failure detection, identification and recovery for the vehicle. The use of human judgements and responses in this manner constrains the design and utilization of the aircraft and its operational parameters, as for example the static stability, structural strength, traffic headways, etc.

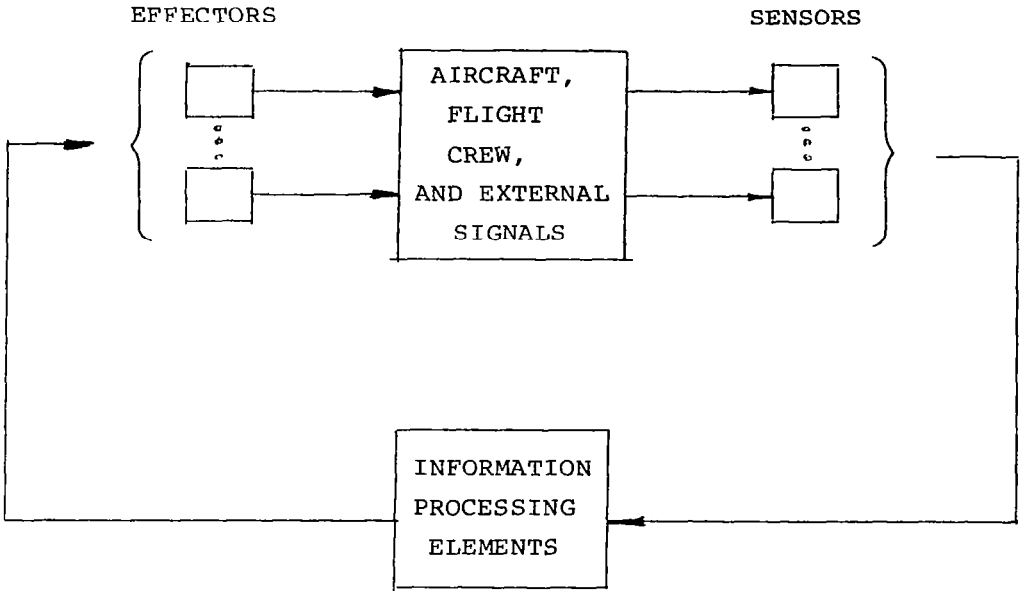


Fig. 1.1 Information Model of Aircraft.

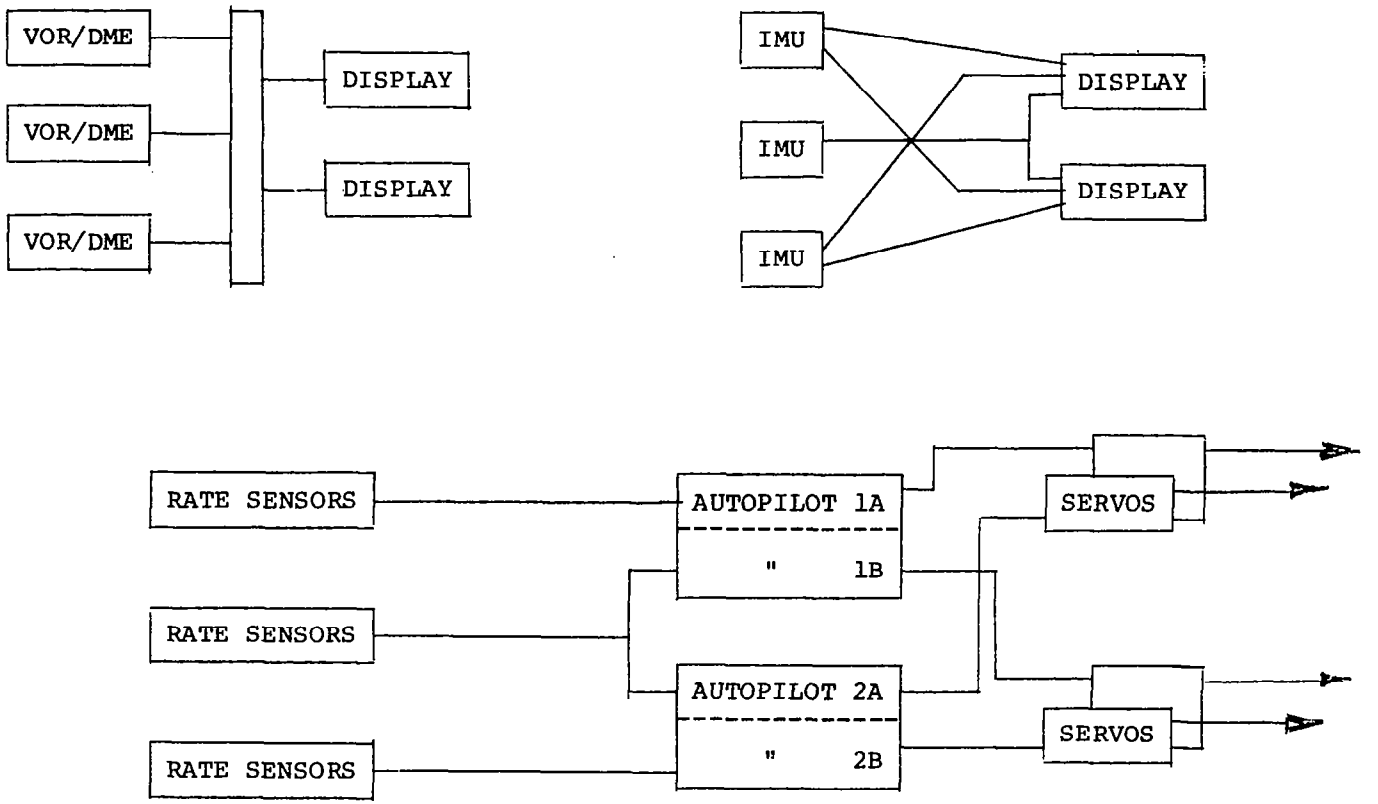


Fig. 1.2 Examples of Contemporary Redundant Independent Subsystems.

The implementation of substantially more sophisticated and autonomous controls calls for capabilities that are apt to require the creation of an integrated system. The incorporation of adequate redundancy and contingency into integrated systems poses significant problems, however. This is the issue which we now address, in the context of the system model of Fig. 1.1.

Since one can never rule out the occurrence of unlikely events, one must resort to probabilistic criteria for system design. Given that sensor and effector subsystem failures will occur in routine fashion, we seek a number of characteristics in subsystem and system design that enhance the probability of system survival. Without going into detail, one can identify broad chracterizations such as Mean Time Between Failure (MTBF), failure independence (i.e. absence of correlation among failures in distinct subsystems), and robustness in the sense of recovering from all failures for which it might potentially be possible to recover. The last of these items is not directly quantifiable. It requires the anticipation of all permutations of failures that will occur in the lifetime of a fleet of systems, and the creation of "contingency modes" of operation.

The desirable characteristics mentioned in the preceding paragraph for systems and subsystems suggest the following design philosophy for critical integrated systems. First, sensors and effectors should be diversified to the point where failure correlations are adequately small. Diversification should not be excessive, however, as it is counterproductive to the second objective, modularity. Modularity means the use of identical modules for multiple functions as well as redundancy, and serves MTBF, logistics, and contingencies. Modularity is beneficial in the contingency sense, as it tends to minimize the variety of contingency modes as well as to make it easier to create algorithms to cover families of contingencies.

With adequate diversification and substantial modularity, an integrated system can combine high survival with systematic utilization of resources. To the extent that contingency modes are algorithmic, the potential exists for supplanting or replacing the judgement factor of current systems with mechanized information processing. Thus we arrive at the third objective of the design philosophy, which is to have an information handling medium that is reliable and available to a degree of probability well in excess of that required for the survival of the aircraft itself.

The structure of a suitably dependable information processing system is the subject for the remainder of this chapter. Prior to closing this section, however, it should be pointed out that the information system embraces both non-critical subsystems and critical self survival. A significant aspect of the system architecture, therefore, involves the assignment of redundancy and survivability to the distributed portions of the system.

1.2 Local Processing

The highly survivable information system whose motivation was established in the preceding section could in principle be located in a single enclosure, with dedicated connections to each sensor and effector. As a practical matter, a certain degree of distribution of the information system is inevitable. Most of the sensors in use today as well as those envisioned for the future rely on electronics for manipulation, interpretation, and perhaps testing. In times past, the notion of system integration carried the implication of using a single large computer on a shared basis for as much of the information processing as possible. This attitude reflected the high risk and cost of digital computers. In the contemporary view, by contrast, the digital computer is less costly and less risky, particularly when it is used for functions of limited size and sophistication. Consequently, rather than trying to push information processing towards a central facility, the tendency today is to take advantage of the specialized electronics for each subsystem, but also the use of an appropriate amount of digital computer processing local to, and/or dedicated to, the subsystem. The fact that subsystems are individually non-critical makes it possible to eliminate fault tolerance as a local processing requirement, provided that the MTBF of the processing equipment is at least commensurate with that of the remainder of the subsystem.

The localization and "de-criticalization" of some of the information processing functions of the system carries a substantial list of benefits, as follows:

1. Bandwidth and Reaction Speed

A central computing facility is a good way to share resources such as memory capacity and sophisticated processing, but it is a poor way to accommodate a mixture of these things along with a requirement for high data bandwidths and fast reaction speeds. Fortunately for the system architect, the majority of high bandwidth and fast reaction

operations are, or can be, dedicated to a sensor, an effector, or a sensor-effector subsystem. The use of local processing relieves a fault-tolerant central computer of a speed requirement that it is ill-equipped to handle, especially given its principal burdens of self survival.

2. Task Switching Overheads

A related benefit of local processing stems from the penalty incurred in time-sharing a central computer among numerous small tasks. The overhead consists of software, execution time, and propensity for data interference.

3. Simplex vs. Redundant

Where local processing can be simplex (non-redundant), a direct cost saving accrues purely on the basis that the mass of redundant hardware is thereby made smaller.

4. Software Partitioning

By partitioning the software for dedicated subsystem functions into dedicated computers, one realizes several advantages. First, it becomes easier to manage the software generation, in that different programming organizations can generate software for different subsystems using different languages and different computers, should these be appropriate. Second, the operating system software of a small computer dedicated to a modest functional scope can become simplified almost to the point of non-existence. Finally, the system is made change-tolerant, by virtue of the isolation afforded by separate computers. That is, the only software needing re-validation after a change is that software co-resident with the changed software in the same computer.

5. Uniform Interfaces

In furtherance of the aim to employ modularity, the liberal adoption of local information processing tends to support the creation of uniform interfaces among subsystems. Such interfaces would be digital, with the possibility of employing codes or echo checks, and would be suitable for dedicated links, buses, and data networks alike.

To sum up the issues of local processing, it has been found that local information processing, including digital processing, can be beneficial for a highly survivable integrated control system. Such distribution of digital computers is clearly feasible at present; indeed, there appears to be an accelerated growth in the number of instances of digital processing in avionics sensors and effectors.

1.3 Data Communications and Information System Structure

At this point in the discussion it has been established that the information processing system possesses non-redundant, non-critical elements local to subsystems, but that the information system as a whole is highly survivable. We have implied that the non-critical failures will be few enough in number on any given flight so that valid contingencies exist and can be found. Exceptions are to be highly improbable. Survival of the aircraft will require the survival of minimum levels of sensors and effectors, motive power, structural integrity, and the information system. Survival of the information system will require not only the survival of minimum levels of local processing for sensors and effectors, but will further require the survival of system integrity, which is an unambiguous successful collaboration of surviving modules.

Whereas system integrity can in principle be embodied in a wholly distributed system to some degree, the subject system architecture employs a fault-tolerant central computer for the maintenance of system integrity. This computer also serves those digital computation functions that are non-dedicated, i.e. which involve the coordination of separate subsystems. Between the central computer and the local processors is a data communication facility, which is composed of non-critical elements, but whose partial survival is critical. The information system structure is shown in Fig. 1.3.

We distinguish three different fundamental fault-tolerant data communications structures. They are the dedicated, or star connection of Fig. 1.4a, the redundant bus connection of Fig. 1.4b, and the network connection of Fig. 1.4c. Each approach has pro and con characteristics, which will not be treated here in detail, but rather briefly discussed.

Dedicated connections are simplex in most cases. The failure of one such connection affects at most one non-critical subsystem. The

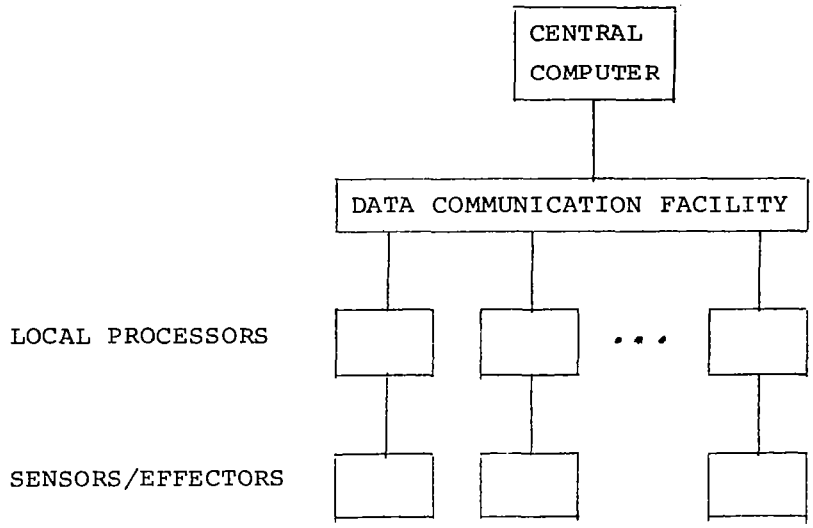
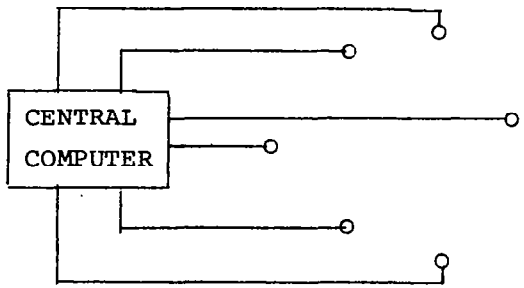
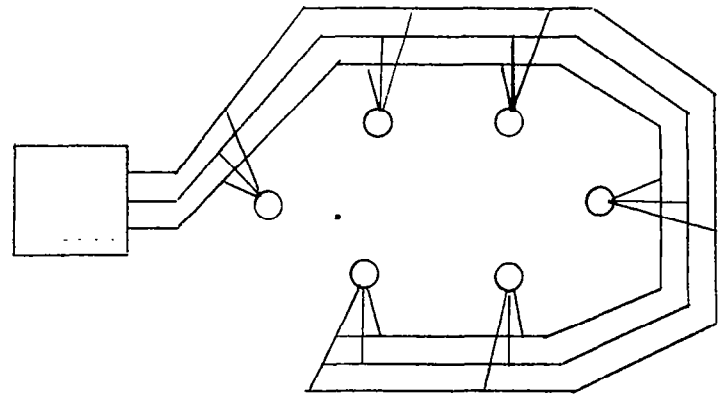


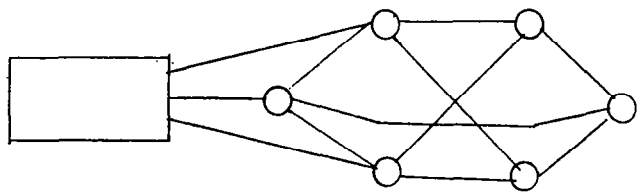
Fig. 1.3 Simplified Representation of Integrated Information System.



a. Dedicated



b. Redundant Bus



c. Network

= Central Computer

= Local Processor

Fig. 1.4 Data Communications Structures.

requirements imposed on the central computer's interface will be taken up in the next chapter. The growth potential of dedicated links is halted once the central computer's enclosure has been built. They moreover tend to under-utilize the interface electronics and connection medium. Their advantage over the other two approaches is their inherent graceful degradation.

A redundant bus abandons graceful degradation, but it better utilizes equipment and it has good growth potential. An individual element can fail so as to make one simplex bus useless. An N-redundant bus can in principle support (N-1) failures, if they are uncorrelated. The use of dispersed bus couplers rather than direct subsystem-bus interfaces tends to minimize bus failure correlations. The nominal topology of a bus is such that point-to-point links are not useful. Variants such as rings, however, can use point-to-point links. This issue is raised because of the degree to which fiber optics are considered promising for lightning strike immunity. So far, only point-to-point fiber optics links have appeared to have a solid foundation.

The network is in some senses a compromise between dedicated and bus connections. The network topology is such that the loss of numerous links is tolerable. The network's linkage mass is comparable to that of the redundant bus, because the network does not need full replication as does the bus. In this respect the network is like the dedicated approach. Error detection is necessary for achieving this, however, which will impose some overhead on the method. Other overheads exist in the form of switching hardware at each node and management software in the central computer. Equipment utilization is more or less comparable to that of the bus, and bus protocols are used for network data handling [1].

It may be presumed that all three of these methods will employ serial data transmission, and that the interface to each subsystem will be comparable, albeit not identical, for the different methods. The interface at the central computer's end is virtually identical for the three systems, differing only in the number of input-output access modules needed. There is no reason why all three of these methods cannot be used in different parts of the same system.

1.4 Central Computing and System Management

This final section of the chapter on System Context will serve to introduce the material of the subsequent chapters describing architecture, reliability, and performance of the central multiprocessor.

The point has already been made that the survival of the aircraft depends not only on the survival of a suitable minimum subset of the sensors and effectors, but also on the ability of the system to "mobilize" them into coordinated action. Such mobilization has a redundancy management aspect wherein data access is maintained. It also has a contingency aspect, wherein the vehicle dynamics are controlled via surviving effectors on the basis of information derived from surviving sensors. The adoption of a fault-tolerant central computer into the system structure makes it possible to assign both the redundancy management and the contingencies in an unambiguous way to a hardware-software entity whose failure and nonavailability are highly improbable.

One can characterize the central computer, then, as having a primary function of simply surviving, along with the functions of system redundancy management and contingency management. In addition to these functions, the central computer has other natural system roles owing to its hierarchical position in the information system structure. These roles include the coordination of sensor-effector activities, of which a digital autopilot is an excellent example. In some cases this function alone can account for a substantial fraction of the central computing resources. Another role is that of command, where there may or may not be interaction with the flight crew. This can be viewed somewhat abstractly as the maintenance of system order and purpose by the delegation of tasks to hierarchically subordinate function centers both within and without the central computer.

The architecture of the central computer is strongly driven by the urgency of preserving a valid data stream. Failure to do so could have catastrophic consequences, such as loss of the aircraft state vector, loss of configuration data, or loss of command. For this reason, every piece of data that is transferred, processed, or stored, is manipulated in triplicate. That is, every processor, memory, and internal bus is operated together with two others in a group denoted a "triad." When a triad delivers a piece of data, three copies of the data are delivered, one from each triad member. The recipient of the

data is also a triad, each member of which has access to all three copies of the data. Each member of the recipient triad makes an independent determination, by voting, of the correct version of the data it receives. If all copies agree, the correct version is obtained from any of the copies. If one copy disagrees, the correct version is taken to be the majority function of the copies. This methodology is the well-known and familiar triple-modular redundancy (TMR) approach.

The central computer is a variant of the classical TMR structure in two ways. First, it is able to replace failed triad members with spare modules. This particular variation of TMR is another well-known and familiar approach, called hybrid redundancy. The second variation is the use of parallelism in the processors and to a certain extent in the memories. Parallelism is a significant dimension because it permits spares to be pooled, and it permits testing to be conducted on some modules while the rest of them are engaged in critical on-line functions.

The parallel-hybrid TMR structure is potentially capable of having a high degree of fault tolerance and high system reliability. It is a costly approach in comparison to a simplex computer, not only because of its hardware triplication, but also because of overheads produced by its parallelism. Nevertheless, this approach is strongly to be recommended, because of several factors:

1. Alternate structures, less expensive in principle, have other cost sources that are not immediately obvious.
2. Procurement, maintenance, logistics, and reliability are all favorably impacted by the use of many small processors as opposed to one large one.
3. The economy of sparing is very high in this approach.
4. This is the only approach known by the authors to be capable of the degree of latent fault exposure that is required to achieve the desired degree of survivability.

There is no unique parallel-hybrid computer design. It can be a multiprocessor, a multicomputer, or a combination of the two. Additionally, numerous architectural alternatives exist. The next chapter describes a design approach based on a multiprocessor structure.

As a final note on the system role of the central computer, this element, like any other system element, can be made redundant for purposes of surviving physical damage. For a life-critical application,

it is likely that this would be done, with perhaps two central computers in two different locations in the aircraft.

CHAPTER 2

THE ARCHITECTURE OF A PARALLEL-HYBRID REDUNDANT MULTIPROCESSOR

The multiprocessor architecture described in this chapter is one that needs to meet severe dependability requirements while at the same time providing substantial throughput with moderate hardware cost. The approach taken was to use conventional processors and conventional memories interconnected in such a way that no single-point failures exist, that failed modules can be retired and replaced automatically, and that as little as possible need be assumed about the nature of random failures.

The architecture has several dimensions. The first area to be discussed is the architecture as viewed by the programmer, where for the most part the redundancy of the computer is invisible. A subsequent section treats the redundancy aspect of the computer, followed by sections dealing with synchronization, configuration, and fault management.

2.1 Nominal Organization

A multiprocessor, loosely defined, is a computer with several processors and a single (possibly multiport) memory accessible to all processors. In the extreme, all instructions and data reside in memory available to any processor, so that processors are "anonymous." Given a suitable state vector, any processor can execute any procedure from any starting point. Motiviations for multiprocessors are typically to increase productivity and availability at the same time, although these two purposes are largely competitive. At any rate, parallelism is intrinsic to the multiprocessor, as each processor is able to execute a different concurrent procedure subject to limitations imposed by resource sharing and sequential constraints on the procedures.

2.1.1 Memory Access

A "canonical form" of a multiprocessor is illustrated in Fig. 2.1, which introduces the notion of memory private to each processor in addition to the common memory. The rationale for this private, or cache, memory stems from the limitations imposed on parallel operation by memory access constraints. In a multiprocessor with highly parallel memory access, memory conflicts would occur only when individual units of data are simultaneously requested, or are locked for sequential conflict resolution. This would be the optimum structure for parallelism, and the cache memory's role is reduced to a possible enhancement of processor execution speed.

In the parallel-hybrid redundant multiprocessor, on the other hand, the memory access is highly serial, for reasons dictated by reliability and economy. This essentially means that the memory has a single port, and that the throughput of the multiprocessor is governed by the bandwidth of this memory port. In this case, the cache memory has a significant role in enhancing parallelism. The combination of processor and cache is a true computer, capable of performing elaborate operations on input data in response to terse commands. This means that the common memory can contain programs written in a language level higher than the processor's machine-language level, and that the processor-cache unit can interpret the higher level statements during the time that other processor-cache units are accessing the common memory. In this mode of system operation, which is really a form of "virtual machine," a memory port of moderate bandwidth can support an instruction execution "bandwidth" that is, at least in principle, almost arbitrarily large.

The degree to which the instruction execution bandwidth can exceed the common memory port bandwidth depends on the parameters of the cache memory, the terseness of the higher level language, and the relative amount of input and output data for each independent procedure. Clearly, the enlargement of the cache memories tends toward a multicomputer organization. Indeed, at some point the total cache capacity becomes adequate to contain everything in common memory, and the usefulness of common memory is reduced to the buffering of inter-process data.

It should be noted that the use of cache memory to eliminate common memory has a significant side-effect. Earlier in this section it was pointed out that in the extreme, the processors in a multiprocessor

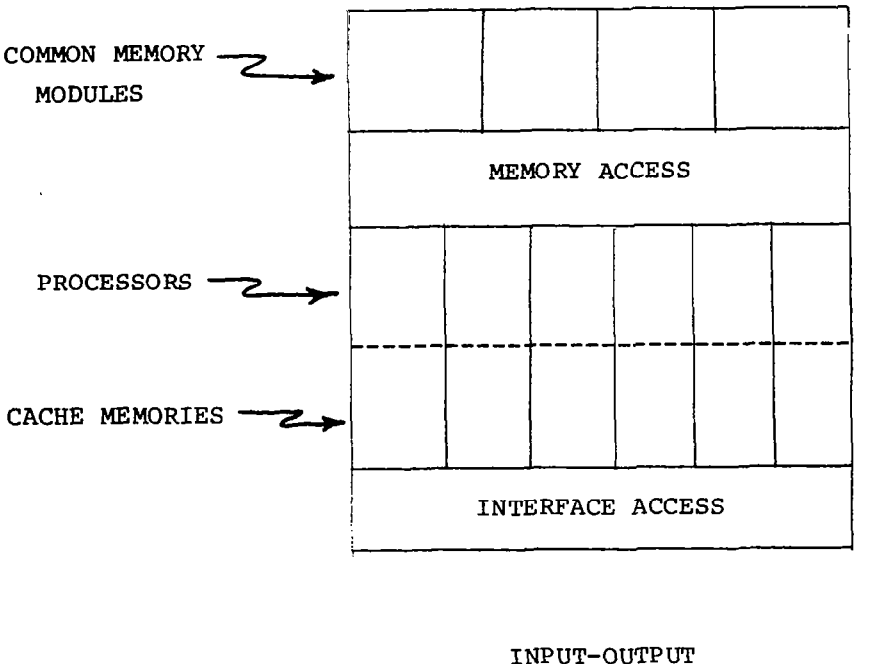


Fig. 2.1 Multiprocessor Functional Form.

are anonymous. This characteristic is significant to our application because of the frequent reconfigurations that need to take place in this computer for latent fault exposure, which is described further on. Anonymity also provides an intrinsic mechanism for dynamic load distribution among available processing resources. The cache memory acts to reduce the anonymity of the processor. To put it another way, the degree of anonymity is determined by the ease of reloading the cache memory. With zero cache memory, anonymity is greatest. As cache memory is increased to support instruction bandwidth enhancement, the anonymity of the processor-cache units depends on the amount of cache memory whose contents are unique to one processor. Note that the incorporation of identical procedural and other constant data, or indeed indentical variable data, in every cache memory has no adverse impact on anonymity.

The use of a cache memory in a sampled-data control application, such as the aircraft application considered here, is generally productive. The typical job step uses rather few data samples as input, and produces one data sample as output. The procedures used tend to lend themselves well to expression as macro-operations, i.e. higher level operations, such as floating point arithmetic, linear combination, elementary functions, vector and matrix operations, and so forth. The incorporation of procedures of this level as cache subroutines is reasonable and profitable in today's technology. The current high annual rate of memory density increase prompts one to observe that a fairly extensive set of procedures, and indeed a hierarchy of procedures, are appropriate for inclusion in cache memories to be produced several years hence.

The memory structure of the parallel-hybrid redundant multiprocessor includes cache memories for data and procedures, partly read-write, but mostly read-only, designed to enhance instruction bandwidth with rather little loss of processor anonymity. The common memory, although highly modular, acts as a single-port paged memory, accessible to one processor at a time via a serial bus with a built-in contention mechanism.

2.1.2 Interface Access

The external interfaces of the multiprocessor could in principle be accessed by the memory bus and be addressed in the memory space. The nature of input-output traffic, however, is enough different from that of memory traffic to justify a separate channel for input-output

data. This interface channel is strongly analogous to the single port memory, in that different processors access this channel at different times. It is a serial bus, also, but contention is resolved by software rather than hardware. The bandwidth is lower than that of the memory bus.

If one were to look for an analogy to the processor's cache memory in relation to the input-output data bandwidth, it would be the local processors dedicated to sensor and effector components. This point was anticipated in the preceding chapter by noting that bandwidth and reaction speed requirements are alleviated by local processing.

2.1.3 Functional Resource Allocation

The programmer sees this multiprocessor as a machine for executing job steps, largely corresponding to periodic sampled-data updates. The magnitudes of these job steps will vary considerably from one control function to another, but will require something of the order of a few milliseconds, on the average, of processor time per job step. The procedure for each job step is written in a suitable language, and resides in common memory. Typically, each job step is scheduled to occur at a given time or following a given event. The relevant dispatch data for each scheduled job step is kept in a queue, where it is frequently examined to see if the job step is eligible to be run, or invoked. The frequent examinations are conducted by processors that have completed their earlier assignments, and are available to undertake new ones. When an available processor determines the subset of eligible job steps, it selects one of them to invoke, where the selection can be made by a simple or complex algorithm. In this way, job allocation is dynamic, and adjusts itself to the momentary load distribution and to module failures. If processors are insufficiently anonymous, this can produce an overhead reduction in system throughput.

Memory management is not a critical issue in this application. No special consideration need be given to this area beyond the fact that the common memory, like the cache memory, will combine read-only and read-write memories. Test and reconfiguration problems are treated later on.

Input-output management in a multiprocessor can be more complex than it is in a single multiprogrammed computer, because as a single-port resource, it impinges on program parallelism. Depending on the statistics of external data traffic and of internal job steps, different

access strategies may be appropriate. The most straight-forward of these is to treat interface access as a single resource that is allocated to a single process for its exclusive use for the short period of time that a process requires access. Access may be granted on a priority basis or a first come first served basis. That is, when a processor needs interface access, it ascertains by means of flags in memory whether the interface is free. If not, the processor waits (with appropriate safeguards against lockup) until it becomes free. A more complex strategy is to quasi-dedicate a process to handle and buffer all input-output traffic. This is apt to improve the net input-output bandwidth at the cost of internal memory bandwidth. It moreover largely reduces the anonymity of that processor, with a consequent penalty in load sharing and recovery speed. The current presumption is that the former approach, using processor contention, will be preferable for this application.

2.2 Redundant Organization

The physical organization of the parallel-hybrid redundant multiprocessor is substantially more complex than the nominal organization outlined in the preceding section. A simplified module diagram of the computer is shown in Fig. 2.2. Superficially, this diagram appears the same as the nominal multiprocessor. The principal differences are that the buses for memory and interface access are redundant, and that the actual number of modules is three times the number of nominal modules plus some number of spares.

As mentioned in the preceding chapter, all activity is conducted by triads of modules and triads of buses. A module triad is formed by associating any three like modules with one another. This means that any module can serve as a spare for any triad. Such flexibility permits the best possible utilization of surviving modules. A single triad of bus lines is active at any one time for each of the memory and interface accesses. In other words, a three member subset of N bus lines is chosen on a quasi-static basis to serve as a bus triad.

Every module of every kind is able to receive data from all incident bus lines, and contains a decision element to formulate a corrected version of bus data. It is necessary for each module to know which three bus lines are the active ones. These three lines are connected to a voter in each module, thus constituting a TMR element. The three active bus lines carry three independently-generated versions of the data, each version coming from a different member of the triad that is transmitting the data. To accomplish this, it is necessary to assign each module to transmit on one specific bus line. Now if

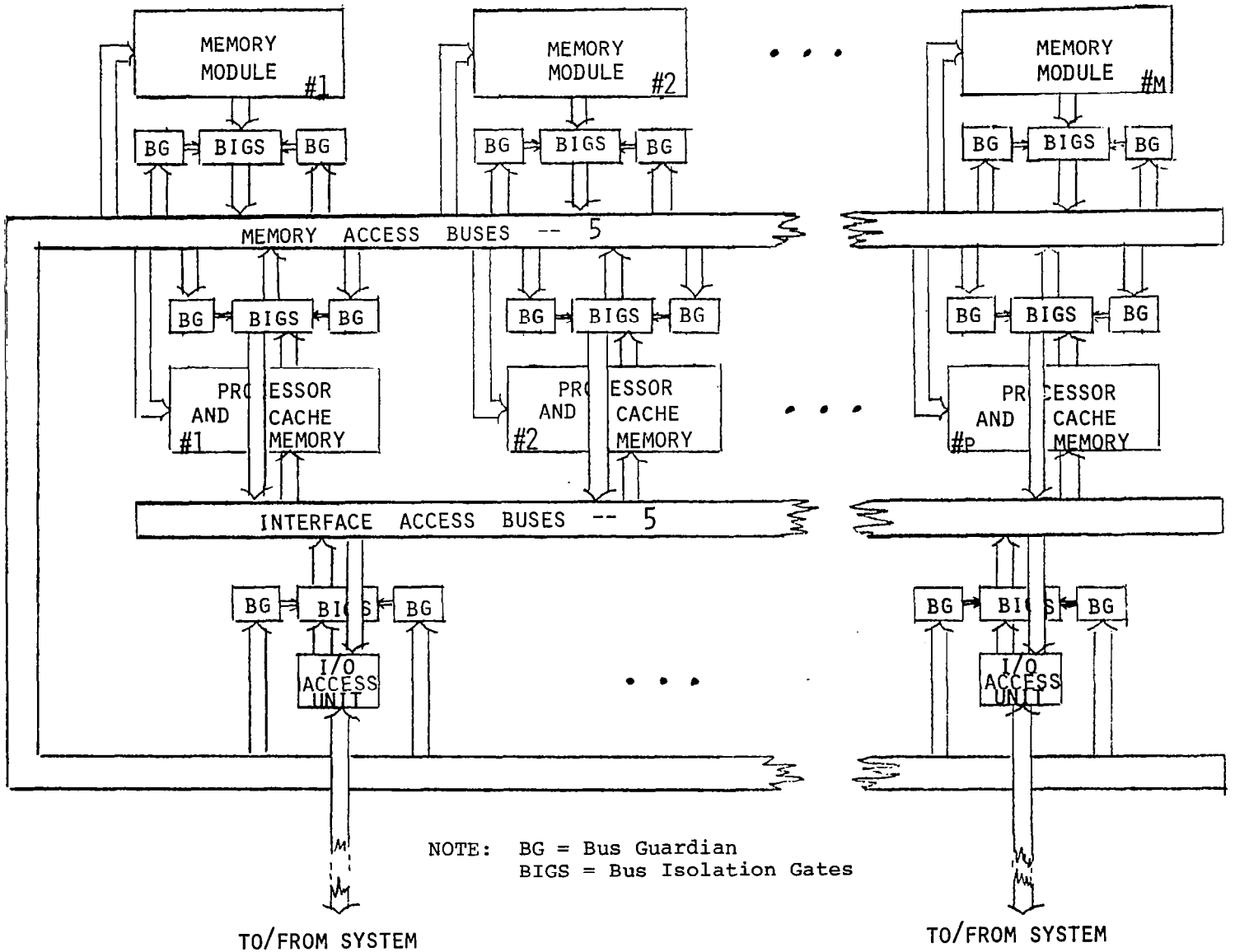


FIG. 2.2 SIMPLIFIED PHYSICAL DIAGRAM OF MULTIPROCESSOR

totally flexible module configuration is to be possible, it follows that the assignment of a module's transmission to a single bus line must be quasi-static and reconfigurable.

2.2.1 Bus Guardians

In addition to the redundancy described in the preceding few paragraphs, the redundant organization differs from the nominal one by virtue of the inclusion of independent submodules called bus guardian units in each processor, memory, and input-output access unit. Guardians are charged with governing the status of their associated modules. This includes power-on status, memory bus triad and transmission selection, and certain self-test configuration selections.

Each of the functions of the guardian has the characteristic that its failure modes have safe directions as well as unsafe ones. By biasing the failure modes toward the safe directions, it is possible to increase the probability of system survival. In general, the safe failure modes of a module are power-off, and bus transmission disconnected. To bias in this direction, one can employ redundant guardians in each module, and require agreement among them to establish power-on and bus transmission enable.

The connection of bus guardians is illustrated in Fig. 2.3. It should first be noted that the guardian principle depends heavily on fault independence. Therefore each guardian derives its power, its bus inputs, and its timing reference independently of all other guardians. It is moreover physically isolated from all other guardians and all modules. A particularly critical area from the isolation viewpoint is the control of the module's transmission interface onto the various bus lines. The bus isolation gates must be highly independent of one another, as must the guardian's enable signals to these gates. This is one of the crucial electrical and mechanical design aspects of the entire computer.

Bus guardians are addressable as part of the common memory address space, and are capable of receiving messages from any processor triad via the active memory bus triad. A message to a guardian contains commands which are staticized by the guardian and applied to its outputs until superseded by a new command message. In this way, the probability is remote that a failed module can assert more than one erroneous data stream. As a result, correct data can be determined by the bus voters, and the malfunctioning module can be switched to a silent state. It is

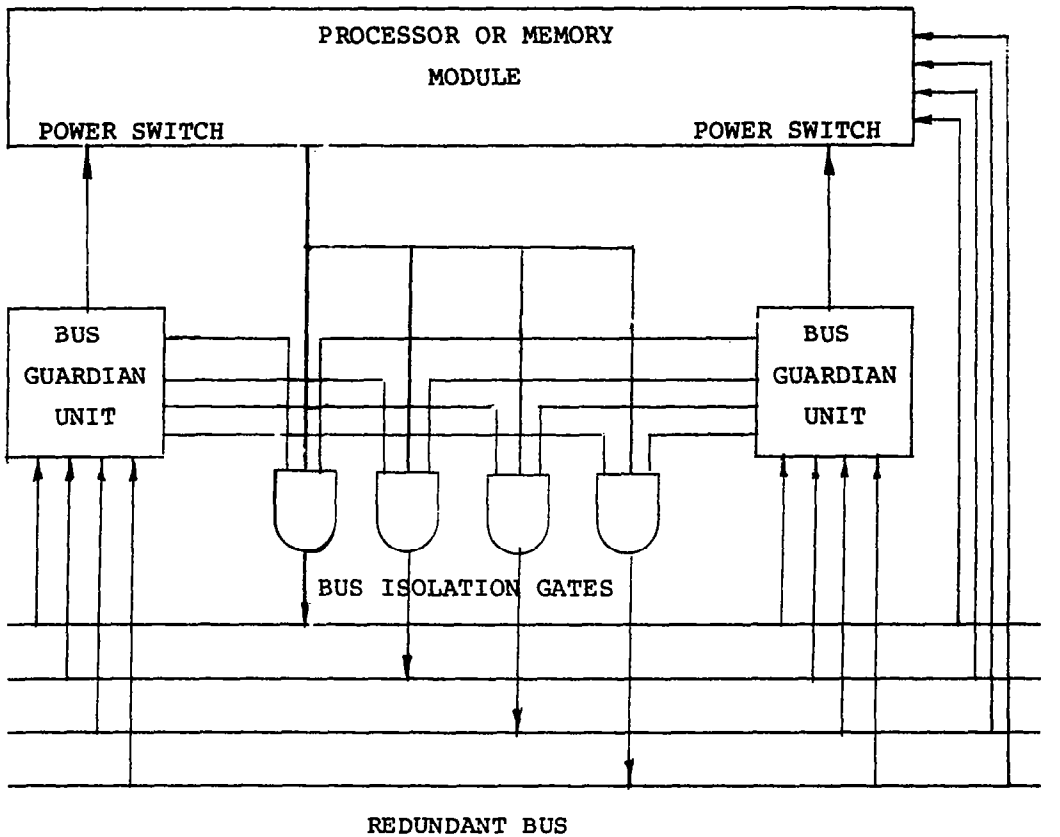


Fig. 2.3 Bus Guardian Connections.

noted in passing that certain failures of a bus isolation gate can render a bus line useless, in which case that active bus triad must be reconfigured to use a spare line. However, most guardian failures appear as passive failures of the processor, memory, or input-output access unit to which the particular guardian unit pertains.

Guardians are used as agents to convey the computer's configuration authority to all elements of the computer. They are highly secure against the random or willful malfunction of any single active transmitting module. They make possible the highly flexible reconfiguration referred to earlier.

2.2.2 Processor and Memory Modules

It was mentioned before that all modules and buses are organized into triads. In the case of processors and memories, there can be numerous triads in existence at the same time, but only one memory bus triad and only one interface bus triad. Each processor triad acts as one functional processor, of which several can work in parallel. Each memory triad acts as a page of memory, of which several can exist at one time, but only one can communicate at a time with a processor triad.

When a processor fails, its triad will attempt to complete its current job step, which it will be able to do unless a second failure prevents it. The period of vulnerability to a second failure will be a fraction of a second. When the job step is complete, one of the processor triads is assigned the task of reconfiguring the injured triad. When the erroneous module is identified, it is removed by commands to its guardians. If a spare is available, it is connected to the appropriate bus by its guardians, likewise upon command by the processor triad assigned to the reconfiguration. Triad identity will be assigned to the spare processor by a direct message. If no spares are available, the injured triad is retired. The resources of the multiprocessor are diminished by one processing unit, and the two unfailed members of the former triad are now available to be used as spares, should further failures occur.

The situation is much the same for memory modules. The principal difference is that memories are not anonymous. In fact, a read-only memory module is totally dedicated to its assigned function, and cannot be used as a spare. When a read-only memory triad is injured by the loss of a memory module, a read-write memory module can be used as a

spare. It must be loaded to agree with the surviving triad members before a second failure occurs. If no spare is available, the triad is reduced to a dyad, which is vulnerable to the next failure, at which time one memory page is lost. This is a significant departure from the flexibility offered by the anonymous processor triads. The eventuality of read-only memory failure must clearly be covered by the inclusion of adequate spares, either read-write memories for flexible pooled use, or extra dedicated copies of read-only memory.

2.2.3 Input-Output Access

Figure 2.2 indicates the existence of input-output access modules connected to the internal interface bus and also to the external environment. In the last section of the preceding chapter, it was pointed out that the external interfaces of the computer could alternatively support dedicated, bussed, or networked link structures to the sensor and effector components. The redundancy structure at this point depends on the redundancy desired in the external interface.

The simplest conceptual structure is for a triple-redundant interface, such as a redundant external bus, where the triple modular redundancy structure is extended through to the component interfaces. Each external bus line can be dedicated to a different input-output access module, which in turn is assigned by its guardian units to transmit on one of the active interface bus lines. More complex variants are possible, in which each access module performs error correction by voting on incoming data from the external bus.

When an external interface is non-redundant, the strategy would be to assign it to a single access module, where the module would transmit on all three active interface bus lines. A malfunctioning access module could pollute the entire interface bus, but with suitable encoding and protocol there would be no serious consequences to the state of the system. The offending access module could be discovered and disconnected by bus guardian commands conducted over the memory bus, the major penalty being a time loss on the remainder of the input-output interface of the computer. For dedicated links, the loss of the link is non-critical by hypothesis. For a network, whose survival is assumed critical, the computer must interface with the network in several places via several distinct access modules. Each such interface would be simplex, but the system would survive the failure of all but one of them.

2.3 Synchronization

The employment of independent redundancy requires some form of synchronization among the independent data sources. Soft, or loose synchronization involves such operations as buffering, comparing or voting, signalling consensus, and marking completed intervals. These can all be done by program, given suitable intermodule data links. Hard, or tight synchronization involves hardware comparison or voting, and a common time reference, where loose synchronization can employ separate time references.

Tight synchronization is employed in the parallel hybrid redundant multiprocessor. It provides the basis for solving some problems and presents some problems of its own. A common time reference, or clock, that supports hardware voting, allows instantaneous validation of internal data, configuration control, and, in some cases, interface data. In this way, it helps to make the redundant multiprocessor resemble the nominal one, which is advantageous to programmers at all levels.

The problems of common clocking stem primarily from the fact that it is critical to computer operation in the dynamic sense. The timing reference must be continuous and must remain within tolerances. A second consideration is that common clocking results in time-correlated data transfer, which is subject to correlated malfunction if subjected to external radiation of electromagnetic energy beyond the levels tolerated by shielding. The first problem has been largely solved by the development of a phase-locked redundant clocking system. The second problem is intrinsic to all synchronization, but is more severe for tight synchronization. The problem also exists in principle for any degree of shielding. When the statistics of such interference are known, the problem can be addressed in the time domain by encoding for error detection, rerun for recovery, or repetition for time independence.

The fault-tolerant clock shown in Fig. 2.4 consists of a set of independent phase-locked oscillators arranged so that the failure of one or more of the oscillators (up to a design limit) does not destroy the phase lock of the survivors. The clock signal from each oscillator is distributed to every module and guardian, so that each can make an independent determination of clocking edges. These independent determinations are made by circuits called clock receivers. In normal, nonfailed operation, the outputs of all the clock receivers are in phase lock with each other and with all the oscillators. The same phase

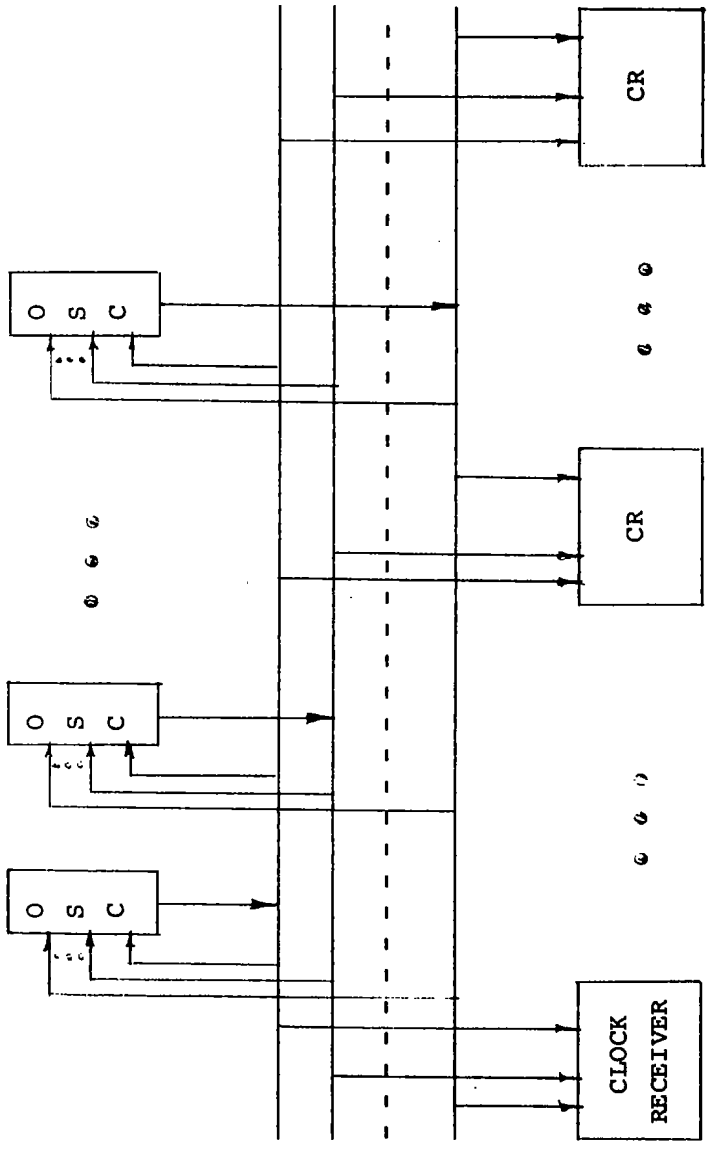


Fig. 2.4 Fault-Tolerant Clocking System.

lock holds when an oscillator fails. The failure of a clock distribution line appears as an oscillator failure, and the failure of a clock receiver appears as a failure of the module or guardian that contains it.

The fault-tolerant clocking network is vulnerable to undetected, or latent, faults. This issue is further treated in subsequent sections.

2.4 Fault Detection, Identification, and Recovery

The central computer is designed to have a highly improbable loss of capability. One can roughly quantify this statement by saying that one of these computers should exhibit a total failure rate of less than 10^{-9} failures per hour in a flight of up to ten hours. This virtually rules out the use of ordinary triple modular redundancy, as the MTBF's achievable in large scale production have been consistently too low for such reliability without replacement of failed modules. Therefore some form of hybrid redundancy is needed. In a simplistic view, hybrid redundancy works by substituting a spare the first time the TMR voters disagree. This view has the shortcoming of not taking latency of faults into account. That is, the first fault may not result in any voter disagreements, whereas when combined with a second fault, it may frustrate recovery. A prerequisite for achieving highly improbable failure in a hybrid system is therefore to expose latent faults by systematic exercising, or "flexing" of all logic elements. The question remains of how often such flexing must occur. Hopkins and Smith [2] have shown that the flexing period must be of the order of seconds for a reasonably sized system with module MTBF's in the ten-thousand hour range. Clearly, then, flexing cannot be relegated to pre-flight checkout, but must rather be conducted routinely in flight. An ordinary hybrid TMR system cannot routinely test itself when performing critical functions, as it is vulnerable during these times. A parallel hybrid TMR system can do this, however, and this becomes an integral part of the computer's architecture. The remainder of this section is devoted to this issue.

The latency problem poses an interesting design dilemma. Redundancy is employed to mask the effects of faults upon the system as a whole. But redundancy requires flexing of all logic, and moreover requires that all possible faults that are created by flexing be made visible to the system, and not masked. The resolution of this dilemma requires reconfiguration of all independent system elements plus the selective generation of faulty symptoms to verify detection

mechanisms. This is why, for example, quadded or interwoven [3] logic is not proposed for this application, as it cannot be reconfigured and tested on line. The same holds for many of the error-correcting coded memory and arithmetic units that have been designed.

In the parallel-hybrid redundant multiprocessor, an error correction mechanism exists in every module in the form of a voter. Each voter must be tested routinely to ensure that its error correcting capability is undiminished, which will be explained further. Of all the voters only those in the processor modules have the additional capacity to detect, as well as to correct errors. This is not to save equipment. Rather, the processor is the only kind of module in the computer that can utilize the information. Processor bus voters under normal conditions will correct single bus errors and will set error latches to indicate which of the buses was in disagreement. At this time the processor can record the identity of the nominal user of the bus for diagnostic purposes. A processor triad can flex its own voters during a test job step by having each triad member purposely utter independent bus data that causes all possible kinds of bus errors. To pass the test, all triad members must receive the same data, form the same corrected result, and indicate the same disagreement patterns in their error latches. This is a relatively simple test procedure, which can be conducted by a processor triad under test while other triads carry on normal functions. In a sense it qualifies the triad to conduct further testing, in which the triad's voters are the decision elements.

The remainder of the system testing function is carried out under the assumption that the processor voters and error latches are operational. We have referred to voters as being responsive to "errors" on the buses. In our nomenclature, a "failure" is a physical malfunction that may cause a "fault," which is a logic variable malfunction. An "error" is a manifestation of a fault when the fault affects the outcome of a data transfer. The test process involves the conversion of every fault into an error, by making calculations whose results are sensitive to each logic variable. Each bus and module, including voters, guardians, isolation gates, clock receivers, oscillators, and data and power interfaces must be exercised in depth.

No attempt is made here to give an exhaustive description of these test algorithms. Some of them indeed remain to be formulated. A few brief outlines will serve to explain the important considerations.

Processor testing involves fairly conventional self-test approaches, except that coverage needs to be higher than that which is typically obtained in computers. A guideline, then, for processor design is to eliminate obscure and pattern-sensitive sequences as much as possible. This would indicate, for example, that a microprogrammable bit-sliced structure is to be preferred over a fully integrated, "tricky circuit" processor. The cache memory is also tested by a conventional program approach. Address faults and pattern sensitivities present the most important problems to be solved.

Memory module testing is similar to cache memory testing. The memory voters are tested by sending single-error messages to a memory triad over the memory bus and verifying correct responses from the triad members.

Input-output access modules are also tested by messages from processors. Where voters are used, they are tested in a manner similar to memory voters. Simplex access units are tested in conjunction with input-output links.

Guardians are tested by reconfiguration commands, and their voters are tested as memory voters are, by erroneous commands.

The clocking system presents a unique testing problem, both because it is nearly separate from the data handling elements, and because the testing of clocking circuitry is fundamentally different from the testing of other logic circuits, which are testable once a valid clock exists. Latencies in the oscillators can occur in the phase-locking circuitry. This is probably the most vulnerable area for in-flight testing. If the a priori probability of failure in phase lock circuits is sufficiently low, it may be possible to perform these tests only at preflight time. Clock receiver latencies, however, can be tested in one module at a time with minimal system vulnerability.

We might summarize the fault detection process as the arrival of disagreement errors at the voters of a processor triad, stimulated by normal or test activity. The detection of a fault initiates the process of fault identification, which is the discovery of the module, bus, or other isolated element in which the failure resides. During the testing process for latent faults, there is relatively little ambiguity in the determination of faulty modules. In normal operation, however, an error on the bus can come from a number of sources. The identification of the faulty module generally requires the "rounding up

of suspects," that is, the listing of elements that transmit on the disagreeing bus. If a module fault is permanent, the module can be found by moving it to another bus. If the bus is faulty, reconfiguration will not move the error to another bus.

Intermittent faults are less easy to identify. When the source of an error eludes detection by disappearing, all of the suspect elements are assigned one demerit, and a reconfiguration is then made to distribute the suspects evenly on different buses. Subsequent error occurrences and reconfigurations will cause a preponderance of demerits to accumulate in the name of the faulty module.

The recovery process is one of assignment and initialization for modules, and voter and transmitter selection for buses. These are all accomplished by the bus guardian units upon receipt of commands from active triads executing system software. Recovery can take place even if single errors are present on the buses. In principle, therefore, an injured processor triad can reconfigure itself.

The use of program restart, or rollback, as a recovery mechanism is secondary, because it is neither very effective nor very easy to implement. The first level of system defense is the masking of errors by the TMR method. The additional system failure rate reduction achievable by rollback can not be measured, a priori, without an understanding of the applications software. It should be anticipated, however, that any event that defeats the TMR masking is apt to destroy the vehicle's state vector, which may or may not be catastrophic. In any event, some degree of program restart capability must be included to support power-up initialization and to deal to some extent with the eventuality of uncovered errors. This will affect both system software and application software.

CHAPTER 3

ASSESSMENT OF THE IMPACT ON COMMERCIAL AIRCRAFT

The next generation of commercial aircraft are expected to make extensive use of digital avionics, with this technology largely replacing most of the existing analog techniques. The general arguments for implementing digital avionics lie in two areas: (1) safety and (2) efficiency. Safety improvements are seen through (a) system integrity improvements through increased reliability and more effective use of redundancy; (b) pilot work load reduction; (c) improved communications capability and; (d) improved system performance. Efficiency improvements are expected due to (a) higher reliability (longer mean time between required maintenance actions) and (b) reduced life cycle costs, through simplified logistics support, a complete self check capability reducing unnecessary removals, a capacity to defer maintenance, and associated efficiency improvements of related subsystems due to the use of an integrated information system.

3.1 Functional Applications

The need for large scale deployment of fault-tolerant computation can be seen as arriving with the next generation of aircraft. Table 3.1 outlines the growth of flight guidance modes since the 1940's in Douglas Aircraft commercial transports, and is typical for the industry. The next Douglas transport is projected as having digital flight control and digital fly-by-wire control of sensors and actuators. The use of digital computation is also expected to extend beyond the flight control, navigation, and guidance function into other aircraft systems and into other functions.

Table 3.2 defines four aircraft functional areas: (a) Navigation and Guidance, (b) Stability and Control, (c) Air Traffic Control, (d) Aircraft Systems Management. These functional areas are ranked by the criticality of the function during or for each mission phase. This is done for current aircraft, next generation aircraft (1980's) and future

TABLE 3.1

GROWTH OF FLIGHT GUIDANCE MODES IN DOUGLAS COMMERCIAL AIRPLANES

(TYPICAL ORIGINAL MODEL)

MODEL	DC-4 1946	DC-6 1947	DC-7 1953	DC-8 1959	DC-9 1965	DC-10 1971	DC-FUTURE
HEADING HOLD	X	X	X	X	X	X	X
ALTITUDE HOLD		X	X	X	X	X	X
HEADING SELECT			X	X	X	X	X
AUTOMATIC-TURN COOR.		X	X	X	X	X	X
AUTO PITCH TRIM		X	X	X	X	X	X
TURN LIFT COMPEN.		X	X	X	X	X	X
VOR		X	X	X	X	X	X
ILS		X	X	X	X	X	X
VERTICAL SPEED				X	X	X	X
BACK COURSE ILS (FLT. DIRECTOR ONLY)		X		X			
INDICATED AIR SPEED HOLD					X	X	X
MACH HOLD					X	X	X
ALTITUDE PRESEL.						X	X
TURBULENCE						X	X
AUTOTHROTTLE					X	X	X
THRUST RATING						X	X
AUTOMATIC LANDING						X	X
GO-AROUND						X	X
TAKEOFF						X	X
COMMAND CONTROL WHEEL STEERING						X	X
INS NAV MODE						X	X
FLY-BY-WIRE							X
RELAXED STATIC STABILITY							X
GUST LOAD ALLEVIATION							X

TABLE 3.1(Continued)

MODEL	DC-4 1946	DC-6 1947	DC-7 1953	DC-8 1959	DC-9 1965	DC-10 1971	DC-FUTURE
MANEUVER LOAD ALLEVIATION							X
FLUTTER SUPPRESSION							X
TYPE OF AUTOPILOT ACTUATOR ELECTRONICS	HYD + PNEU	VAC TUBES	VAC TUBES + MAG AMPS	TRANSISTORS + MAG AMPS	TRANSISTORS + SOME ICs	INTEG CIRCUITS	DIGITAL
AUTOPILOT ACTUATOR MECH CONTROLS	HYD FLYING TABS	ELEC MOTOR FLYING TABS	ELEC MOTOR FLYING TABS	ELEC MOTOR HYD AIL MECH RUDDER + ELEV	ELEC MOTOR HYD RUDDER MECH ELEV + AIL	HYD VALVE FULL POWER	HYDRAULIC

SOURCE: Digital Flight Control Systems - Considerations in Implementation and Acceptance, Douglas Paper 6342, W.B. Yopp and S.D. McDonnell, 1975.

TABLE 3.2

FUNCTIONAL CRITICALITY ACCORDING TO MISSION PHASE

Mission Phase

1. TAKEOFF - Duration of phase: 1 minute

Functional Area	Criticality
(1) Navigation and Guidance	
A. Current Aircraft	
a. Area Navigation (R-Nav)	1
b. Inertial Navigation System (INS)	1
c. Autoland (Autothrottle)	0
d. Radar Altimeter	1
e. Autopilot	0
Modes: (for example)	
Attitude Hold	
Heading Hold	
Indicated Air Speed Hold	
Mach Hold	
Constant Vertical Speed	
Glide Slope Hold	
B. 1980's Aircraft	
a. Digital Autopilot with Category IIIb (see to taxi)	0
b. Hybrid Navigation	1
c. 4-D Flight Path Control	1
d. Heads Up Display	0
C. 1990's Aircraft	
(2) Stability and Control	
A. Current Aircraft	
a. Autopilot: Yaw Damper	0
B. 1980's Aircraft	
a. Cockpit Displays - Electronic Attitude Direction Indicator(EADI)	3
b. Cockpit Displays - Electronic Horizontal Situation Indicator (EHSI)	3
c. Flight Envelope Limiter	3
d. Ride Improvement System	1

TABLE 3.2 (continued)

C. 1990's Aircraft	
a. Full Scale Stability Augmentation System (Hard SAS)	4
b. Gust Load Alleviation	4
c. Flutter Mode Control	4
d. Maneuver Load Control	3
(3) Air Traffic Control	
A. Current Aircraft	
a. Weather Radar	1
b. Ground Proximity Warning System (GPWS)	0
B. 1980's Aircraft	
a. Digital Weather Radar Processing	1
C. 1990's Aircraft	
a. Digital Data Link (including Metering and Spacing Function)	1
b. Airborne Traffic Situation Display (ATSD)	1
c. Collision Avoidance System (CAS)	1
(4) Aircraft Systems Management	
A. Current Aircraft	
a. Fuel Control (Tankage)	0
b. Fuel Control (Engines)	3
c. Engine Inlet Control	1
d. Center of Gravity Indicator (Pre-flight)	0
e. Weight Monitor (Pre-flight)	0
f. Cabin Environment	1
g. Air Data System (Temperature, Pressure)	3
h. Master Caution System	3
i. Fire Warning System	3
j. Power Generation and Distribution	3
k. Automatic Braking System	3
l. Aircraft Integrated Data System (AIDS - Maintenance only)	1
B. 1980's Aircraft	
C. 1990's Aircraft	
a. Integrated Data Management System	1

TABLE 3.2 (continued)

Mission Phase

2. CLIMB - Duration of phase: 10-30 minutes

Functional Area	Criticality
(1) Navigation and Guidance	
A. Current Aircraft	
a. Area Navigation (R-Nav)	2
b. Inertial Navigation System (INS)	2
c. Autoland (Autothrottle)	0
d. Radar Altimeter	1
e. Autopilot	2
Modes: (for example)	
Attitude Hold	
Heading Hold	
Indicated Air Speed Hold	
Mach Hold	
Constant Vertical Speed	
Glide Slope Hold	
B. 1980's Aircraft	
a. Digital Autopilot with Category IIIb (see to taxi)	2
b. Hybrid Navigation	2
c. 4-D Flight Path Control	2
d. Heads Up Display	0
C. 1990's Aircraft	
(2) Stability and Control	
A. Current Aircraft	
a. Autopilot: Yaw Damper	2
B. 1980's Aircraft	
a. Cockpit Displays - Electronic Attitude Direction Indicator(EADI)	3
b. Cockpit Displays - Electronic Horizontal Situation Indicator (EHSI)	3
c. Flight Envelope Limiter	3
d. Ride Improvement System	1

TABLE 3.2 (continued)

C. 1990's Aircraft	
a. Full Scale Stability Augmentation System (Hard SAS)	4
b. Gust Load Alleviation	4
c. Flutter Mode Control	4
d. Maneuver Load Control	3
(3) Air Traffic Control	
A. Current Aircraft	
a. Weather Radar	1
b. Ground Proximity Warning System (GPWS)	0
B. 1980's Aircraft	
a. Digital Weather Radar Processing	2
C. 1990's Aircraft	
a. Digital Data Link (including Metering and Spacing Function)	3
b. Airborne Traffic Situation Display (ATSD)	3
c. Collision Avoidance System (CAS)	3
(4) Aircraft Systems Management	
A. Current Aircraft	
a. Fuel Control (Tankage)	3
b. Fuel Control (Engines)	3
c. Engine Inlet Control	3
d. Center of Gravity Indicator (Pre-flight)	0
e. Weight Monitor (Pre-flight)	0
f. Cabin Environment	3
g. Air Data System (Temperature, Pressure)	3
h. Master Caution System	3
i. Fire Warning System	3
j. Power Generation and Distribution	3
k. Automatic Braking System	0
l. Aircraft Integrated Data System (AIDS - Maintenance only)	1
B. 1980's Aircraft	
C. 1990's Aircraft	
a. Integrated Data Management System	3

TABLE 3.2 (continued)

Mission Phase

3. CRUISE
Duration of phase: 0.5 hours - 8 hours

Functional Area	Criticality
(1) Navigation and Guidance	
A. Current Aircraft	
a. Area Navigation (R-Nav)	2
b. Inertial Navigation System (INS)	2
c. Autoland (Autothrottle)	0
d. Radar Altimeter	1
e. Autopilot	2
Modes: (for example)	
Attitude Hold	
Heading Hold	
Indicated Air Speed Hold	
Mach Hold	
Constant Vertical Speed	
Glide Slope Hold	
B. 1980's Aircraft	
a. Digital Autopilot with Category IIIb (see to taxi)	2
b. Hybrid Navigation	2
c. 4-D Flight Path Control	2
d. Heads Up Display	0
C. 1990's Aircraft	
(2) Stability and Control	
A. Current Aircraft	
a. Autopilot: Yaw Damper	2
B. 1980's Aircraft	
a. Cockpit Displays - Electronic Attitude Direction Indicator(EADI)	3
b. Cockpit Displays - Electronic Horizontal Situation Indicator (EHSI)	3
c. Flight Envelope Limiter	3
d. Ride Improvement System	1

TABLE 3.2 (continued)

C. 1990's Aircraft	
a. Full Scale Stability Augmentation System (Hard SAS)	4
b. Gust Load Alleviation	4
c. Flutter Mode Control	4
d. Maneuver Load Control	3
(3) Air Traffic Control	
A. Current Aircraft	
a. Weather Radar	2
b. Ground Proximity Warning System (GPWS)	0
B. 1980's Aircraft	
a. Digital Weather Radar Processing	2
C. 1990's Aircraft	
a. Digital Data Link (including Metering and Spacing Function)	3
b. Airborne Traffic Situation Display (ATSD)	3
c. Collision Avoidance System (CAS)	3
(4) Aircraft Systems Management	
A. Current Aircraft	
a. Fuel Control (Tankage)	3
b. Fuel Control (Engines)	3
c. Engine Inlet Control	3
d. Center of Gravity Indicator (Pre-flight)	0
e. Weight Monitor (Pre-flight)	0
f. Cabin Environment	3
g. Air Data System (Temperature, Pressure)	3
h. Master Caution System	3
i. Fire Warning System	3
j. Power Generation and Distribution	3
k. Automatic Braking System	0
l. Aircraft Integrated Data System (AIDS - Maintenance only)	1
B. 1980's Aircraft	
C. 1990's Aircraft	
a. Integrated Data Management System	3

TABLE 3.2 (continued)

Mission Phase

4. DESCENT
Duration of phase: 10-30 minutes

Functional Area	Criticality
(1) Navigation and Guidance	
A. Current Aircraft	
a. Area Navigation (R-Nav)	2
b. Inertial Navigation System (INS)	2
c. Autoland (Autothrottle)	2
d. Radar Altimeter	2
e. Autopilot	2
Modes: (for example)	
Attitude Hold	
Heading Hold	
Indicated Air Speed Hold	
Mach Hold	
Constant Vertical Speed	
Glide Slope Hold	
B. 1980's Aircraft	
a. Digital Autopilot with Category IIIb (see to taxi)	2
b. Hybrid Navigation	2
c. 4-D Flight Path Control	2
d. Heads Up Display	3
C. 1990's Aircraft	
(2) Stability and Control	
A. Current Aircraft	
a. Autopilot: Yaw Damper	2
B. 1980's Aircraft	
a. Cockpit Displays - Electronic Attitude Direction Indicator(EADI)	3
b. Cockpit Displays - Electronic Horizontal Situation Indicator (EHSI)	3
c. Flight Envelope Limiter	3
d. Ride Improvement System	1

TABLE 3.2 (continued)

C. 1990's Aircraft	
a. Full Scale Stability Augmentation System (Hard SAS)	4
b. Gust Load Alleviation	4
c. Flutter Mode Control	4
d. Maneuver Load Control	3
(3) Air Traffic Control	
A. Current Aircraft	
a. Weather Radar	2
b. Ground Proximity Warning System (GPWS)	2
B. 1980's Aircraft	
a. Digital Weather Radar Processing	2
C. 1990's Aircraft	
a. Digital Data Link (including Metering and Spacing Function)	3
b. Airborne Traffic Situation Display (ATSD)	3
c. Collision Avoidance System (CAS)	3
(4) Aircraft Systems Management	
A. Current Aircraft	
a. Fuel Control (Tankage)	3
b. Fuel Control (Engines)	3
c. Engine Inlet Control	3
d. Center of Gravity Indicator (Pre-flight)	0
e. Weight Monitor (Pre-flight)	0
f. Cabin Environment	3
g. Air Data System (Temperature, Pressure)	3
h. Master Caution System	3
i. Fire Warning System	3
j. Power Generation and Distribution	3
k. Automatic Braking System	2
l. Aircraft Integrated Data System (AIDS - Maintenance only)	1
B. 1980's Aircraft	
C. 1990's Aircraft	
a. Integrated Data Management System	3

TABLE 3.2 (continued)

Mission Phase

5. APPROACH AND LANDING
Duration of phase: 2 minutes

Functional Area	Criticality
(1) Navigation and Guidance	
A. Current Aircraft	
a. Area Navigation (R-Nav)	1
b. Inertial Navigation System (INS)	1
c. Autoland (Autothrottle)	4
d. Radar Altimeter	4
e. Autopilot	4
Modes: (for example)	
Attitude Hold	
Heading Hold	
Indicated Air Speed Hold	
Mach Hold	
Constant Vertical Speed	
Glide Slope Hold	
B. 1980's Aircraft	
a. Digital Autopilot with Category IIIb (see to taxi)	4
b. Hybrid Navigation	1
c. 4-D Flight Path Control	4
d. Heads Up Display	4
C. 1990's Aircraft	
(2) Stability and Control	
A. Current Aircraft	
a. Autopilot: Yaw Damper	1
B. 1980's Aircraft	
a. Cockpit Displays - Electronic Attitude Direction Indicator(EADI)	3
b. Cockpit Displays - Electronic Horizontal Situation Indicator (EHSI)	3
c. Flight Envelope Limiter	3
d. Ride Improvement System	1

TABLE 3.2(continued)

C. 1990's Aircraft	
a. Full Scale Stability Augmentation System (Hard SAS)	4
b. Gust Load Alleviation	4
c. Flutter Mode Control	4
d. Maneuver Load Control	3
(3) Air Traffic Control	
A. Current Aircraft	
a. Weather Radar	1
b. Ground Proximity Warning System (GPWS)	3
B. 1980's Aircraft	
a. Digital Weather Radar Processing	1
C. 1990's Aircraft	
a. Digital Data Link (including Metering and Spacing Function)	3
b. Airborne Traffic Situation Display (ATSD)	3
c. Collision Avoidance System (CAS)	3
(4) Aircraft Systems Management	
A. Current Aircraft	
a. Fuel Control (Tankage)	1
b. Fuel Control (Engines)	3
c. Engine Inlet Control	1
d. Center of Gravity Indicator (Pre-flight)	0
e. Weight Monitor (Pre-flight)	0
f. Cabin Environment	1
g. Air Data System (Temperature, Pressure)	3
h. Master Caution System	3
i. Fire Warning System	3
j. Power Generation and Distribution	3
k. Automatic Braking System	3
l. Aircraft Integrated Data System (AIDS - Maintenance only)	1
B. 1980's Aircraft	
C. 1990's Aircraft	
a. Integrated Data Management System	1

aircraft (1990's). Once introduced, a function is expected to continue into later aircraft designs.

The definitions of criticality are as follows:

0. Off (function either unused or unimplemented).

1. Non-essential.

2. Results in reduced operative performance and increased pilot workload.

3. Will become potential safety hazard if unattended.

4. Crucial: instantaneous safety hazard.

Table 3.2 is essentially a complementary addendum to the Table 2 of "Design of a Fault-Tolerant Airborne Digital Computer," Vol. II by Ratner et al. [4]. Whereas Ratner defined the concept of criticality of classes, Table3.2 attempts to expand by more closely defining the impact of function failure on the pilot and by adding the dimension of time or flight phase to the criticality of a function.

Not all functions are active during all phases of a flight and are not equally critical during all phases of the flight. Since all functions are not performed simultaneously, the processing requirements of the computer are computed by summing the various processing requirements within the individual mission phases and selecting the mission phase with the highest requirement. This should be done for the full system and should also be done when dropping the less critical tasks. The resulting figures would indicate the total processing power required for the system to perform all tasks, and to what extent failures can impact this basic performance before critical tasks are affected. Memory requirements for the various applications should be considered differently, as all programs for all phases must be carried at all times even when the programs are not being executed. When failures occur which reduce memory capacity, noncritical programs and data can be deleted. Once a program is deleted, however, it can not be recovered during a later flight phase.

Of the more likely functions expected on the next generation of aircraft, the autoland function tends to dominate many of the requirements. Autoland tasks are the most critical, as there is virtually no time to recover from system failures. Additionally the computational loads involved are at a peak. Autoland program memory requirements are

also large. Some relief is available from the shortness of the autoland phase, which greatly reduces the system exposure to autoland failure. It is in fact this very brief exposure period which allows certification of the existing triplex and deal-dual analog autoland functions. Hard stability augmentation, gust load alleviation and flutter suppression, if and when implemented, will place considerably increased reliability constraints on the computer because of the almost continual exposure to critical function loss. Interestingly enough, most of these prolonged exposure functions, while considerably boosting reliability requirements, make little impact on performance requirements. This is because they occur in non-autoland phases of flight, where calculations are not so complex as the autoland calculations. A notable exception is flutter supression.

The integrated system will be partitioned into a highly reliable central multiprocessor(s) and local processing elements. Local processors are located at and service aircraft subsystems. They format the data required by the integrated system as a whole and communicate it back to the central multiprocessor. They also take direction or commands from the central multiprocessor. Required computations are partitioned between these local elements and the central site. At one extreme the local processor is used as a simple data formatter with all computation being done locally. At the other extreme most of the computation is done locally with the central site merely serving as a message switching site to allow the exchange of data among the various subsystems. Either extreme is awkward, and the actual partitioning will be somewhere between the extremes. Factors which will affect this partitioning are (a) criticality of the computation: since the central site is ultra-reliable an ultra-critical task is likely to be done there; less critical tasks can be done locally. (b) Globalness of the task: computation which uses data available only globally, that is from many subsystems, tends to be centrally sited; computation which uses inputs and controls outputs all of which are locally available are good candidates for local processing. Engine control is a good example of this; failure of a single engine controller is not critical and most of the control can be done locally with only occasional changes in thrust set points from the central site being required. (c) Bandwidth reduction: if preprocessing of sensor data locally reduces the amount of information being shipped to the central site, then the overall I/O system can be built with a lower bandwidth. (d) Political/Engineering considerations: certain control functions are likely to be partitioned

according to manufacturer. Thus the engine manufacturer will provide an engine controller, the radar manufacturer a signal processor, and the displays manufacturer a display processor. This will almost certainly continue to be the case until the reliability of a central multiprocessor and its I/O system can be demonstrated. Even then, the partitioning makes a great deal of sense in many cases, as it allows for the autonomous operation of subsystems. This may possibly be unimportant in the context of an integrated system, but will certainly be beneficial for the manufacturing, repair or testing of the subsystems away from the aircraft. Partitioning at this time remains fairly undefined pending the exact definition of the aircraft/function implemented. Certain isolated cases such as engine control or display processing will almost certainly be local while others such as air data computation will be central, but partitioning of the remaining functions will await greater detail of the specifics of a particular situation.

A major concern in advanced commercial aircraft design is fuel conservation. A study by Lockheed summarized in Figures3.1 and3.2 showed that active control technology applied to an L-1011 type aircraft could result in 4.4 per cent savings in fuel. To achieve this, maneuver load control is utilized allowing a lighter main wing, and stability augmentation is used, allowing reduction of the size of the tail surfaces. The aircraft is statically stable in most areas of the flight envelope, although stability margins are reduced.

For a single function such as maneuver load control, a triplex or quadruplex ($[\text{Fail-op}]^2$) computer may turn out to be sufficient. The suitability of such a simple approach is aided by two factors. First, exposure to computer failures is very small. Maneuver load control is only critical during high-g maneuvers. While it is true that a transport must be certified for high-g maneuvers, the risk due to such maneuvers is computed as the product of the likelihood of having to make such a move and the likelihood of the computer failing to perform adequately during the maneuver. Much of the maneuver load control is designed to increase the fatigue life of a weaker wing. Since a single failure can have only a small impact on fatigue life, maneuver load control is not generally critical. Secondly, a single system such as this is small, and it is almost always more economic to solve a small problem in the particular than to develop a more global solution.

The building of multiple, simple dedicated boxes for each such function is not likely to produce a system which is simple and economic.

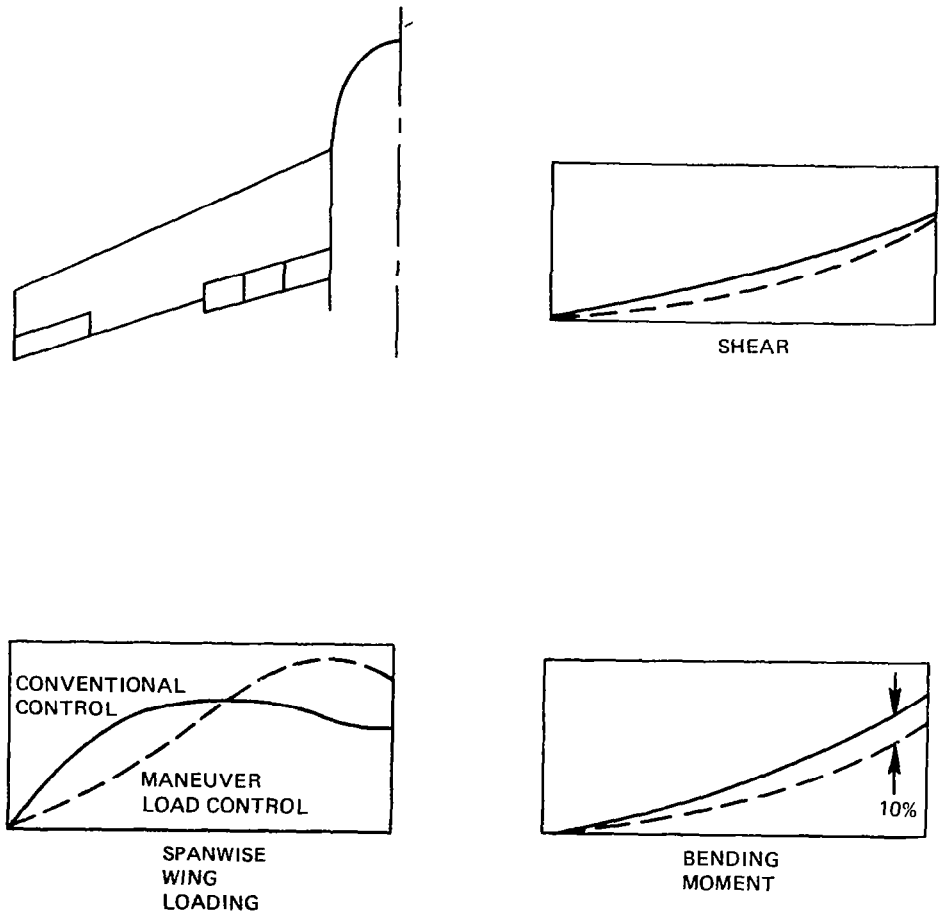


Fig. 3.1 Potential Impact on Design.
(Courtesy of Lockheed California Company)

L-1011		FUEL SAVINGS, % (DUE WEIGHT AND DRAG)
• REDUCE HORIZONTAL TAIL SIZE BY 350 FT^2		2-1/2
• REDUCE VERTICAL TAIL SIZE BY 220 FT^2		1-1/2
• ADD 19,000 LB TAKEOFF WEIGHT WITH NO STRUCTURAL CHANGE TO WING		—
	TOTAL	4%

NOTE:

MAJOR DEVELOPMENT COST TO MODIFY EXISTING AIRCRAFT; REQUIRES INCREASED STABILIZER TRAVEL, ALL MOVING VERTICAL, AND RELOCATION OF LANDING GEAR

Fig. 3.2 Potential Impact on Design.
(Courtesy of Lockheed California Company)

Indeed the opposite is true. A properly designed integrated system offers the potential for considerable efficiency improvements over a multitude of dedicated solutions. Reliability constraints tend to be greater for the integrated system,though. Where previously many failures could be projected to occur at non-critical moments for the affected subsystems, an integrated system is by its nature much more exposed to failures. A single failure within an integrated system is likely to affect many subsystems. Where the failure of one might not be critical, the failure of several together can have a aggregate affect which is critical. Additionally, the aircraft is sensitive to failure in the integrated system whenever it is critically sensitive to the failure of at least one of its subsystems. Because the reliability constraints for such an integrated system are very high and because such systems will be fairly complex, there is currently a tendency by industry to back away from fully integrated systems. Highly critical functions are unloaded from the integrated system and placed in dedicated small independent subsystems. This is a viable approach as long as many of the benefits of an integrated system can be achieved and the reliability requirement of the integrated can be kept moderate by unloading onto only a small number of independent subsystems. Many of the advanced design concepts for flight control will not allow spin off of the critical calculations from the integrated systems. It should be expected, therefore, that initial use of the fault-tolerant multiprocessor will be in reduced criticality situations. As the pressure to use more advanced flight control builds, and when the integrated system has demonstrated the required reliability it will be used in increasingly critical applications.

3.2 Maintenance Considerations

On-board computing capability has a direct effect on maintenance costs of commercial aircraft. The ability to perform in-flight monitoring with assured detection and isolation of errors to the line replaceable unit (LRU) of the multiprocessor should eliminate unnecessary removals. Additionally, the ability of the multiprocessor to monitor the performance of, and test attached equipment should greatly reduce unnecessary removals of those units and cut the diagnostic cost of locating faults. Industry estimates are that 50% of all electronic LRU removals are unnecessary, at an estimated cost to the airlines of about $100 million per year.

In addition to aiding the maintenance precedure, the fault-tolerant nature of the computer allows spare modules to be included as part of the system. This enables maintenance to be deferred until a maintenance center is reached. If sufficient spares are carried on board it will be unnecessary to stock spares at field depots, and maintenance can be concentrated at a main center. Resulting savings in logistics of scheduling and numbers of stocked spares should be considerable.

Certain recommendations can be made regarding the fault-tolerant multiprocessor.

a. Sufficient spare capacity should be carried to assure that maintenance can be postponed for at least 50 hours for at least 99 per cent of the maintenance events.

b. Maintenance of the computer should be centralized, possibly becoming the responsibility of the manufacturer.

Wherever maintenance can be postponed by 50 hours, it is assured that the aircraft will pass through the main maintenance base of an airline before a maintenance action is required. This allows a single site for the airline to be chosen as the airline maintenance and spare stockage site. Since most maintenance can be done centrally, and because quality control can be held tighter for a single site than for many, it will be advantageous to centralize maintenance. A logical extreme of this might be for the manufacturer to maintain service centers that maintain these computers for several airlines.

3.3 Operational Considerations

An integrated highly reliable aircraft information processing system may impact airline operations in several areas. The overall system should have increased reliability when compared to todays systems, offering improved dispatch reliability. Improved maintenance and deferred maintenance will also eliminate many delays. A dependable autoland capability will reduce weather delays. Data logging and equipment history recording, using the excess or spare compuational capacity of the system, will improve overall aircraft maintenance, and may eliminate the need for the flight engineer. More reliable avionics will mean more extensive use of automatic flight modes eliminating the potential for many pilot errors. Autoland, for example, is expected to be several orders of magnitude more reliable than manual landing in clear weather. Automated checklist and automatic contingency programming

further reduce the possibility of crew error. Optimal flight profiles with optimal engine control can be performed using data and control available in the integrated system. While many of these tasks are not critical, the acceptability of automated solutions to these tasks is hinged upon the economy and continuous availability of the computer system.

3.4 Industry Visits

To insure a realistic assessment of the C.S. Draper Laboratory concept for a fault-tolerant multiprocessor, staff members of the MIT Flight Transportation Laboratory and the C.S. Draper Laboratory undertook a series of visits to selected airlines, airframe manufacturers, and avionic equipment manufacturers.

Airlines visited were Eastern Air Lines and Pan American World Airways. Airframe manufacturers visited were Lockheed Aircraft Corporation (Lockheed-California Co., Burbank, California) and McDonnell Douglas Corporation (Douglas Aircraft Co., Long Beach, California). Avionics manufacturers included Bendix Corporation (Flight Systems Div., Teterboro, N.J.), Rockwell International Corporation (Collins Radio Group, Avionics Div., Cedar Rapids, Iowa) and Sperry Rand Corporation (Sperry Flight Systems, Phoenix, Arizona).

At the airlines the specific groups visited were responsible for avionics selection and operational maintenance. As might be expected, Pan American with world-wide routes and maintenance, differed substantially from Eastern, with its tight short haul structure, as to operation and maintenance philosophy. Pan American indicated a strong desire to bring as much of their maintenance operations as possible back to their main base in New York. In general, maintenance that is deferrable for 50 hours can be done in New York. Maintenance which can be deferred for 300 hours would be done at regularly scheduled maintenance periods. Eastern, in contrast, showed little interest in deferring avionics maintenance for long periods of time. Much of the complex avionics, such as inertial units, are not maintained by Pan American, but by the manufacturer. This is in line with this report's recommendation as to the best maintenance procedure for the fault-tolerant multiprocessor. Eastern, instead, does all of its own maintenance. Eastern's self reliance might result from a lack of many of the complex subsystems found on Pan Am jets. Eastern planes do not have inertial navigation systems, for example. The tendency is probably

toward manufacturer's responsibility for maintenance, particularly for systems requiring expensive test equipment and highly skilled technicians. Once the investment is made for such a facility it is likely that it could handle the entire fleet of a particular type of aircraft. A single airline may not find it economical to service such equipment alone. A possible alternative is for several airlines to form cooperatives to service equipment, or for one airline to make the investment and service smaller airlines for a fee. Ample examples exist in Europe of cooperatives for pooled aircraft maintenance, and of airlines performing services for one another for fee in this country.

Beyond maintenance neither Eastern nor Pan Am felt a great need for additional functions. Both expressed a desire for more reliable operation with existing functions. This reflects a general feeling that for a function to be widely accepted and used, it must be very nearly continuously available. Thus a good argument can be made for fault tolerance, even for relatively non-critical tasks.

Both of the airframe manufacturers visited indicated that the next generation or the next derivative aircraft would use digital avionics. Some advantages are expected from reduced static stability and maneuver load control, although such steps toward a control-configured vehicle will be small and cautious. The flight control system will be critical only during autoland and during high speed cruise where the yaw damper may be vital. Fall-back control systems will allow manual control under most conditions of the aircraft even if the flight control system fails. Some concern was expressed that if the computer did become as vital as some of the more advanced concepts suggested, that at least two computer sites would be required so that physical damage to one of the sites would not result in the loss of the aircraft. The manufacturers foresaw no problems with FAA certification of functions based upon digital technology, although they noted that software modularity could ease the certification problem.

The visits to avionic equipment manufacturers reinforced the belief that the next generation of aircraft would have a higher complement of digital computation, but they did not foresee a great deal of integration of functions. Avionic manufacturers expressed some concern that non-dispatch-critical functions would be integrated into a central computer, even though the concept of the fault-tolerant computer allows for failures of these functions. They also felt that certain functions would continue to be performed in analog, rather than digital, fashion,

particularly the yaw damper. Avionics manufacturers agreed that power supply failures and transients, as well as connectors, caused the greatest amount of problems for avionics equipment on current generation of aircraft. Again, FAA certification of new digital functions was not foreseen as a problem and could be accomplished through simulation.

3.5 Summary

The greatest advantage of a fault-tolerant multiprocessor lies in the area of safety: the slow degradation of computing capability while maintaining high reliability for flight critical functions. To insure the survivability of the system in case of non-catastrophic physical damage to the aircraft, the central computer must not be confined to a single physical location.

Some additional problems (which are not unique to fault-tolerant multiprocessors) that must be considered in the implementation of the system are power supplies (both transients and failures) and connectors. The above two areas cause the great majority of failures (or reported failures) in current avionics equipment. Sensor and actuator reliability will also become more critical in the next generation aircraft. Finally, the central computer must be resistant to failures induced by lightning strikes on the aircraft.

In the software area, it is not anticipated that FAA certification of software for the fault tolerant multiprocessor will be significantly more difficult than any other software certification for avionics equipment. Modularity of software, to the degree possible, will probably ease the process. The computer language does not have to be decided on the basis of higher order language or machine language: rather the most appropriate mix from a programming point of view should be used. Certification will not be more complicated in any case.

In summary, no major objections or constraints to the use of a fault-tolerant multiprocessor in airline operations exist, while the concept does offer several significant advantages.

CHAPTER 4

MALFUNCTION TOLERANCE

The usually high level of dependability required in the central computer makes it mandatory to consider all possible sources and effects of probable malfunctions. The probabilities associated with exposure to hazards are important here, as they are in any reliability analysis. The fact that reconfiguration and recovery are needed to meet reliability goals raises other issues of importance, having to do with the probabilities associated with the detection and identification of malfunctions, reconfiguration and recovery of the system, and the system status following a malfunction event. All these considerations relate both to the design and the evaluation of the system.

To give some structure to this subject we first define the malfunction sources. Certain of these malfunction sources, external to the system, are able to be dealt with by isolation of the computer in the form of malfunction prevention. Where prevention measures are inadequate, malfunction sources must be dealt with by tolerance, commonly called fault tolerance. Both prevention and tolerance pose architectural constaints as well as design requirements extending to the total aircraft system. The purpose of this chapter is to introduce the reader to the nature of malfunction sources and the ways in which they impact the system architecture and design.

4.1 Malfunction Sources

A malfunction is a general term for anominal behavior. Numerous kinds of malfunctions are distinguished, ranging from microscopic disorders in an integrated circuit to total aircraft impairment. Within the information processing segment of the total system, we are concerned about avoiding malfunctions that preclude the availability of viable contingencies. We can think of potential malfunctions as being infinitely rich in number and variety, and tractable solely because they can be treated as classes and subclasses.

The first class of malfunctions to be examined is that resulting from externally induced phenomena, such as physical penetration, radiation (atomic, electromagnetic), temperature extremes, or excursion of prime power. The common thread in these diverse physical environments is that their effects can not be confined or localized to one or a few sub-portions of the information system. The entire system is vulnerable at one time, and for an arbitrarily high exposure it can not be made otherwise. That is, the shielding, structure, environmental control, and prime power generation must all be designed to withstand stated levels of exposure to known hazards. Exposures in excess of these levels are potentially catastrophic.

Induced malfunctions can not be dealt with in absolute terms by internal redundancy, because of their high correlation across separate elements of the system. Nevertheless, not all overexposures will defeat a redundant system, and the recovery possibilities of the redundancies should not be ignored. Other than this, the induced malfunction problem is treated as a somewhat separate, external issue from the system architecture. One exception to this last statement is in the input-output linkages from the central computer to local computers, and thence to the sensor and effector subsystems. The potential exposures in these linkages to radiation and damage are particularly high, and warrant careful consideration of linkage technology and topology.

The second malfunction class is of malfunctions whose sources are internal to the system. It is customary to subdivide this class into two subclasses corresponding to random and systematic occurrences. Actually, however, it is more nearly true that a continuum exists between the two extremes, which needs to be characterized by simple and joint probability distributions. For our present purposes of general description, it is reasonably accurate to use the concept of a correlation coefficient together with the assumption of random malfunction events with a constant hazard rate. For independent malfunctions within a single element protected by other, redundant, elements, the correlation is taken to be zero. For malfunctions affecting two or more redundant elements identically, such as design errors, or pattern-sensitive anomalies, the correlation is taken to be one. In this case, the redundant group behaves exactly as a simplex element. In the case of internal systematic problems, such as crosstalk or generic hardware weakness, the correlation is intermediate between zero and one. As a

general rule, the redundant system failure rate is linearly sensitive to correlation parameters, whereas it is sensitive to some power, typically the second, of the hazard rates.

The third class of malfunction sources will simply be denoted as "other sources." The first two classes are broadly enough defined to be stretched to cover everything, but it is useful to emphasize certain sources separately. Thus we include in this third category the deficiencies resulting from lapses in system specification, that is, where the domain of operation and the domain of design are not matched. Software in this sense is a specification. It specifies the sequential rules of hardware utilization. Logic design is also a specification in this sense, as are design factors related to the human interfaces and the sensor and effector interfaces.

The architectural implications of this category are that the system must be tractable and understandable enough to reduce the probability of occurrence of such malfunctions to a negligible level.

4.2 Malfunction Consequences

It has been useful to characterize the various possible malfunctions according to the levels at which they affect the system. There are physical malfunctions that occur within hardware elements, such as a short circuit in a transistor. These have been referred to by various writers as faults and failures, and in this report the word failure has been intended to refer to this category. A physical malfunction may or may not result in a logic malfunction, in which a logic variable is at some time or another complementary to its correct value. Where authors use the word "fault" for physical malfunction, they use "failure" for logic malfunction, and vice versa. A logic malfunction can occur in the absence of a physical malfunction, notably from induced sources.

A logic malfunction may or may not produce a data malfunction, often called an error. A data malfunction can occur in the absence of a logic malfunction, notably from specification lapses. A data malfunction, in turn, may or may not produce a subsystem malfunction, which in turn may or may not produce system malfunction.

We have portrayed a propagation chain from physical malfunctions to system malfunction, with some external entry points. Whether propagation takes place from one level to another depends on whether a

causal link exists in the first place, and whether the phenomenon is masked by a redundancy. Thus a logic malfunction produces a data malfunction only if it impacts the outcome of an operation. Even then, it may not, as for example when the data results from the voting of three inputs, only one of which suffers a data malfunction.

A key point, often overlooked in simplistic treatments of redundancy, is that redundancy always has a limited capacity to mask malfunctions, and this capacity can degrade to zero without affecting the apparent behavior of the system. Therefore, a system designed to have tolerance may in fact have none at the inception of a critical mission. Alternatively it may have some tolerance, but less than the design level, and less than what is assumed. The preceding chapter made this point in the discussion of detection, identification and recovery. The system reliability impact of this issue is treated in the next section. It is worth reiterating here that masking is a two-edged sword. On one hand it is a mechanism for holding malfunctions at a low system level, while on the other hand it may obscure the fact that the malfunction has occurred and thereby has reduced the system's tolerance to future malfunctions.

4.3 Tolerance Renewal

The primary advantage of hybrid redundancy over TMR is that injured triads are reconfigured back to a state where they can once again mask malfunctions. This is a process of tolerance renewal. In principle, the system failure rate is restored it its design value by the reconfiguration process. If reconfiguration were to fail, the system failure rate would increase, possibly by many orders of magnitude.

In practice, there are several ways in which an injured triad can fail to be reconfigured. These include exhaustion of spare modules, malfunction of the reconfiguration mechanism, failure to detect the need to reconfigure, and perhaps the use of a defective spare module. We can characterize the process of tolerance renewal as the detection and location of any physical malfunction, the removal of vulnerability from the triad containing the malfunction, the replacement, by spares, of functions thus removed, and the initialization of the reconstituted triad. All mechanisms involved in this process are subject to malfunction, of course, and such malfunctions constitute injury to their triads, and require that tolerance renewal be carried out.

The idealized renewal of tolerance, together with a sufficient complement of spares, has an interesting theoretical consequence if the hazard rate is constant. Figure 4.1 illustrates this concept. The system failure probability increases monotonically with time in the absence of tolerance renewal. At any arbitrary point in time, if the system is tested and found to be perfect, the reliability becomes equal to one. If the test reveals an injury, then the system is reconfigured to a state of virtual perfection and the reliability is likewise equal to one. This paradoxical result depends on the absence of any deterioration of the renewal mechanism, however. The concept's utility is to suggest to the reader that the dynamics of redundancy management are all-important in maintaining the system reliability at a level that is unreachable by non-redundant elements.

The actual system behavior differs in several respects from this concept. First, the supply of spares is not inexhaustable. Second the tolerance renewal mechanism is subject to degradation. Finally, the ability to test the system is limited by its probabilistic nature. These three items will be briefly discussed in the paragraphs following.

The supply of spares is a degree of freedom available to the system designer. In the parallel-hybrid redundant multiprocessor, there can be arbitrary numbers of processors, memories, interface access modules, memory bus lines, interface bus lines, oscillators, and power converters. If the failure rates of these elements are known, and if the minimum numbers of each needed for system survival are known, the probability of exhaustion of spares as a function of time can be calculated using conventional combinatorial analysis.

The tolerance renewal mechanism in the parallel-hybrid redundant multiprocessor is largely contained in the voters and the bus guardian units. Both the voters and the guardian units possess bus line interfaces, and therefore are both capable of degrading elements (i.e. bus lines) outside of their own modules (e.g. processor, memory, interface access). This by itself is not qualitatively different from a single malfunction. The important concern is that all guardians in a single module may fail in such a way as to enable that module to transmit on more than one bus line. As mentioned in the preceding chapter, design steps are taken to minimize the probability of this eventuality, but the probability is finite that it will happen. A subsequent failure of the module in a malevolent state could cause an entire central computer to malfunction.

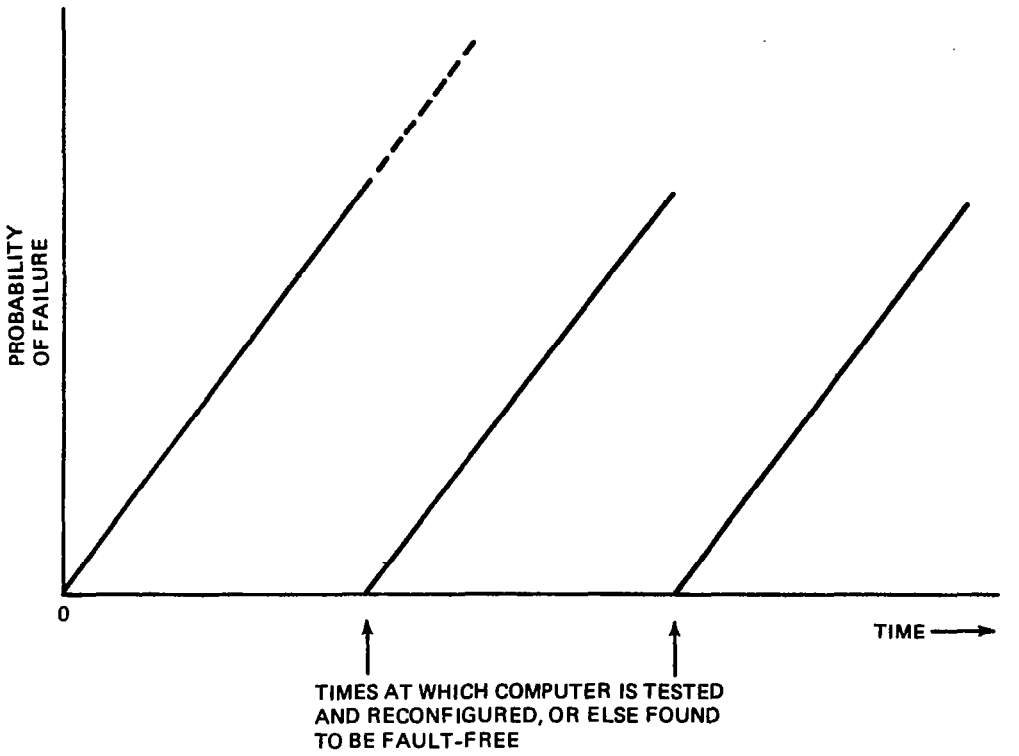


Figure 4.1. Idealized Tolerance Renewal.

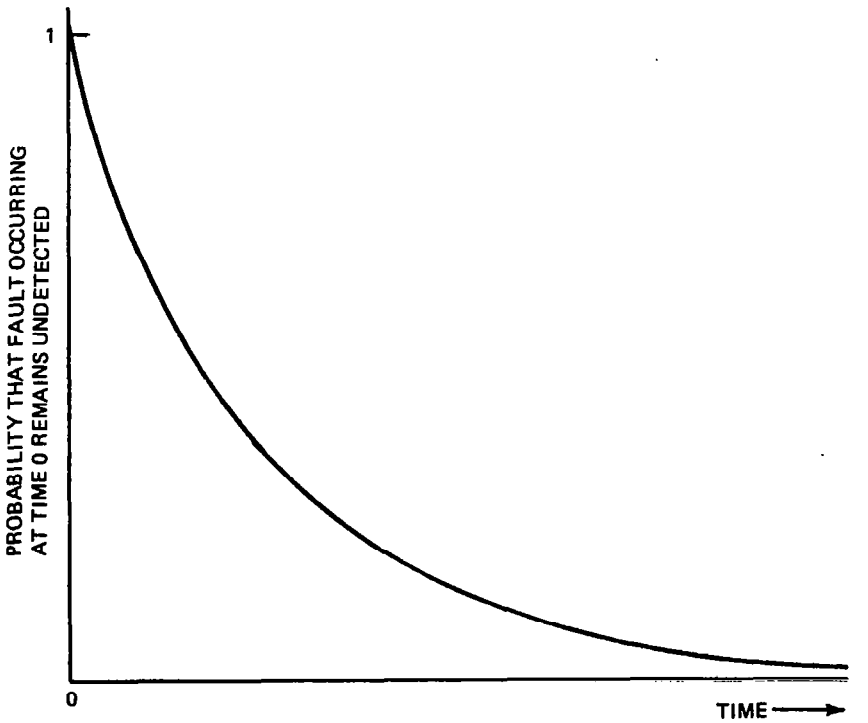


Figure 4.2. Fault Latency Model.

Finally, we deal with the problem of detecting malfunctions when they occur. If a malfunction results in a bus data malfunction, it can be detected by the voters. If not, it is termed "latent." Latent malfunctions must be considered at least as harmful as visible ones. It is for this reason that each critical element must be subjected to periodic test while the system is on line. The preceding chapter discussed the matter of periodic testing for some classes of elements. As an approximation to the testing process we assume that the detection of malfunctions is an exponentially distributed random process with a constant rate of discovery. Figure 4.2 illustrates the nature of the distribution. According to this model, of all the possible malfunctions that will occur, most of them will be detected after a lapse of several time constants, but some will never be detected. The parameter ρ represents the rate of discovery. The system malfunction probability that results from finite malfunction detection time can be modeled as a Markov process with constant hazard rates and constant discovery rates.

4.4 Reliability and Maintenance

The multiprocessor's design approach to system reliability consists of a combination of shielding, environmental control, redundancy, reconfiguration, test algorithms, voting, and high reliability design and manufacture of all hardware elements. In addition, the system software must be virtually perfect. To a certain extent, these facets of system reliability are synergistic. That is, once the central computer attains a certain degree of reliability, it becomes competent to serve as its own manager, and thereby becomes capable of attaining even greater survivability by judicious utilization of resources.

Prior to each flight, the multiprocessor will test itself and the rest of the information system to purge it of latent malfunctions and establish accurately the degree of tolerance remaining. If this is adequate for dispatch, the flight will proceed, maintaining high reliability by the tolerance renewal procedure, including frequent testing of every element. In this way, a log is maintained of the status of every element of the system. Intermittent, transient, and permanent malfunctions can be distinguished, and the momentary tolerance made known to the flight crew. If changes in flight plan or envelope are called for, these can likewise be made known.

Upon landing, the need for maintenance, if any, of the information system will be readily discernable,including in most cases the identities

of the elements that have malfunctioned. A possible byproduct of system fault tolerance is the realization of considerable operational cost saving by postponing maintenance until the aircraft arrives at a base where this is most economically accomplished. If the probability of needing a module change earlier than, say, one hundred flight hours can be held below one per cent, it could be well worth the inclusion of one or two extra spares to make this possible.

CHAPTER 5

A RELIABILITY AND AVAILABILITY STUDY OF THE FAULT-TOLERANT MULTIPROCESSOR

The objectives of this study may be summarized as follows:

1. Assess the availability/reliability of the baseline multiprocessor configuration with variations.
2. Evaluate existing computer programs or develop alternate programs towards this goal.
3. Evaluate the results.

The following sections deal with each objective in detail.

5.1 Evaluation of Existing Reliability Programs

Two general reliability assessment programs, Care II and Tasra, were evaluated to determine their applicability to the present task. An analysis of these programs resulted in the following conclusions.

The first program, Care II, does not lend itself well to modeling parallel hybrid redundancy. It would have to be modified considerably to cover this type of system architecture. The program does take into account degradation in the system reliability due to a lack of coverage but the modeling of coverage is subjective.

The second program, Tasra, is a combinatorial reliability model. It does not take into account coverage, or undetected failures.

Since it is preferable to compare different system candidates on a common ground, a good deal of thought was given to ways in which we might be able to conform these existing programs to our reliablity prediction task. It is our considered opinion that to do so would result in either a distortion of these programs to a point where they are no longer standard in a useful sense, or a highly erroneous reliability model. As an alternative, we have observed that the Markov process model, although not standard in the programmatic sense, is a widely used and well recognized mathematical tool which can model the

parallel-hybrid system with high fidelity and relative simplicity.

Based on these considerations, a Markov process model of the multiprocessor system was developed to predict the system failure probability due to a lack of perfect coverage. The next section describes the model in detail.

5.2 First Markov Process Model of the Multiprocessor

A system is said to be governed by a Markov process if the next state of the system depends only on its present state, and is completely independent of the past history of the system states. Such a system can be accurately represented by a Markov model.

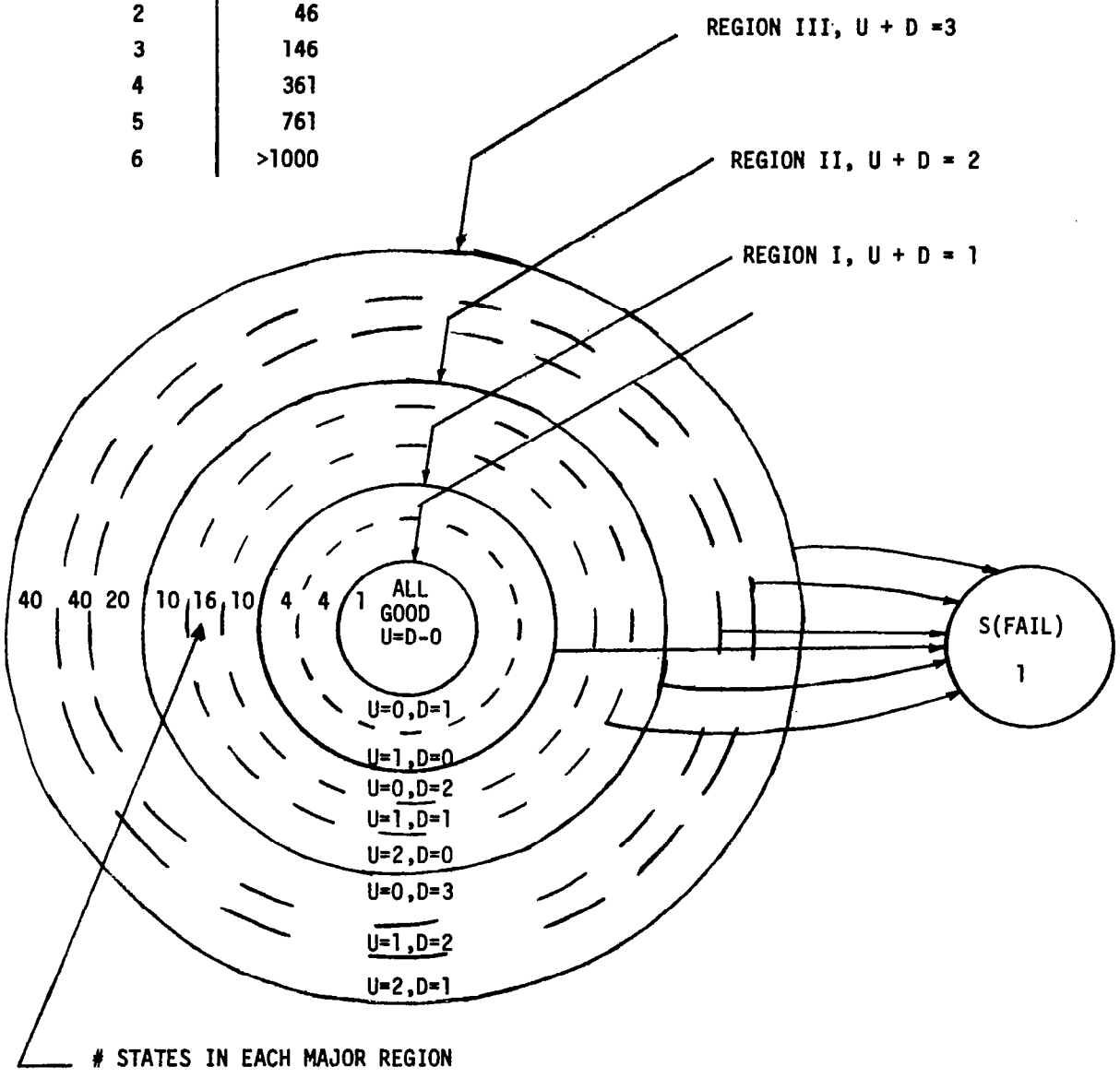
In the present case, the computer system starts out in a state where all units are functioning correctly. Each occurrence of a fault moves the system to a new unique state. Similarly, each fault detection (followed by reconfiguration of the system) also results in a new state. Since all of these state transitions can be defined independent of the past history of the system, it follows that such a system can be represented by a Markov model.

Figure 5.1 shows the number of states in a general Markov model of the multiprocessor. The system states are divided into major groups or regions. Each such region is identified by the total number of component failures. The following four component types are distinguished: processor, memory, bus and the bus guardian unit (BGU). Each major region is further subdivided into a number of minor regions. A minor region is identified by

1. the number of undetected failures U, and
2. the number of detected failures D.

An undetected failure implies that either the failure is not known (a latent failure) or that the failure has been identified but the recovery process has not been completed yet. In any case, the major point of this diagram is to show the relationship between the number of faults and the number of states in the Markov model. To solve the model numerically it is necessary to truncate the model to a finite number of states. Also, in order to obtain the solution efficiently it is necessary to minimize the number of states commensurate with the required accuracy of the results. The model was truncated to 146 states based on the following reasons. The probability of the

# FAULTS, U+D	# STATES IN MARKOV MODEL
2	46
3	146
4	361
5	761
6	>1000



U = UNDETECTED FAULTS
D = DETECTED & REPAIRED FAULTS

Figure 5.1 The Number of States in a General Markov Model.

system being in a state outside the truncated model after 100 hours is only 10^{-12} (assuming there are 100 units each with a failure rate of 10^{-4} per hour). Since any truncation of the model increases the estimated system failure probability, it is a conservative approximation.

Details of the 146-state model of the multiprocessor are shown in Figures 5.2 and 5.3. Figure 5.2 shows a global view of the system states and transitions between states, while Figure 5.3 is a section of the state transition diagram.

The states in the Markov model are distinguished by the following 8-tuple vector

$$S\ (P_U,\ P_D,\ M_U,\ M_D,\ B_U,\ B_D,\ G_U,\ G_D).$$

The eight elements of the vector show the number of undetected and detected failures in the processor (P), memory (M), bus (B) and BGU (G) units, respectively. Three types of state transitions may take place. The first type is from one major region to another (see Fig. 5.1).

Type I Transition:
$$U(t_1) = U(t_0) + 1$$
$$D(t_1) = D(t_0) - 1$$

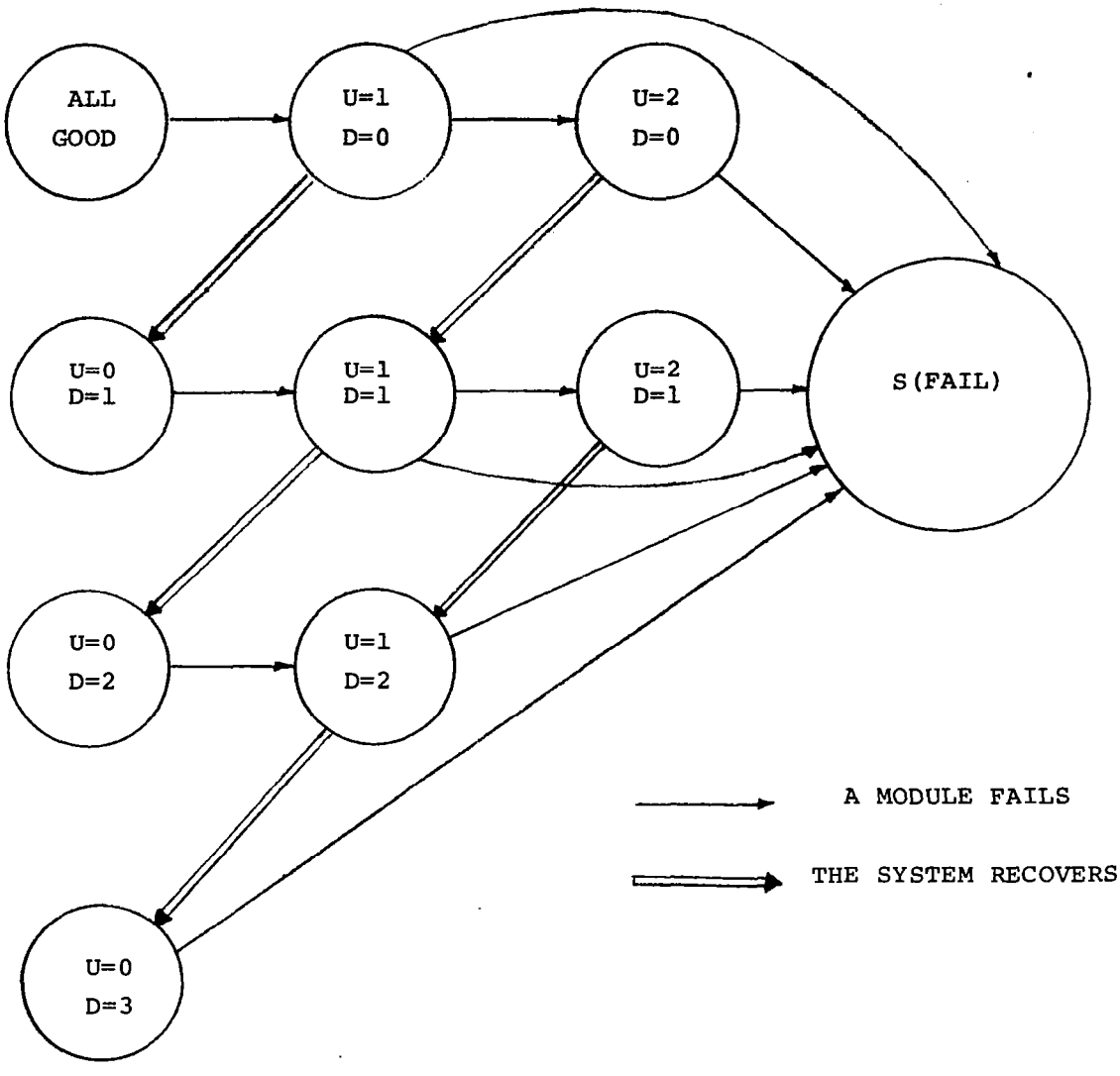
This transition takes place when a unit fails. Each failure is initially undetected. The time for detection and recovery depends upon a number of factors including the type of failed unit, test cycle frequency etc. This type of transition increases the number of undetected failures by one. Of course the total number of failures also increases by one.

Type II Transition:
$$U(t_1) = U(t_0) - 1$$
$$D(t_1) = D(t_0) + 1$$

This transition takes place from one minor region to another. It occurs when the system repairs itself, that is, identifies a failure and recovers from it. Therefore the number of undetected failures decreases by one, the number of detected failures increases by one, while the total number of failures remains unchanged.

The last type of transition takes place when a unit failure results in a catastrophic failure that is, the system fails.

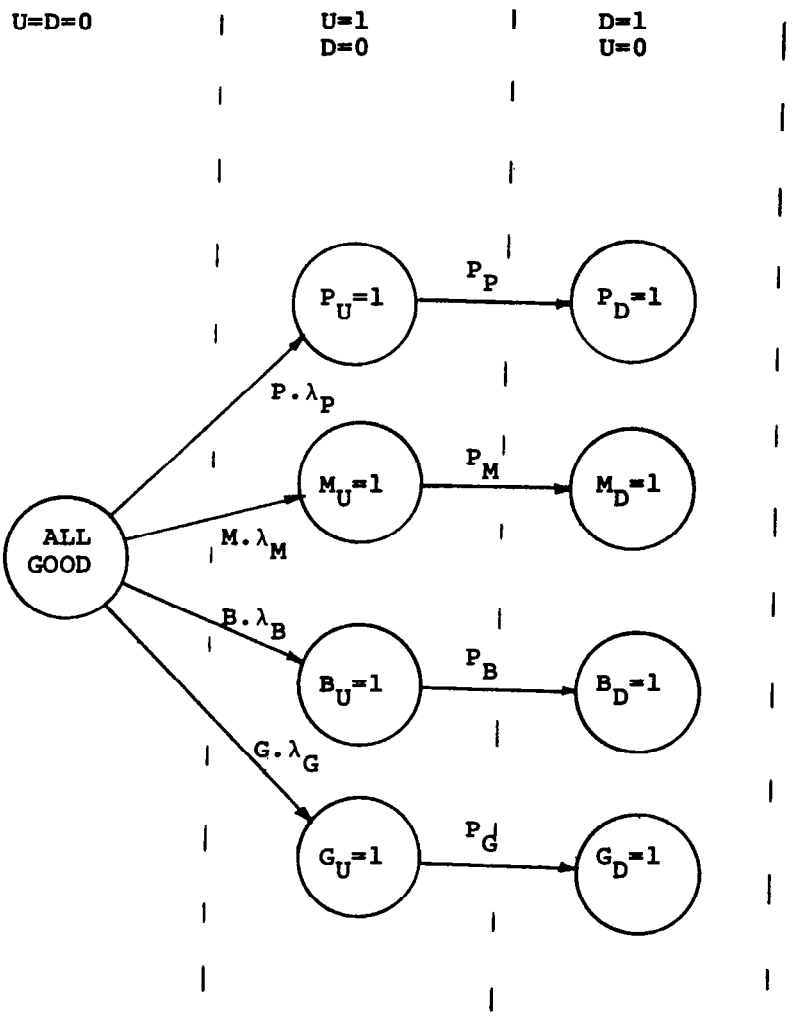
Tables 5.1 and 5.2 show the transition rates for the model described above. The following definitions and assumptions were made to arrive at the transition rates.



U= # undedected failures

D= # detected failures

Figure 5.2 The Markov Model.



(The 8-Tuple State Vector $S(P_U, M_U, B_U, G_U, P_D, M_D, B_D, G_D)$ Uniquely Defines Each State of the Model.)

Figure 5.3 A Section of the Transition Diagram.

TABLE 5.1

FAILURE TRANSITION RATES

U	P_U	M_U	B_U	G_U	$P_U = P_U + 1$	$M_U = M_U + 1$	$B_U = B_U + 1$	$G_U = G_U + 1$	S(FAIL)
0	0	0	0	0	$(P-P_D-P_D^*)\lambda_P$	$(M-M_D-M_D^*)\lambda_M$	$(B-B_D)\ \lambda_B$	$2(P-P_D-P_D^*$ $+M-M_D-M_D^*)\lambda_G$	0
1	1	0	0	0	$(P-P_D-P_D^*-3)\lambda_P$	$(M-M_D-M_D^*)\lambda_M$	$(B-B_D-2)\lambda_B$	$2(P-P_D-P_D^*$ $+M-M_D-M_D^*$ $-3)\lambda_G$	$2\lambda_P+2\lambda_B+4\lambda_{G_{DIS.}}$
1	0	1	0	0	$(P-P_D-P_D^*)\lambda_P$	$(M-M_D-M_D^*$ $-3)\lambda_M$	$(B-B_D-2)\lambda_B$	$2(P-P_D-P_D^*$ $+M-M_D-M_D^*$ $-3)\lambda_G$	$2\lambda_M+2\lambda_B+4\lambda_{G_{DIS.}}$
1	0	0	1	0	$(1-\frac{2}{B^*})(P-P_D-P_D^*)$ λ_P	$(1-\frac{2}{B^*})$ $(M-M_D-M_D^*)\lambda_M$	$(B-B_D-1)\lambda_B$ $-\frac{6}{B^*}\lambda_B$	$(2-\frac{4}{B^*})$ $(P-P_D-P_D^*$ $+M-M_D-M_D^*)\lambda_G$	$\frac{3}{B^*}\ 2\lambda B+\frac{2}{3}(P-P_D-P_D^*)\lambda_P$ $+\frac{2}{3}\ (M-M_D-M_D^*)\lambda_M$ $+\frac{4}{3}\ (P-P_D-P_D^*+M-M_D-M_D^*)\lambda_{G_{DIS}}$
1	0	0	0	1	$(P-P_D-P_D^*-\frac{P}{P+M})\lambda_P$	$(M-M_D-M_D^*$ $-\frac{M}{P+M})\lambda_M$	$(B-B_D-2)\lambda_B$	$2(P-P_D-P_D^*$ $+M-M_D-M_D^*)\lambda_G$ $-5\lambda_G$	$4\lambda_{G_{DIS}}+\lambda_{G_{en}}+\frac{P}{P+M}\lambda P$ $+\frac{M}{P+M}\ M+2\lambda_B$
2 ALL COMBINATIONS					0	0	0	0	$(P-P_D-P_D^*-P_U)\lambda_P$ $+(M-M_D-M_D^*-M_U)\lambda_M$ $+(B-B_D-B_U)\lambda_B$ $+2(P-P_D-P_D^*+M-M_D-M_D^*$ $-P_U-M_U)\lambda_G$

$P_D^* = \frac{P}{P+M}\ G_D$

$M_D^* = \frac{M}{P+M}\ G_D$

$B^* = (B-B_D)$ or 3 whichever is greater

TABLE 5.2

RECOVERY TRANSITION RATES

S(INITIAL)	S(FINAL)			
$U > 0$	$P_D = P_D + 1, P_U = P_U - 1$	$M_D = M_D + 1, M_U = M_U - 1$	$B_D = B_D + 1, B_U = B_U - 1$	$G_D = G_D + 1, G_U = G_U - 1$
	$R_P \cdot P_U$	$R_M \cdot M_U$	$R_B \cdot B_U$	$R_G \cdot G_U$

MODEL PARAMETERS

1. Initial Configuration

   P = Number of Processors.

   M = Number of Memory units.

   B = Number of Buses.

   G = Number of BGUs = 2 (P + M)

2. Component Failure Rates

   λ_P = Processor Failure Rate.

   λ_M = Memory Failure Rate.

   λ_B = Bus Failure Rate.

   λ_G = BGU Failure Rate.

3. Recovery Rates

   δ_P = Processor Failure Recovery Rate.

   δ_M = Memory Failure Recovery Rate.

   δ_B = Bus Failure Recovery Rate.

   δ_G = BGU Failure Recovery Rate.

There are a number of state transitions that lead to a catastrophic failure, that is, the system failure. The following combinations of undetected double failures were assumed to cause such transitions.

1. Two processors or memory modules fail in a single triad, or two active buses fail.
2. One BGU fails in the disable mode and a processor or memory unit in the same triad (but with a different BGU) also fails.
3. One active bus fails and a processor or memory unit talking on one of the good active buses fails.
4. One active bus fails and a BGU on a good active bus fails in the disable mode (disabling its associated processor or memory unit).

5. Two BGUs in the same triad but belonging to different units fail in the disable mode.

The following combination of three faults (undetected or detected) causes the system to fail.

Two BGUs in a single processor or memory unit fail in the enable mode and the associated unit also fails.

It was further assumed that all triple undetected faults cause system failure. This assumption simplifies the model a great deal by reducing the number of states in the model. However it does not contribute to errors in any significant way since the probability of having 3 undetected failures is of the order of magnitude 10^{-16} (assuming 100 units with a failure rate of 10^{-4} per hour and a recovery time of 1 sec.). In any case, it is a conservative assumption.

The last assumption, inherent in the truncated Markov model, is that all quadruple faults (detected or otherwise) cause system failure. Based on the above assumptions, the state transition rates may be derived as follows.

The transition rate for type I and III transitions is determined by the product of the unit failure rate and the total number of active units of the type involved in the transition. For example, consider the following transition.

$$S(U = 0) \longrightarrow S\ (U = 1,\ P_U = 1)$$

This transition involves the failure of a processor unit. There are P units of this type. Of these P_D units have failed. In addition, due to the failed BGUs, of which there are G_D, some processors have been disconnected. These can be estimated to be

$$= \frac{P}{P + M} \quad G_D = P_D^*$$

The required transition rate therefore equals $(P - P_D - P_D^*)\ \lambda_P$. Similarly, the number of active BGUs for the transition

$$S\ (U = 0) \longrightarrow S\ (U = 1,\ G_U = 1)$$

equals $2\ (P - P_D - P_D^* + M - M_D - M_D^*)$.

This is twice the sum of the active processor and memory units. When there is an undetected bus fault, the transition rate from this state to the failed system depends on the type of the bus unit involved. If the fault is in one of the spare buses, the system can tolerate another

failure of any type. However, if the fault lies in one of the active buses, a number of second failures could bring the system down. Let us assume that the rate at which such critical faults occur is X per hour. As a first approximation the probability that the failed unit is one of the three active buses is

$$\frac{3}{B - B_D} .$$

Here $B - B_D$ is the total number of buses that could have failed. Therefore the effective critical failure rate or transition rate becomes

$$\frac{3X}{B - B_D} .$$

A computer program was developed to solve the 146-state Markov model numerically. This program, encoded in the PL/I language is listed in Appendix I. The following algorithm was used to solve the model numerically.

$$\underline{P}_{k+1} = \Delta t \cdot \underline{P}_k \cdot Q$$

Here,

$\underline{P}$ = state probability vector (146 X 1)

Q = state transition matrix (146 X 146)

$\underline{P}_k$ = k^{th} state probability vector

$\underline{P}_{k+1}$ = state probability vector after a time-step Δt.

$Q(I,J)$ = transition rate from state I to J

$Q(I,I)$ = computed such that row I sums to unity.

The time-step Δt is chosen such that the fastest transition rate in the system is less than unity. Selection of the time-step is affected by two conflicting factors, the accuracy of the results and the computational speed. Smaller steps result in more accurate results but require more computer time. Figure 5.4 shows the system failure probability due to lack of coverage for a baseline system configuration (see Table 5.3) as a function of time. Results were obtained with three different time-steps as shown in Fig. 5.4. A time-step of 10 msec was chosen based on these results to solve the Markov model for other configurations.

The following results were obtained regarding the probability of system failure due to a lack of perfect coverage (PF_C).

First of all, the system failure probability increases linearly as a function of time, as evidenced by the results shown in Fig. 5.5.

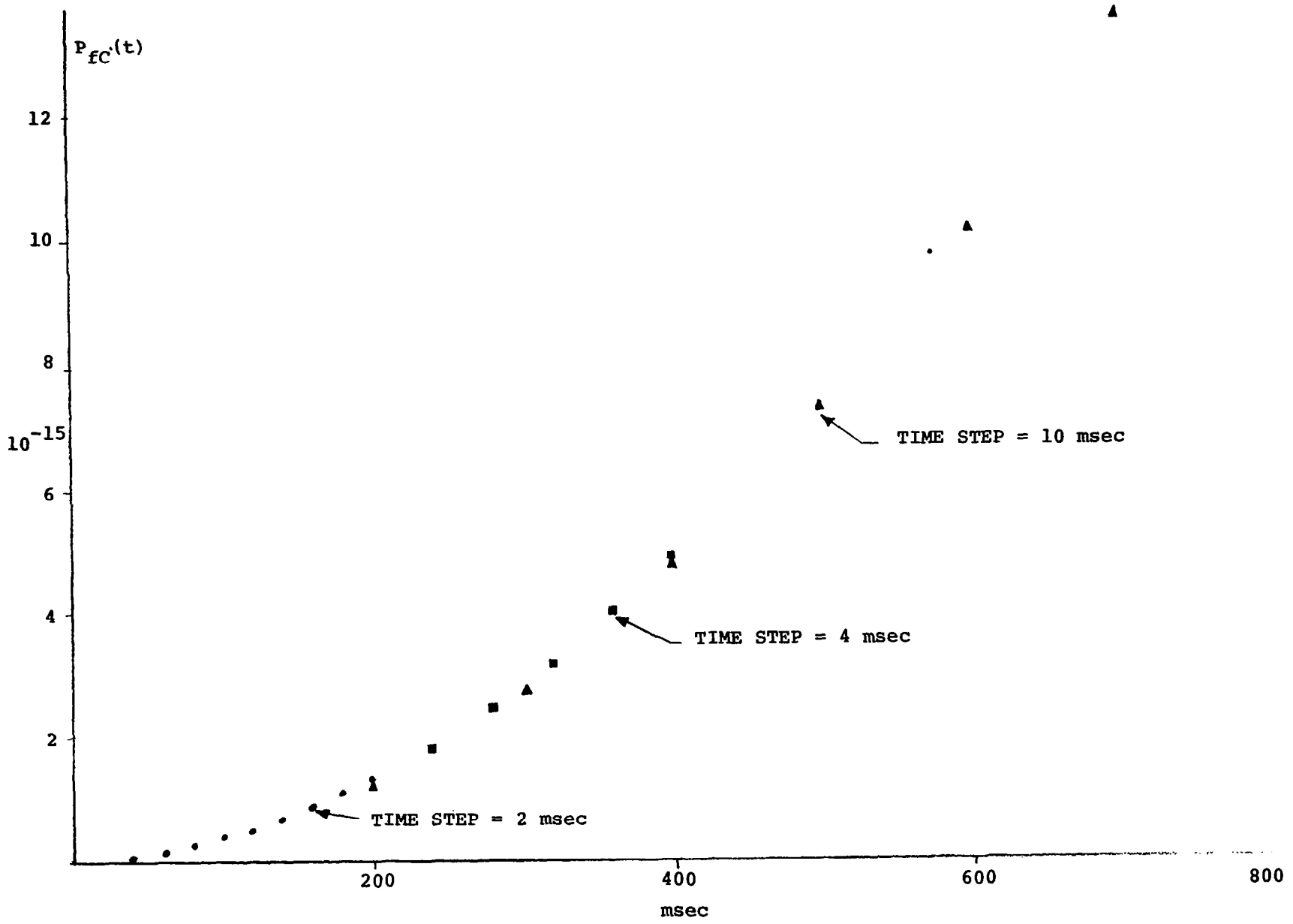


Figure 5.4 Effect of Time Step Variation on Markov Model Solution.

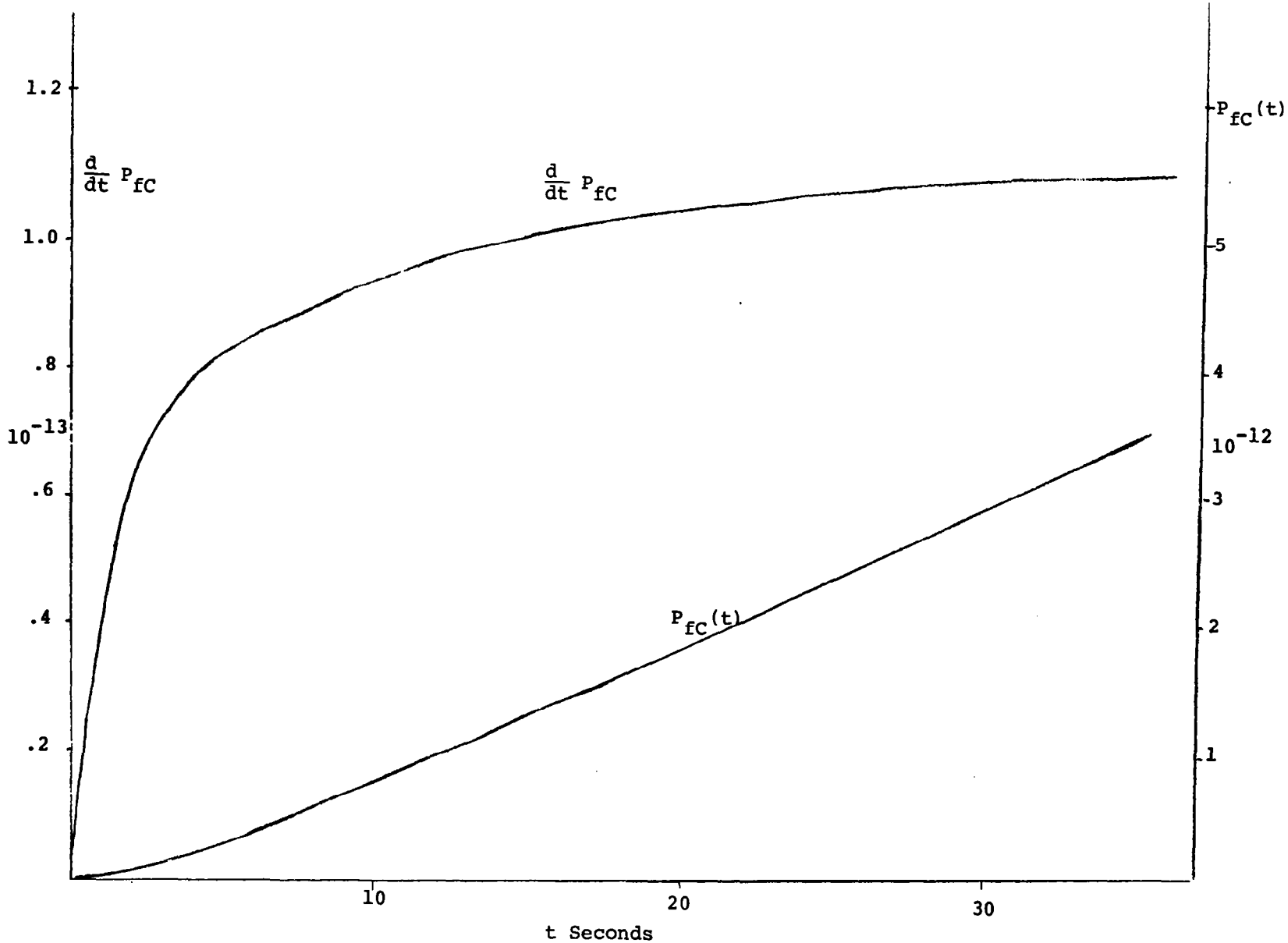


Figure 5.5 Prob. of Failure Due to Lack of Coverage and its Rate of Change.

Although these results apply only to the first minute of the computer operation a simplified Markov model was solved to extend the same results to more than five hours. This second model is described in a later section of this chapter.

A number of runs were made to assess the impact of variations in the system configuration and the component failure and recovery rates on the system reliability. In all these cases the iterative solution of the Markov model was continued to a point where the failure probability attained a nearly constant slope. This value of the slope was used to compute the system failure rate per hour, and is shown in Figures 5.6 to 5.9. In Fig. 5.6 it is seen that improving all the component failure rates by one order of magnitude results in the betterment of the system reliability by two orders of magnitude. On the other hand, changing the recovery rates ten-fold produces a ten-fold change in the system reliability as shown in Fig. 5.7. In both of these cases the failure rates or the recovery rates were varied collectively for all types of units.

The next two figures show the effect of varying the failure and recovery rates for each type of unit individually. It is seen that the system failure probability due to the lack of coverage is most sensitive to variations in the processor and memory failure rates around the operating point defined by the baseline parameters. It is least sensitive to the bus failure rate variation. This is simply due to the fact that the processor and memory failure rates are the highest of all at the operating point, and therefore contribute the most to the system failure probability. Similarly, since the recovery time from a BGU failure is the greatest, the system reliability is most affected by changes in this parameter.

Figure 5.8 also shows the effect of varying the number of units in the multiprocessor. This variation does not significantly impact the system failure probability due to the lack of coverage. In fact, decreasing the number of processors improves the system reliability a little since there are fewer processors to contribute to the system failure rate.

5.3 Second Markov Process Model of the Multiprocessor

The model described in the preceding section has 146 states. Although this has the benefit of representing the actual system accurately, it suffers from being computationally complex and inefficient.

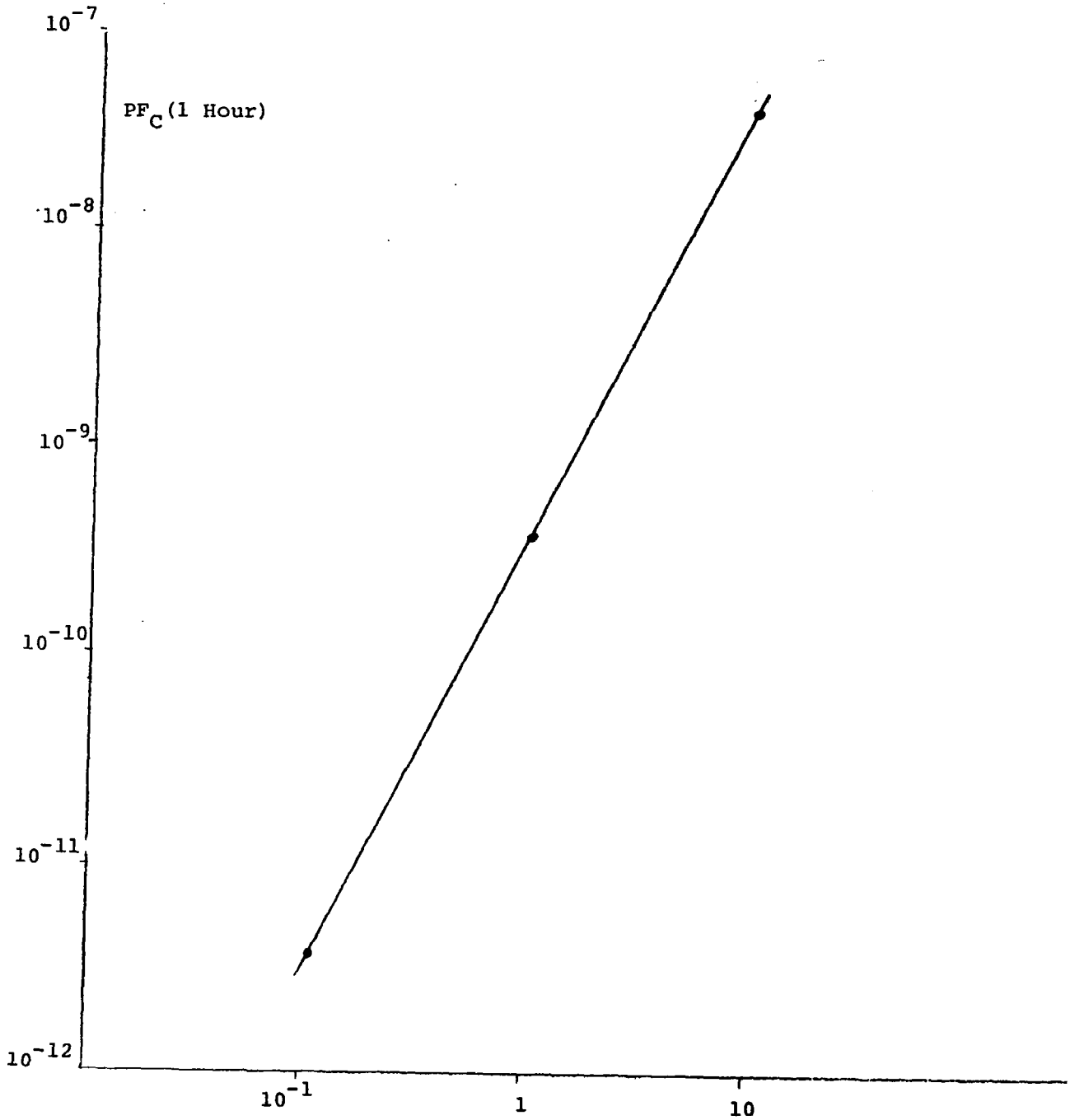


Figure 5.6 Failure Rate Factor λ.

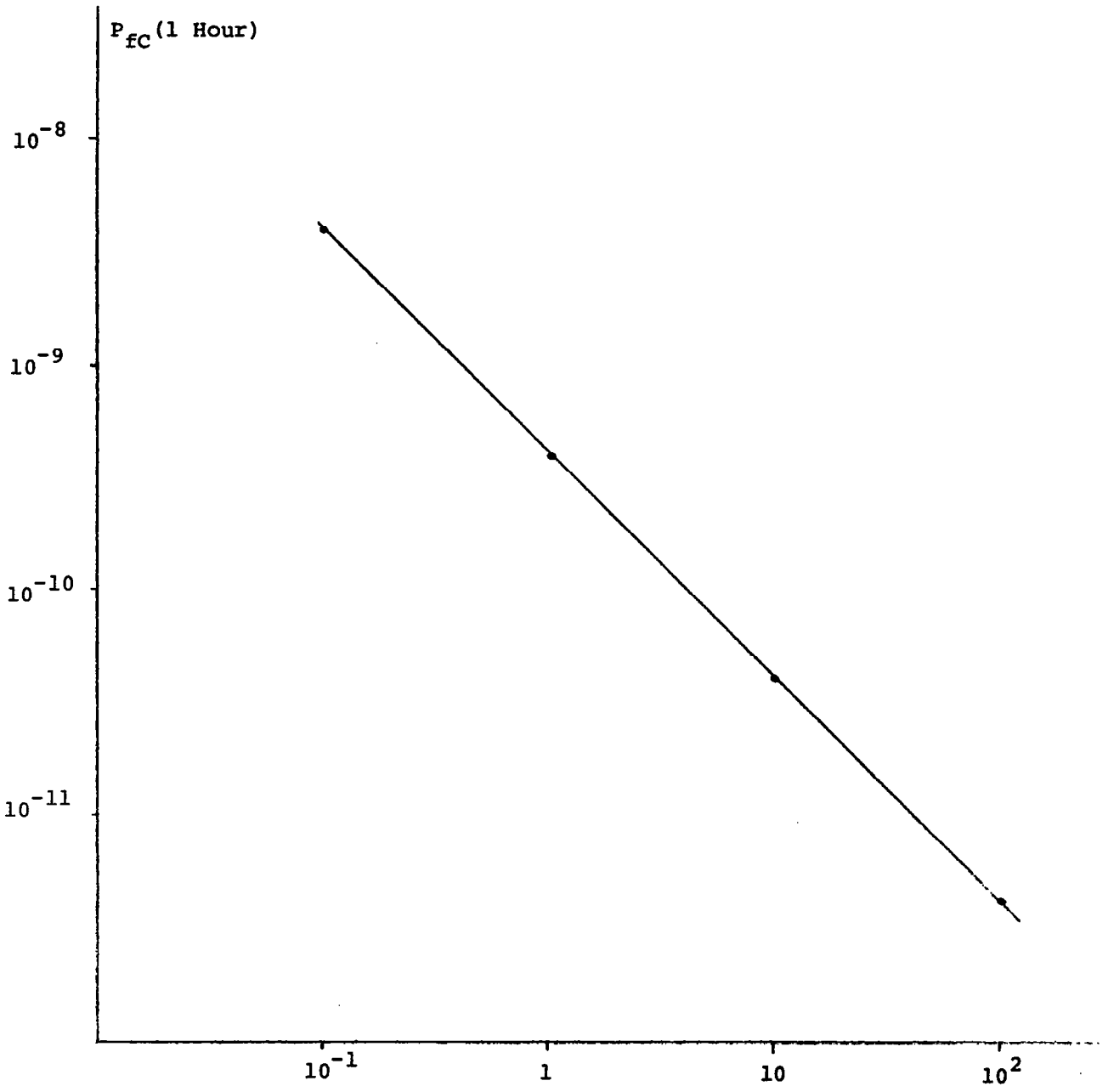


Figure 5.7 Recovery Rate Factor ρ.

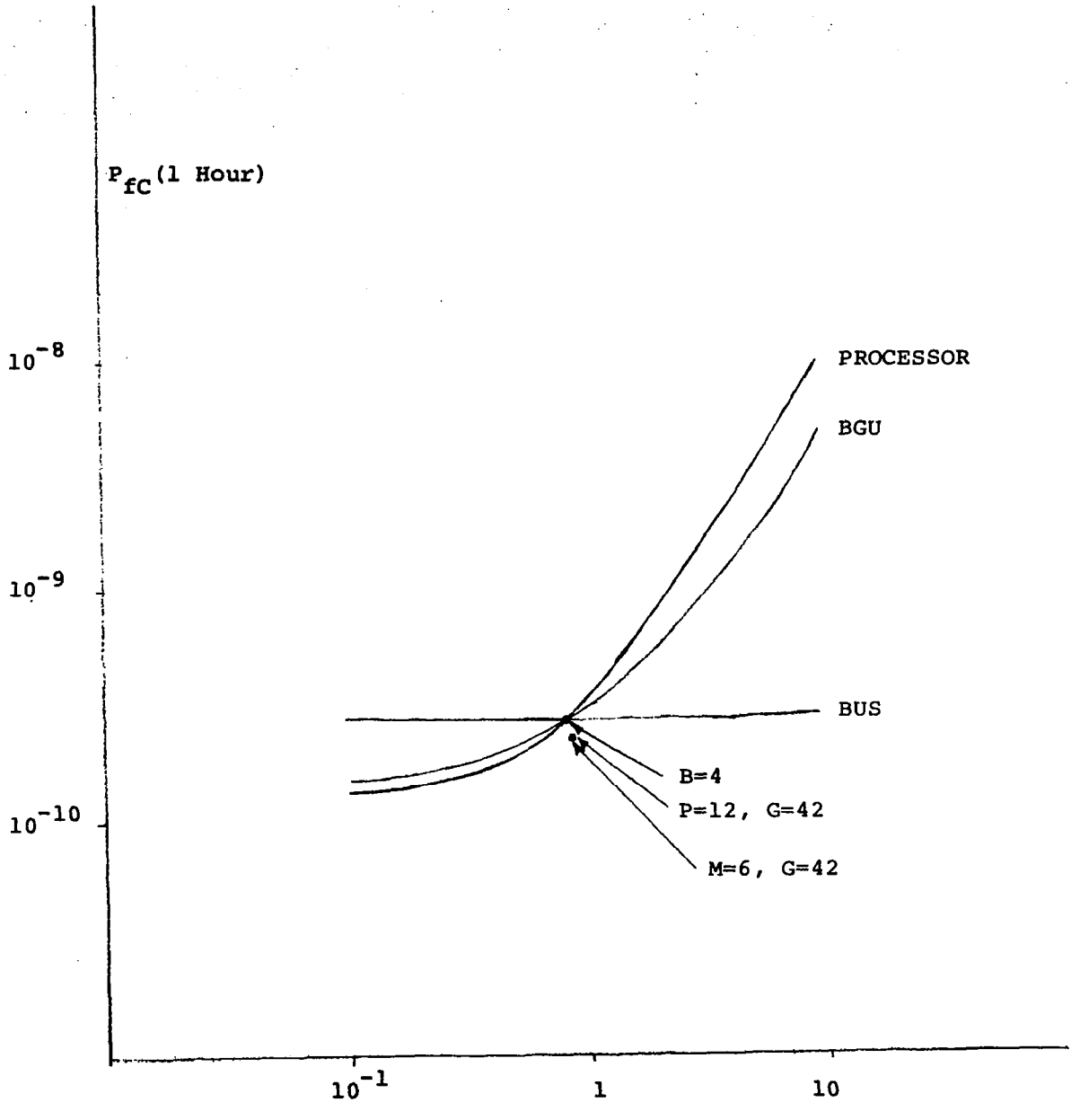


Figure 5.8 Component Failure Rate Factor.

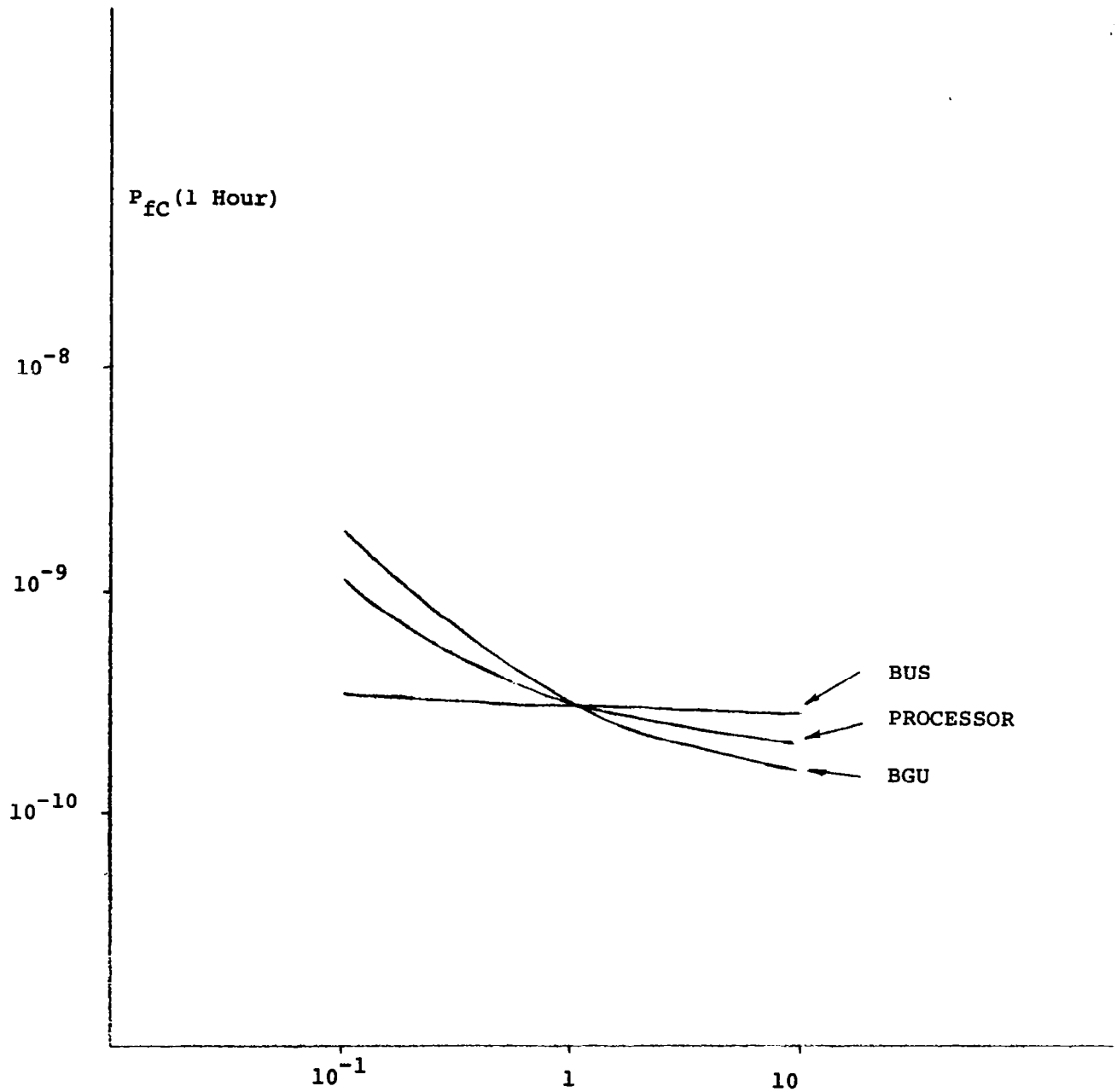


Figure 5.9 Component Recovery Rate Factor.

The computation of the system failure probability was therefore limited to a short time span, and it was assumed based on these results that the failure probability continues to grow linearly as a function of time. To verify this assumption, a second Markov model was developed with only 11 states, as shown in Fig. 5.10.

On comparison of this model to the earlier one of Fig. 5.2 the major differences become obvious. First, the detection of a failure returns the system to the perfect state, which is the starting state. This approximation was based on the fact that this model is being used to predict the failure probability due only to the lack of coverage and not due to the exhaustion of spares or equipment. This second phenomenon is considered separately in a later section. The reason one can separate the effects of these two mechanisms of failure is that each one dominates the system reliability in a different time span.

The second approximation involves truncating the model to two faults. That is, any combination of three faults is assumed to cause a system failure. In this case the approximation can be justified based on the amount of error caused by the truncation of the model as explained below.

Figure 5.11 shows that in the steady state in a typical case the system failure rate is 1.87×10^{-10} per hour. Of this 1.85×10^{-10} is contributed by double faults. That is, the triple fault combinations increase the system failure rate only by one per cent. Therefore truncation of the model at 2 faults causes less than one per cent inaccuracy in the results.

The computer program to solve this model is listed in Appendix II. The result is presented in Fig. 5.12. The system failure probability, in fact, does increase linearly with time, assumed earlier, for the first five hours. The iterative solution of the Markov model was stopped at this point. However it is obvious that the failure probability will continue to grow linearly as long as the probability of the system being in the "All Good" state (see Fig. 5.11) remains almost unity. This will be the case for at least several hundred hours.

5.4 Combinatorial Reliability Model of the Multiprocessor

So far we have investigated the effect of the lack of perfect coverage on the system reliability. This analysis showed the degree of susceptibility of the system to two or more near simultaneous faults,

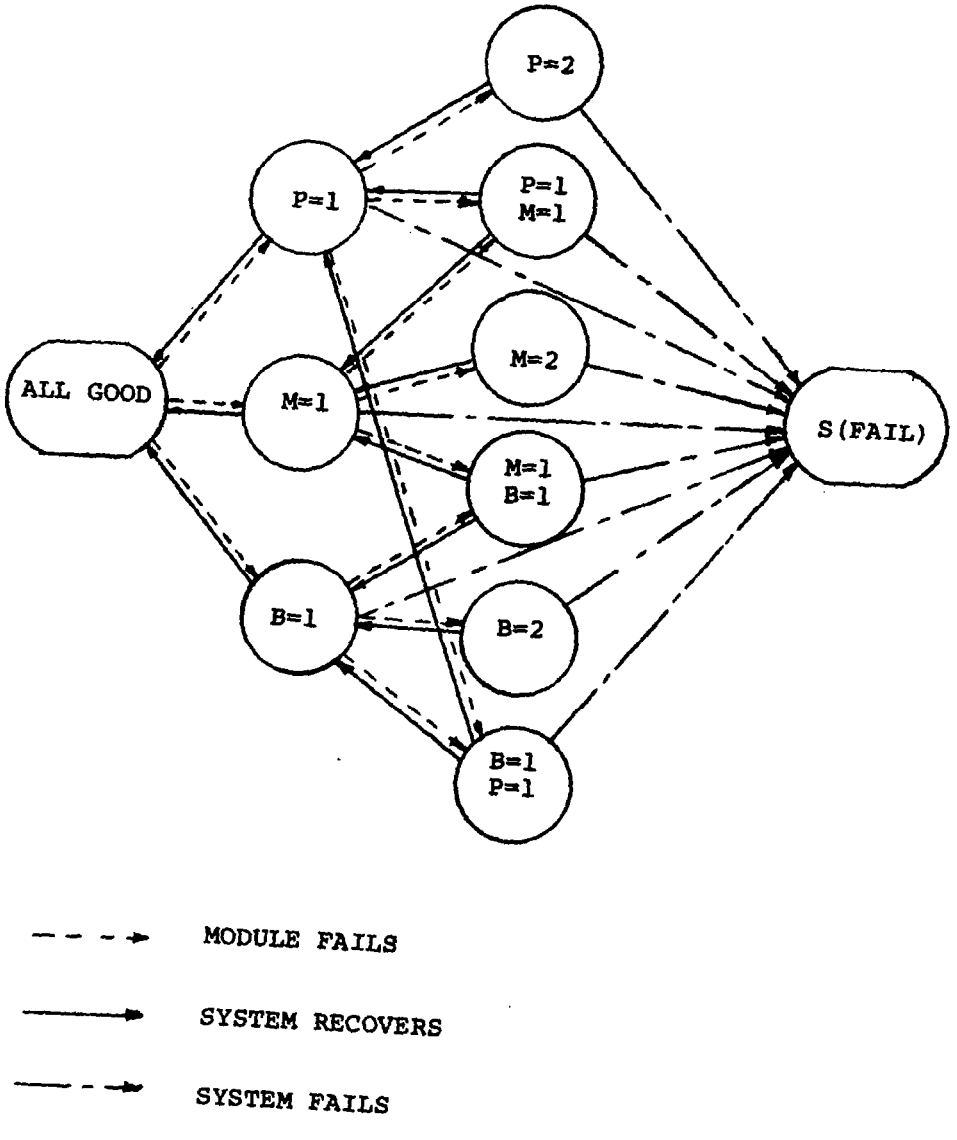


Figure 5.10 Simplified Cards Markov Model.

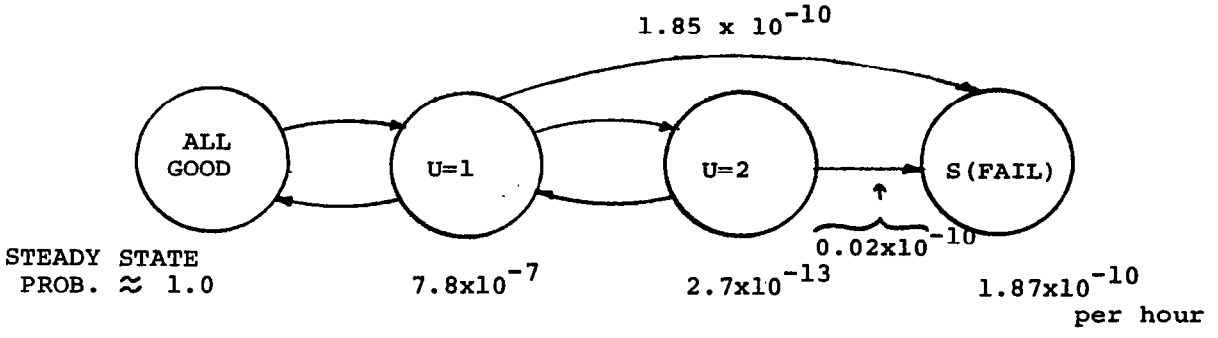


Figure 5.11 Relative Contributions to System Failure State.

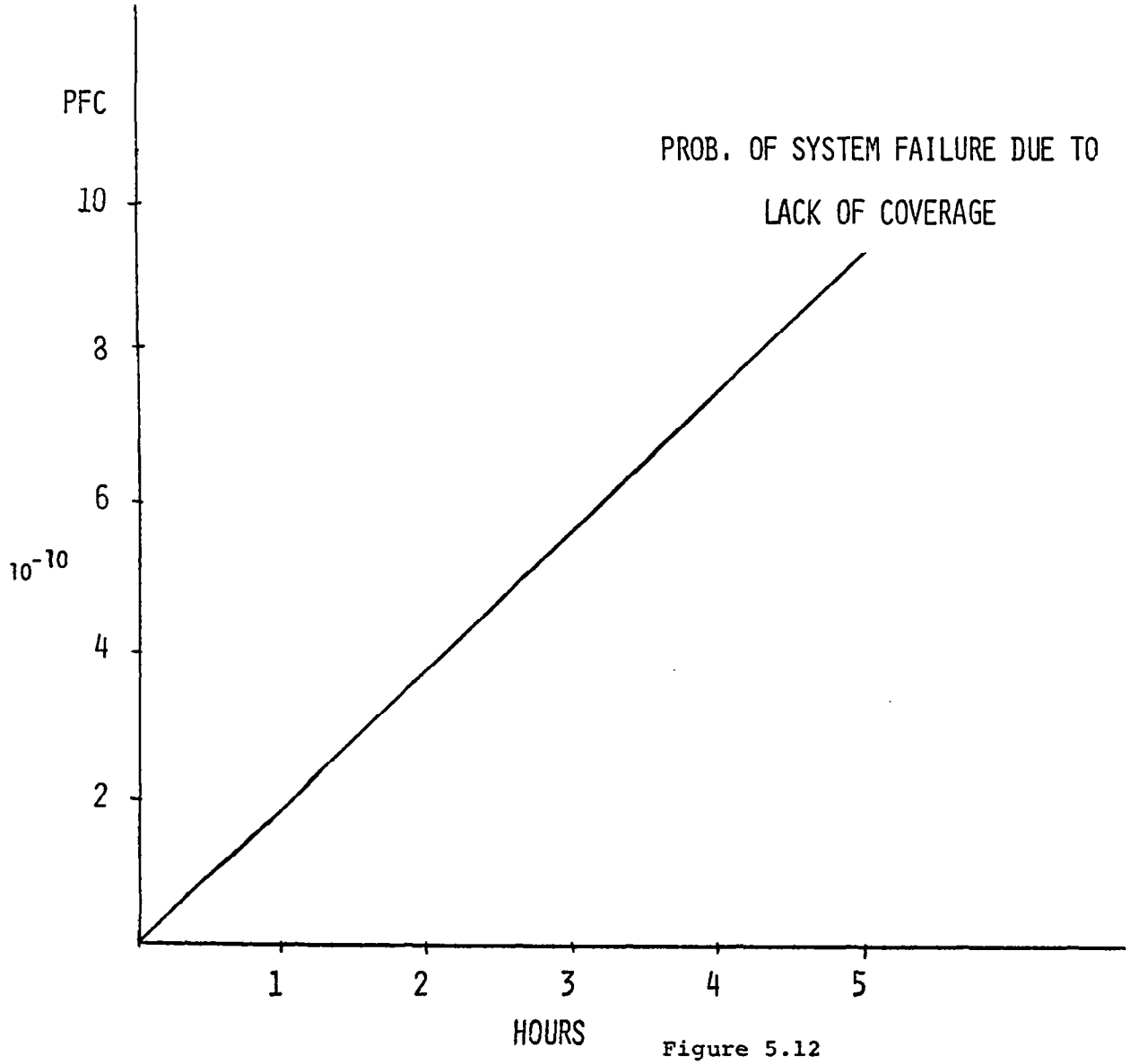

PFC
10
8
6
10^{-10}
4
2
PROB. OF SYSTEM FAILURE DUE TO
LACK OF COVERAGE
1
2
3
4
5
HOURS

Figure 5.12

that is, the system failure probability due to non-zero time taken to detect, identify and recover from a failure. It was assumed during this analysis that enough units of each kind were still in working condition to perform all the critical tasks. Eventually, of course, enough failures would occur so that the supply of spare units would be exhausted and the system would degrade in compuational power and hardware to a point where it could no longer perform the minimum work load required for mission success. To study this mechanism of failure, it was assumed that the time to detect, identify and recover from a failure equals zero. That is, the system has perfect coverage. With this assumption, the problem of determining the system failure probability as a function of time and number of units of each kind becomes a simple exercise in combinatorial analysis.

Figure 5.13 shows a link diagram of the system depicting each unit as a serial or a parallel link in a chain that must remain unbroken for the system to function successfully. Starting from the left, the first group of parallel links represents P processors. Let us assume that a minimum of P_o processors, M_o memories and B_o buses are required for flight critical performance. A BGU failure is considered to be an effective processor or memory failure. Therefore, in the first group, at least P_o out of P links must remain operational for the system to be successful. Here each link has a failure rate of $\lambda_P + 2\lambda_G$. Similarly, the next two link groups represent the memory and the bus units respectively.

Using the link diagram of Figure 5.13 one may write the following expression for the system failure probability by inspection.

$$PFS = 1 - PSS$$

$$PSS = \sum_{NP=P_o}^{P} \binom{P}{NP} (PSG^2 \cdot PSP)^{NP} (1 - PSG^2 \cdot PSP)^{P-NP} \cdot$$

$$\sum_{NM=M_o}^{M} \binom{M}{NM} (PSG^2 \cdot PSM)^{NM} (1 - PSG^2 \cdot PSM)^{M-NM} \cdot$$

$$\sum_{NB=B_o}^{B} \binom{B}{NB} PSB^{NB} \cdot (1 - PSB)^{B-NB} \cdot$$

Here, PF = Probability of failure of a processor (P), memory (M), BGU (G) or the system (S),

and PS = Probability of success of P, M, G or S.

DETERMINATION OF NUMBER OF SPARES FOR EACH TYPE OF MODULE

MINIMUM EQUIPMENT REQUIRED FOR FLIGHT CRITICAL PERFORMANCE: P_0, M_0, B_0

RELIABILITY LINK DIAGRAM

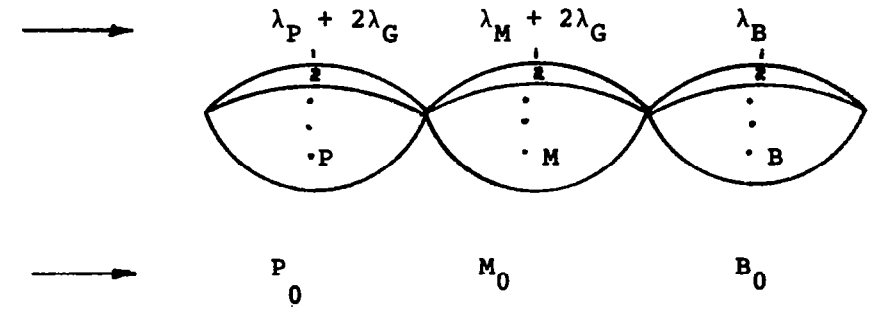


Figure 5.13

A computer program was written in the PL/I language to compute the system failure probability using the above equation. Figure 5.14 shows the results for the baseline parameter values listed in Table 5.4. It was assumed that at least three processor triads and one memory triad will be required to do all the flight critical tasks. Initially, the probability of failure due to exhaustion of spares is a few orders of magnitude smaller than that due to simultaneous failures. However the former increases at a much faster pace. Therefore at some point approximately a few hundred hours from the beginning the likelihood of the baseline system failing from a lack of equipment becomes stronger than from simultaneous failures. This predominence of each of the two failure mechanisms in a different mission time-span justifies their independent evaluation. The combined system failure probability due to all modes of failure is discussed in the last section of this chapter.

5.5 System Failures Due to BGU Enable Failures

A Bus Guardian Unit links the processor and memory modules with the buses. Its function is to prevent a failed processor or memory unit from corrupting information on the buses. However a BGU is also susceptible to hardware failures. BGU failures may be divided into two classes: (1) Disable Mode and (2) Enable Mode.

In the disable mode, the unit associated with the failed BGU can no longer communicate on the bus. This is akin to losing a spare unit. However the second failure mode is not so benign. In the enable mode, a failed BGU connects its associated unit to more than one bus. Of course, two such failures do not harm the system until the associated unit also fails. Then the result is a catastrophic system failure. If there is more than one BGU in each processor or memory module, then all of them have to fail in the enable mode before the system is affected.

The following equation expresses the system failure probability due to the BGU enable failures in terms of the processor, memory and BGU failure probabilities.

$$FS = 1 - (1 - PFP \cdot PFGE^2)^P \ (1 - PFM \cdot PFGE^2)^M$$

Here PF = probability of failure of a processor (P), memory (M) or the system (S),

and PFGE = probability of BGU failure in the enable mode.

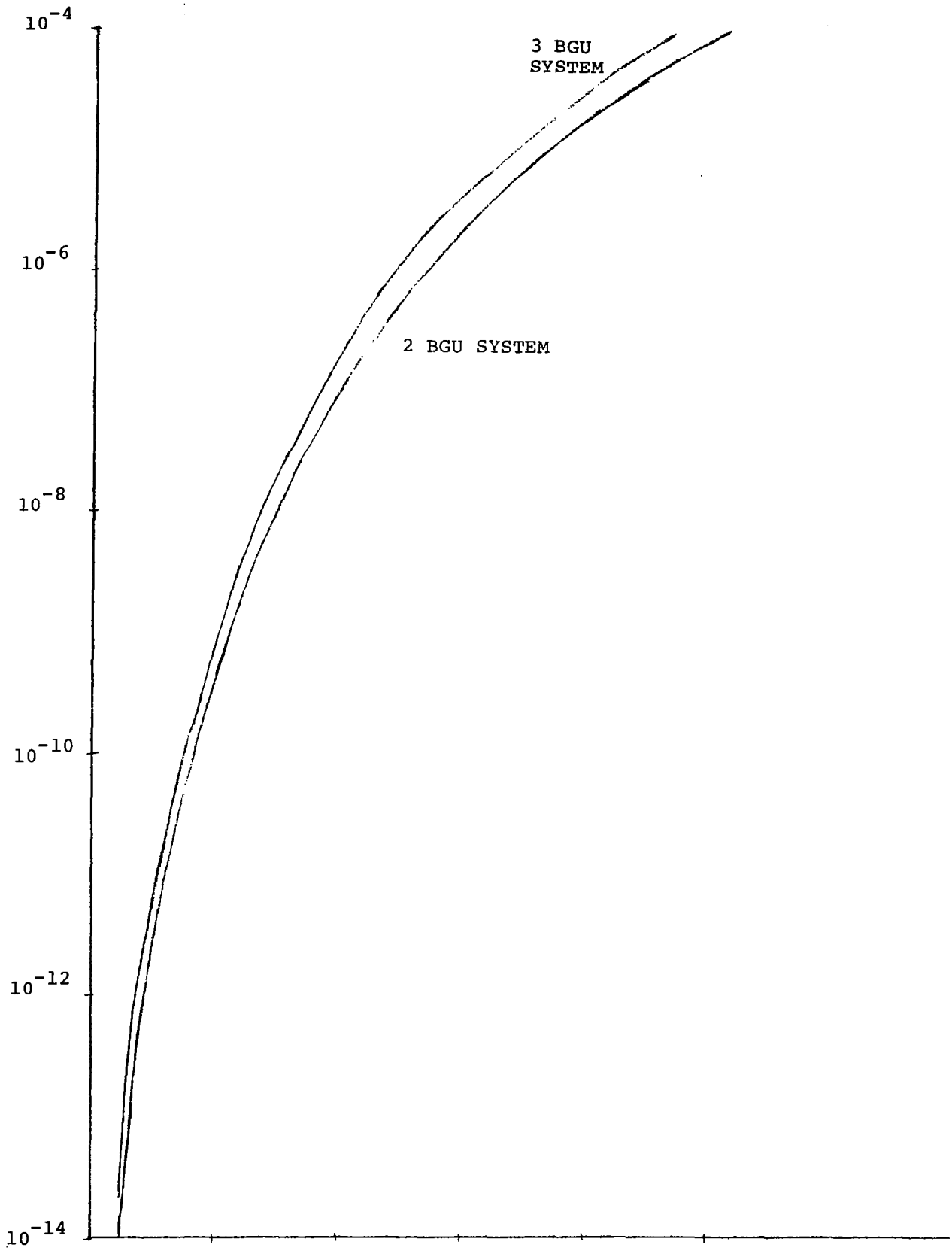


Figure 5.14 Prob. of Failure Due to Exhaustion of Spares.

TABLE 5.3

BASELINE PARAMETER VALUES

SYSTEM CONFIGURATION

# Processors	= 15
# Memory Units	= 9
# Buses	= 5
# BGUs	= 48

FAILURE RATES		PER HOUR	MTBF
Processor	=	10^{-4}	10^{4} Hours
Memory	=	10^{-4}	10^{4}
Bus	=	10^{-6}	10^{6}
BGU	=	10^{-5}	10^{5}
BGU Enable	=	5×10^{-6}	2×10^{5}
BGU Disable	=	5×10^{-6}	2×10^{5}

RECOVERY RATES (Time to Detect, Identify & Recover From a Failure)

		RATE	TIME
Processor	=	1 per second	1 second
Memory	=	1	1
Bus	=	10	0.1
BGU	=	0.1	10

TABLE 5.4

BASELINE PARAMETER VALUES

INITIAL SYSTEM CONFIGURATION		MIN. REQ'D FOR FLT. CRITICAL PERFORMANCE
# Processors	= 15	$P_o = 8$
# Memory Units	= 9	$M_o = 2$
# Buses	= 5	$B_o = 2$
# BGUs	= 48	

FAILURE RATES	PER HOUR	MTBF
Processor	10^{-4}	10^{4} Hours
Memory	10^{-4}	10^{4}
Bus	10^{-6}	10^{6}
BGU	10^{-5}	10^{5}
BGU Enable	1×10^{-6}	10^{6}
BGU Disable	9×10^{-6}	1.1×10^{5}

This equation assumes two BGUs in each processor and memory module. Figure 5.15 shows the system failure probability due to this failure mechanism as a function of time. Baseline parameter values from Table 5.4 were used to arrive at these results. It was assumed that the enable mode failures can be biased to be less probable than the disable mode ones. The impact of improving the BGU enable failure mechanism is also shown in the same figure. An order of magnitude improvement in the BGU failure rate decreases the system failure probability by two orders of magnitude.

Another way to diminish the system susceptibility to enable mode failures is to increase the number of BGUs in each module. Figure 5.15 shows the system failure probability when each module has 3 BGUs. The result is a 3 order of magnitude decrease in failure probability. However it also slightly increases the system failure probability due to the exhaustion of spares as shown in Fig. 5.14. This mode of failure also impacts system availability or dispatch reliability. This is dealt with in the next section.

5.6 System Availability/Dispatch Reliability

Dispatch reliability in the present context may be defined as the probability of having the computer system before each flight in a condition good enough to meet the mission reliability requirements. That is, there should be enough operating modules in the system to assure a given probability of success at the end of, say, a ten hour flight. Dispatch reliability is important because it directly affects system maintenance cost. For an airline, a high dispatch reliability of the computer system could mean fewer maintenance points along its routes with consequent savings in operating cost.

A given level of dispatch reliability can be achieved for the present system by varying the number of spares for each type of module. A relationship is developed between these two variables as follows.

Table 5.5 relates the system size to the system reliability for the baseline failure rates defined in Table 5.4. Let us assume that the required probability of failure at the end of a 10 hour flight is 10^{-9} or smaller. It is now possible to pick out from Table 5.5 a minimum system size that will meet the required mission reliability criterion. This system size is 11 processors, 5 memory units and 3 buses. To this basic size we would have to add enough spares to meet a required dispatch reliability PS at a time T. Table 5.6 shows three system

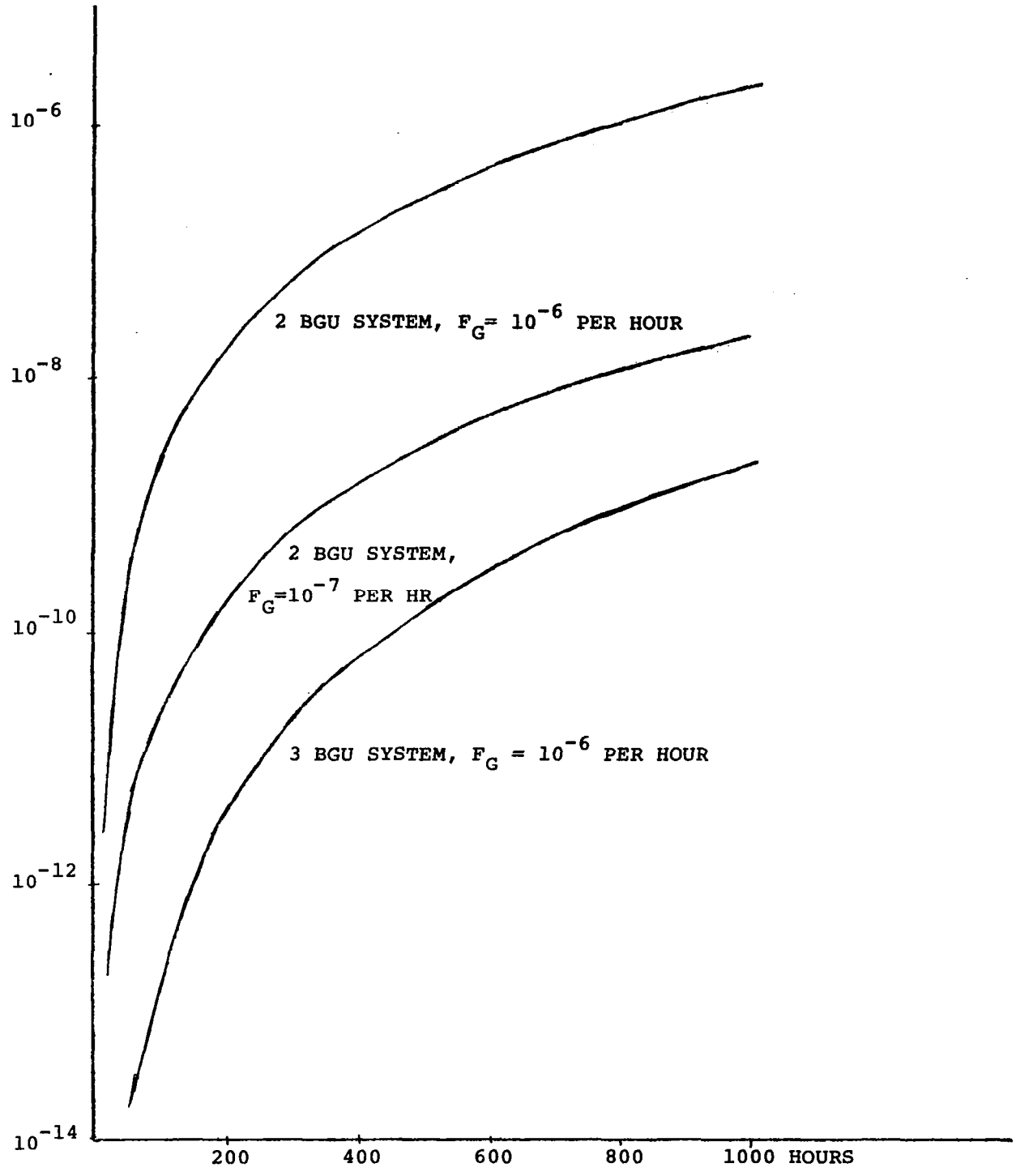


Figure 5.15 System Failure Prob. Due to BGU Failures in Enable Mode.

TABLE 5.5

EQUIPMENT FAILURE PROBABILITIES

P = 11, M = 5, B = 3

OPERATING EQUIPMENT BEFORE TAKE-OFF			PROB. OF FAILURE DUE TO LACK OF EQUIPMENT @ 10 HOURS
P	M	B	
15	9	5	$<10^{-16}$
12	7	4	2×10^{-12}
12	7	3	3×10^{-10}
11	6	3	9.8×10^{-10}
11	5	3	9.9×10^{-10}
11	4	3	7.9×10^{-9}
11	3	3	4.4×10^{-6}
10	5	3	2.1×10^{-7}

TABLE 5.6

INITIAL CONFIGURATION SO THAT AT LEAST 11 PROCESSORS, 5 MEMORY UNITS AND 3 BUSES WILL SURVIVE WITH A PROB. PS AFTER 300 HOURS

DESIRED	REQUIRED CONFIG.			
PS	P	M	B	ACTUAL PS
0.99	13	7	3	0.988
0.995	14	7	3	0.996
0.999	15	8	4	0.9998

configurations to meet three different dispatch reliability requirements at a fixed time T equal to 300 hours. This figure was chosen for time because for most airlines an aircraft would undergo periodic engine maintenance at its home base, which has all the maintenance facilities, in 300 hours or less. For a fixed system configuration, decreasing the aircraft turn-around time would obviously increase the system dispatch reliability. This is shown in Fig. 5.16 for each of the three system configurations of Table 5.6.

For the present system, the probability of failure during a flight depends not only on the amount of equipment operating before take-off but also on the number and type of failed units in the system before take-off. This is due to the BGU enable mode failures described in the preceding section. Table 5.7 shows the system failure probability after 10 hours for various initial configurations of failed modules. It is evident that the most significant contribution to the system failure probability is made by a BGU failed in the enable mode. The probability of at least one BGU failing as such in the baseline system in 300 hours is 0.015. That is, the dispatch reliability would be 0.985 considering only the BGU enable failures.

One way to reduce the impact of BGU failures on the system dispatch reliability is to increase the number of BGUs in each processor and memory module from two to say, three. The result is again shown in Table 5.7. For this system configuration, the dispatch reliability for a 300 hour aircraft turn-around period would increase to 0.999994. The next section summarizes the results of the relibility study.

5.7 Summary

So far we have investigated three different mechanisms of failure which can result in a catastrophic system failure. These are

1. Occurrence of near simultaneous failure of two or more units,
2. Failure of Bus Guardian Units in the enable mode,
3. Exhaustion of equipment.

It was stated earlier that each cause is predominant during a different time span. Therefore it was possible to study the probability of system failure due to each cause separately. Figure 5.17 combines the results of the three different studies. It shows for a representative case the

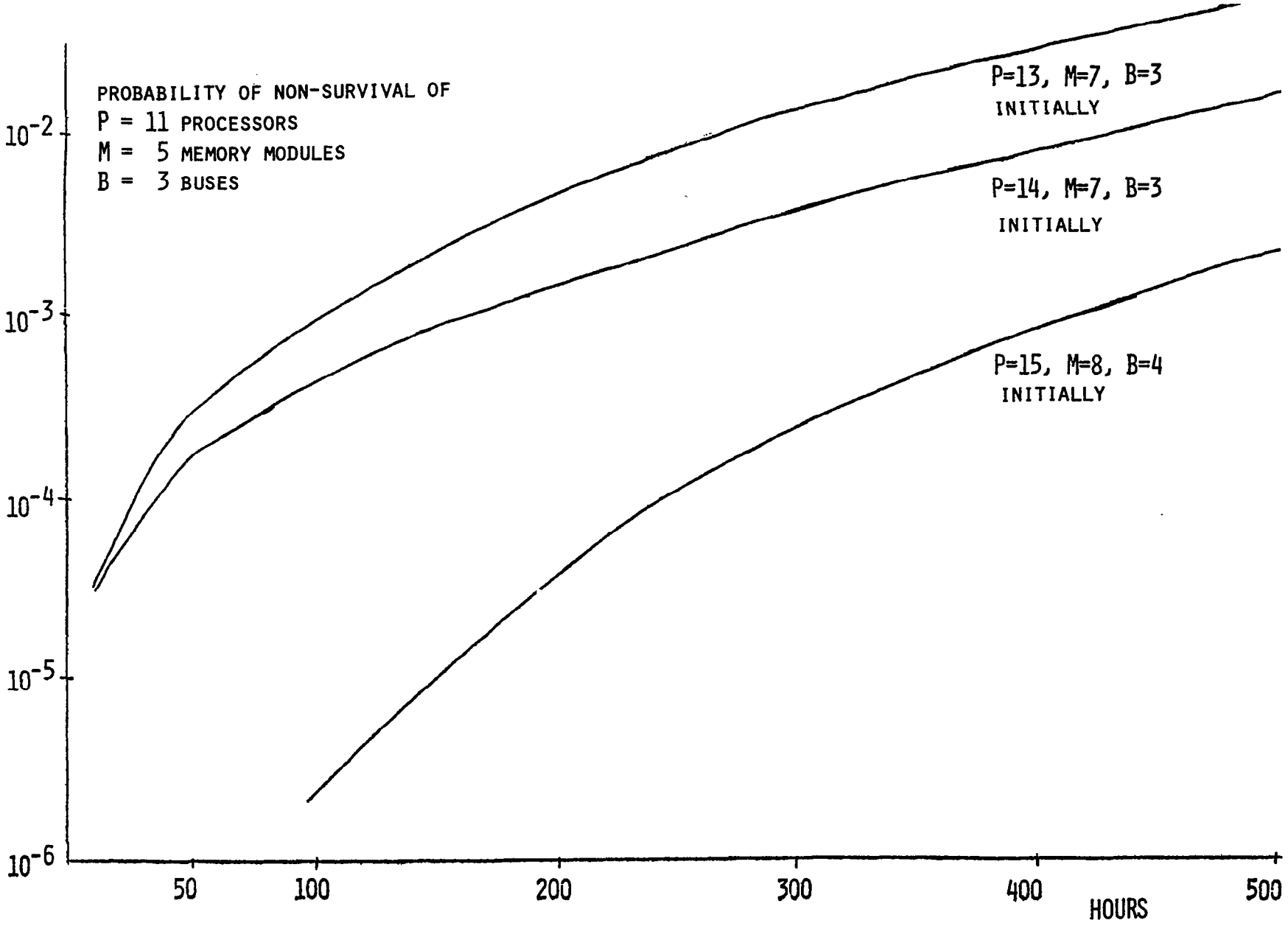


FIG. 5.16 PROBABILITY OF SPARE MODULE EXHAUSTION, EXAMPLE

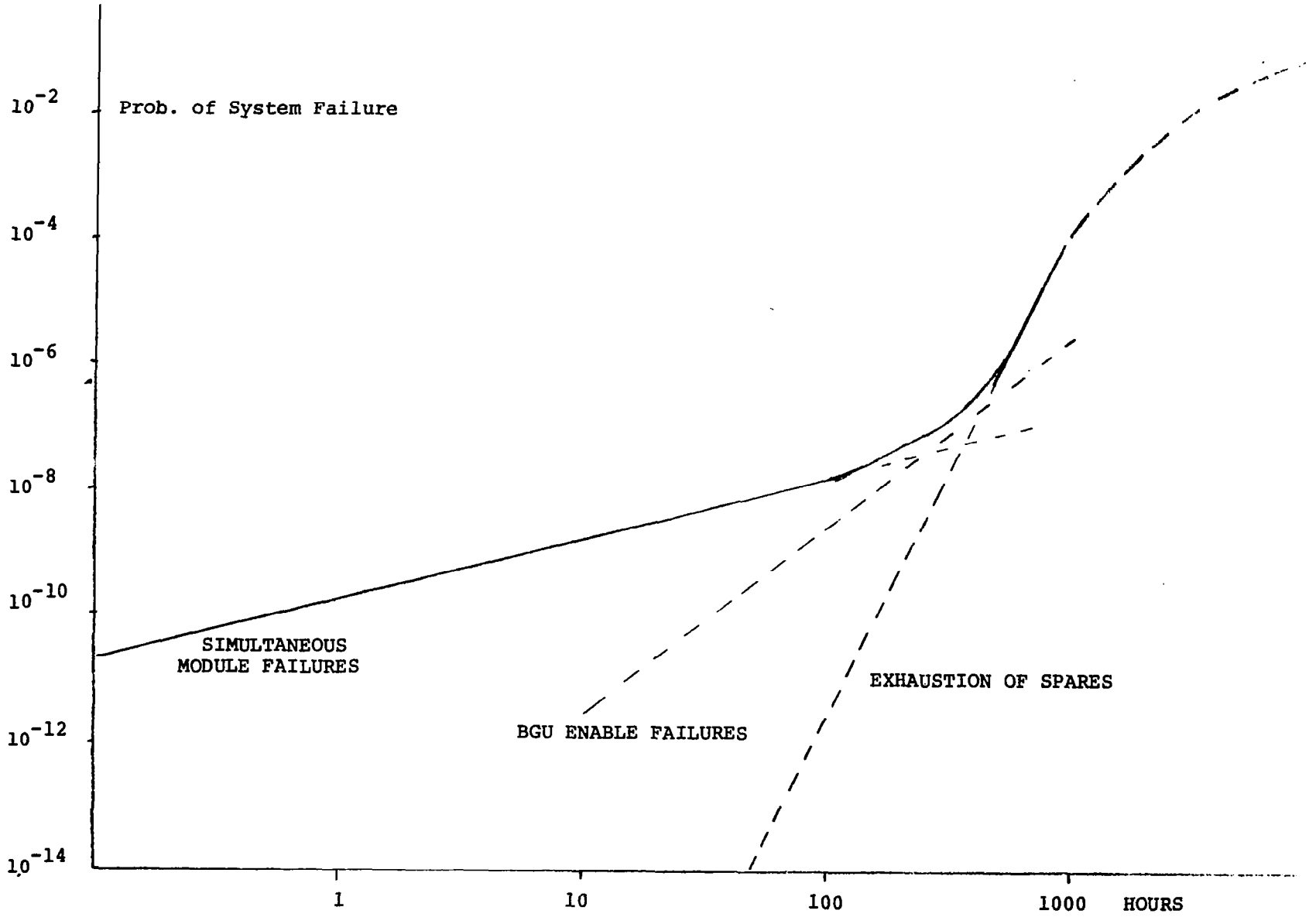

Prob. of System Failure
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
SIMULTANEOUS
MODULE FAILURES
BGU ENABLE FAILURES
EXHAUSTION OF SPARES
1
10
100
1000
HOURS

Figure 5.17

TABLE 5.7

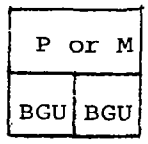
ANALYSIS OF SYSTEM FAILURES DUE TO BGU FAILURES IN ENABLE MODE

TOTAL EQUIPMENT: $P = 15, \quad M = 9$

FAILURE RATES: $\lambda_P = 10^{-4}$ per hour

$\lambda_M = 10^{-4}$ per hour

$\lambda_{GE} = 10^{-6}$ per hour



INOPERATIVE EQUIPMENT BEFORE TAKE-OFF	PROB. OF SYSTEM FAILURE AT 10 HOURS
NONE	1×10^{-13}
X	1×10^{-10}
X X	2×10^{-10}
X X X	3×10^{-10}
X	1×10^{-8}
X X	2×10^{-8}
X X X X	2×10^{-8}
X X	1×10^{-5}
X X	1×10^{-3}
X	3 BGU SYSTEM 1×10^{-13}
X X	3 BGU SYSTEM 1×10^{-8}

PROB. 1 BGU FAILS IN ENABLE MODE (OUT OF 48 BGUs) IN 300 HRS: 0.015.

PROB. 2 BGUs FAIL IN THE SAME MODULE (OUT OF 72 BGUs) IN A 3 BGU SYSTEM IN 300 HOURS: 6.5×10^{-6}.

system failure probability due to all possible failure modes for a time period of one thousand hours. It is seen that initially the probability of catastrophic system failure is low and dominated by near simultaneous failure of two or more units. Eventually, the system may fail due to a lack of spares.

Results regarding system availability can be summarized as follows. The BGU enable mode failures can have significant impact on the system availability for the baseline system configuration and failure rates. This can be almost totally eliminated by biasing BGU failures against the enable mode or by having three BGUs in each processor and memory module. Having taken care of this, the baseline system is seen to have a dispatch reliability of greater than 0.9999.

CHAPTER 6

AN APPROACH TO SOFTWARE RELIABILITY

This chapter describes the specific means by which a low-cost system could support development of software for the fault-tolerant multiprocessor computer system.

6.1 Introduction

Software development has proved to be overly expensive, in terms of both initial costs and overall life cycle costs. Advanced software development tools are required to ensure manageable software life cycle costs. This chapter introduces and justifies a comprehensive software methodology called Higher Order Software (HOS) which is applicable to software development for the multiprocessor.

HOS is a discipline which lowers software life cycle costs. The basic HOS axioms, tools, and methods have been described elsewhere [5]. This chapter introduces the HOS tools and describes their functions briefly. Subsequent portions of this chapter describe applicable HOS methods and detail how these HOS facilities could be implemented at low cost.

6.2 Higher Order Software Methodology

HOS originated during software development for APOLLO and for SKYLAB. The APOLLO software development required approximately 2,000 man years, during which extensive software anomaly lists were maintained for later analysis. These data show that 13% of the software errors uncovered during systems integration were analytical errors, such as specifying sine when cosine was intended. An overwhelming 73% majority of the errors, on the other hand, were associated with interactions between modules. These errors fall into categories such as misinterpreting subroutine calling sequences, passing the wrong arguments, treating variables in the wrong way, misinterpreting interface specifications, and the like. Because most errors involved interactions between modules, it would have been little good to "prove the correctness" of individual

modules. The APOLLO experience showed that programmers can in general develop correct modules. Most errors arise in interfacing modules incorrectly. Accordingly, HOS methods concentrate on ensuring that interfaces between modules are correct.

In the course of HOS development, six axioms concerning relationships between system modules were stated. These axioms are defined rigorously in [5] and stated informally at the end of this chapter. If all system components interact in a manner consistent with the HOS axioms, the interface errors which so plagued the APOLLO software development effort are automatically eliminated. The effect of following the HOS axioms is to guarantee interface consistency on a system-wide basis. This would have eliminated 73% of the APOLLO software integration errors.

Programmers tend to regard the HOS axioms as obvious, and claim to follow them. Experienced managers,however, are well aware that programmers do not always program in a systematic manner. In order to ensure that systems follow the rules, HOS incorporates analyzers which detect system specifications and module interactions which are not consistent with the axioms. The HOS analyzers and other automatic tools are listed and described briefly here, and are discussed in detail below.

1. Specification Language and Specification Analyzer.
   The specification language is used to describe relationships between system modules in an unambiguous manner. The specification analyzer examines the system specifications for relationships between modules which are not consistent with the axioms. The analyzer also generates a diagram showing how the system components interact. The system description presented to the specification analyzer is not intended to describe a specific system implementation. This function is served by other HOS tools.

2. Higher Order Interface Language (HOIL).
   A HOIL is used to implement the software portion of the system after it has been described using the specification language. A HOIL is a programming language in which statements describing interactions between modules can be easily identified so that they can be checked for consistency. A HOIL may or may not contain other facilities such as supplied by compiler languages.

3. Interface Analyzer.
   After modules have been implemented using a HOIL, the interface analyzer examines interfaces between the modules to determine whether or not the interfaces are consistent with the specification. This is feasible because a HOIL provides information which makes interfaces easy to locate.

4. Control Mapper.
   The mapper also examines interface specifications after the modules have been implemented using the HOIL. The mapper draws a picture of the relationships between modules, which is checked to be sure that the specification has been implemented properly. This map reveals the relationship between the various software components, and serves as a documentation aid.

5. Concordance Generator.
   The concordance program produces a list of all data in the system and tells where each datum is referenced. References are categorized in terms of examine only, store only, and both store and examine. This list makes it easy to find all modules which reference a given datum, regardless of the different names which are used in different modules.

6. Narrative Documentor.
   The documentor collates the program listing with descriptions of the data used by the program. When data are referenced in a module, the documentor copies the data description into the program listing. This ensures that data have to be documented only once, and that the documentation and the source listing are kept together. This is helpful during maintenance.

Most of these facilities do not require any more information than is already available to the language processor, and some compilers provide part of this information to users. All FORTRAN compilers, for example, are aware of the uses of all variables which are referenced in a program, but many do not print this information in the listing. Furthermore, FORTRAN compilers do not generate information which can be examined for consistency between modules. The HOS specification analyzer ensures that only proper relationships between modules are included in the design, and a HOIL processor generates information needed so that other routines can verify that the interfaces described in the specification have been implemented properly. The information provided by a HOIL processor makes it possible to generate the data

concordance and control map economically. These tools fit together smoothly because the fundamental HOS axioms are well chosen. Not only do the axioms seem obvious, but when systems are designed to be consistent with the axioms, they become easier to understand, maintain, and modify.

6.2.1 Specification Language

The HOS specification language allows a designer to describe system structure and control flow unambiguously. The description of relationships between the parts of a system is independent of how the parts are implemented. For example, information is transmitted to and from hardware modules via wires. Information is passed between software functions through the calling sequence. Finally, if work is performed by a human operator, information is passed via a display and keyboard of some sort. The specification language allows all of these information transfers to be described in the same way. This separation between specification and implementation allows the design to be completed before final partition between hardware and software, so that tradeoffs between hardware and software can be optimized over the entire design. The important concept is that the specification language provides a consistent mechanism for describing information flow and control structures. This eliminates interface errors which are not otherwise found until system integration, by which time they are costly to eliminate.

Most system specifications and designs are ambiguous. There are often difficulties in understanding the intent of the specifications, and time is lost in verifying that the interfaces are consistent. The HOS specification language is formalized so that a specification can be checked automatically for consistency before implementation. This eliminates inconsistencies in the specification which are usually not found until well into the implementation. In addition, as modules are implemented, their interfaces are compared with the specification.

The specification language does not ensure that the modules which implement the specification will do what was originally intended. It does, however, make it possible to verify interfaces between parts of the system, and ensures that all portions of the specification use a common syntax. The purpose of the specification language is to minimize confusion by ensuring that all parties express their part of the design in the same language and that each part of the system is consistent with the rest of the system. Because the HOS specification only defines

interfaces between parts of the system, it is not possible to automatically generate the system from the specification. It is, however, possible to add module definitions to the interface specifications, and have the specification evolve into the final system.

6.2.1.1 Control Structures

In present design practice, complex problems are divided into smaller problems, each of which is defined as a new problem. The division process continues until the problems are small enough to solve. Hopefully, the aggregation of the small solutions solves the original problem. In order for this to work, it is necessary to define problems and their solutions so that the solutions fit together properly. One of the functions of HOS is to monitor the process of dividing problems to ensure that interfaces between parts of the solution are correct. The HOS axioms define a set of valid methods of dividing big problems into small problems so that the solutions work together. In other words, HOS formalizes present intuitive design practices so that system components work reliably and predictably.

Solutions which collectively solve a larger problem can be thought of as occupying a common hierarchical level which is subordinate to the level occupied by the larger problem. The larger solution may be part of the solution of an even larger problem, and so on. This hierarchy is the result of the decomposition of functions into simpler functions. The function decomposition is carried to the point where the functions can be implemented easily.

The lack of standard terminology for parts of a solution causes a great deal of confusion between solutions to problems and the means used to implement the solutions. Mechanisms used to solve problems are referred to as subsystems, modules, components, functional modules, subroutines, functions, or whatever, depending on the perspective of the persons responsible for the solutions.

For the sake of defining a precise term, "function" refers to any problem solution, no matter how the solution is implemented. A function may or may not be made up of subordinate functions. The defining characteristic is that a function is given information, operates on this information, and gives back other information.

It is common to refer to a "large" function as a "system". The formal HOS terminology does not draw distinctions between functions which are made up of subordinate functions and functions which are not. However, it is satisfying to have a special word which refers to the topmost function and it is common to refer to this function and all of its subordinate functions collectively as a "system".

The relationship between functions is called a "control structure". The control structure describes information flow between functions and determines when each function is invoked. When the control structure is drawn out so relationships between functions are visible, the resulting picture is called a "control map". The HOS specification language is really a means of describing a control structure. This description lists the input variables and output variables for each function. A function description, of course, may consist of a number of subordinate functions. The specification analyzer enforces rules based on the HOS axioms which restrict the relationships between functions so that the interfaces are correct. The specification analyzer examines the function descriptions and maps the specified system. The map generated by the specification is later compared with the map showing relationships between the functions as they are implemented. This highlights differences between the specification and the functions which are actually implemented.

An initial high level specification contains only a few functions, and it is relatively easy to keep track of interfaces at that point. As the design evolves, however, more and more functions are added. Current system specifications are checked manually at several points in the design cycle, but this is too costly to be done at each stage of refinement. The automatic HOS analysis may be repeated as often as necessary to ensure that the specification is consistent at each stage before work starts on the next stage. This ensures that errors are eliminated at the earliest possible time. The analysis not only verifies that the specified information flow is appropriate, but it ensures that the overall control structre is valid.

6.2.2 Higher Order Interface Language (HOIL)

The HOIL provides the means by which the software portion of the control structure is implemented. The purpose of any language is to allow its users to communicate their thoughts to other parties. Computer language processors are the intermediaries by which the programmer's

thoughts are rendered in a form which is comprehensible to the computer. Compilers and assemblers examine statements expressing the desires of the programmer in order to generate computer code sequences which perform the requested functions. Language processors not only generate code to operate on the data types supported by the language, but in addition generate code for subroutine linkage, parameter passing, task invocation, and the like. In general, the more code sequences that are generated automatically, the easier a language is to use. Furthermore, the more computer-dependent details that are handled automatically, the less the programmer has to know about the specific machine, and the easier it is to transfer programs to other computers.

A Higher Order Language allows programmers to express operations on variables but demands that variable usage in each module be consistent with variable use in other modules. That is, HOIL statements which manipulate variables can be located by the interface analyzer, which is aware of global considerations regarding variable usage, and verifies that such usages are consistent. Therefore, the HOIL not only generates code sequences to operate on the variables, it also generates the information required to verify that only appropriate operations are requested.

Some compilers enforce consistent variable use within individual modules. FORTRAN compilers, for example, automatically convert variables to a common type before performing arithmetic. However, this does not prevent a particular variable from being treated as integer in one module, real in another, and complex in a third. Such inconsistent variable usage usually causes errors, and such errors are often extremely costly to locate. It would be cost effective to eliminate them on a global basis using a HOIL and an interface analyzer.

Past efforts at making better software development tools have assumed that these tools would emerge as additions to compilers and augmentations of compiler languages. A common objection to the use of a compiler to generate code for limited memory flight computers is that the code generated by a compiler is usually less efficient than code generated by hand. As will be shown below, hoever, it is possible to obtain the benefits of a HOIL by augmenting an assembly language.

6.2.3 Interface Analyzer

A control structure which conforms to the HOS axioms is analogous to a well-ordered military hierarchy. The design of the chain of command assures that orders are accepted only from an immediate superior and are transmitted only to the subordinate or subordinates responsible for their execution. Orders or information which bypass the chain of command are frowned upon. Such orders may be tolerated in a hierarchy composed of humans, because people are sufficiently adaptable to work out what to do under unexpected conditions. Functions, however, cannot deal with the unexpected and therefore have no capability to "muddle through". In a system structured according to HOS rules, no function may be activated except in response to requests from above. No function may access information unless the right to use it is transmitted from above, and all output information is returned only to the authorized function.

Just as people in a hierarchy are tempted to try to obtain information or exercise influence outside of channels, programmers are tempted to "improve" software systems by distributing information in unauthorized ways. Human functions are somewhat self-directing, and if their input changes format without notice, they can often figure out what to do. While unexpected changes merely inconvenience the human hierarchy, they are disastrous in a software system. One of the most common unauthorized software interfaces is for one subroutine to take advantage of the fact that critical data are left in certain registers by another routine. This saves a few machine cycles loading and storing the variables, but the interface between the two routines is nonstandard. When a maintenance programmer changes the register usage in the first program, the other program blows up. This rule violation greatly increases maintenance costs.

Checking the interfaces in a system is analagous to examining a chain of command and ensuring that all relationships between members of the hierarchy are correct. Not only must the communication paths be appropriate, but the messages transmitted on these channels must be germane. Only necessary information may be passed to a function, and a function may only return information needed by its superior. The HOS interface analyzer examines relationships between functions and verifies that they correctly implement the control structure defined using the specification language.

In a software system, functions are often called subroutines, and information is passed between subroutines via calling sequences. The interface analyzer examines calling sequences to determine the hierarchical relationships between subroutines and the information which is passed between them. This illuminates the control structure which has been implemented by the collection of subroutines. The control structure realized by the subroutines is compared to the control structure specified in the design, and discrepancies are noted. This allows HOS users to be sure that a collection of subroutines actually implements the intended control structure, and assures managers that the programmers have followed the rules.

6.2.4 Control Mapper

The mapper uses information obtained by the interface analyzer to draw a diagram showing the relationships between functions. This hierarchical diagram contains basically the same information as the specification, but is derived from the functions as they are implemented. This makes the system structure visible, so that it can be compared with the intended structure. It is possible, of course, that discrepancies may result from design changes caused by unforeseen implementation difficulties. In this case, the specification is changed to reflect the realities of the implementation, but in either case, it is necessary to resolve any discrepancies.

The information in this version of the control map is somewhat redundant with respect to the control map generated by the specification language analyzer, but it is directed at a different audience. The specification control map clarifies functional relationships for the designer before implementation. The function control map illustrates functional relationships which have been implemented using the HOIL.

Past efforts at generating system structure diagrams have attempted to derive complete flow charts of each function by examining the code. This has proved to be very difficult in the general case. Systems structured according to HOS principles, however, are much easier to diagram than systems which do not follow the rules. Furthermore, HOIL statements make the specifications of the relationships between functions more visible, which simplifies the mapping task significantly. As a result, the HOS control structure mapper is able to illustrate the system structure clearly and accurately.

6.2.5 Concordance Program

The task of the concordance program is to list all modules which affect data in the system. This information is obtained from the interface analyzer. The concordance is not intended as a design aid, although it can be useful during system implementation. The concordance is most useful during maintenance, when a programmer who is not familiar with the overall system must quickly isolate the effects of a proposed change. The concordance helps maintenance to be carried out with minimal knowledge, and minimizes the probability that a change will have unexpected side effects.

The importance distinction between the HOS concordance and past concordance programs is that the HOS facility follows data, not names which are used to refer to data. In most software systems, the names by which one routine refers to data which it passes to a subroutine are different from the names by which the subroutine refers to the same data. Because it has proved difficult to coerce or persuade programmers to use consistent variable names, it is necessary that the concordance program track the renaming of data across module interfaces. As shown in the example below, the HOS concordance lists fewer separate variable names because it combines references to the same datum.

In addition to listing references to data which are passed between functions, the concordance generator processes data which are internal to a function. When data are internal to a function, concordance generation is simply a matter of noting the line numbers of all statements which refer to the name of the internal data.

6.2.6 Narrative Documentor

It is not uncommon for documentation to become separated from the code which implements the functions. Even when the documentation is preserved, it usually is made obsolete as the system changes. The HOS documentor automatically collates available documentation with the program listing of each function, and flags functions which have been updated without corresponding documentation changes. The documentor has access to a library containing abstracts of each function and descriptions of each datum. The documentor processes a function after it has been processed by the interface analyzer and the concordance generator. At this point, all data and all functions referenced by the function are known, and the documentor appends the appropriate abstracts to the function listing. This generates some redundant information,

because data descriptions are repeated whenever data are referenced, but a maintenance programmer modifying a function finds all pertinent information in the same place.

Another important advantage is that data descriptions and function abstracts are maintained in the same data base as the source code. Thus, whenever a function is changed, the documentor is aware of the change, and signals that the documentation must be brought up to date. As variables are added, the documentor demands that their descriptions be added to the abstract file. At any time, management can interrogate the documentor to determine how many functions have been changed without corresponding documentation updates, and how many variables are undocumented. This encourages programmers to keep the documentation current. Not only does this eliminate the usual frantic documentation rush at the end of the project, but documentation is actually available during the project. It has been found that system documentation can be quite useful during a project, especially when it proves necessary to replace project personnel. New staff members can make use of the documentation to familiarize themselves with the system.

6.2.7 Example

The following example illustrates the application of HOS techniques to a very simple system. The example shows how the tools are used, and what their outputs look like.

6.2.7.1 Development of the System Specification

As was stated above, the specification language is concerned mainly with relationships between functions. Detailed function descriptions are deferred until the functions are implemented. Because function details are unimportant at this stage, the specification language requires only that inputs, outputs, and hierarchical relationships be defined.

For example, a function may initially be specified as

$$Y = F(X).$$

The input variable of function F is X, and the output variable is Y. This is a sufficiently detailed definition of the function for the purposes of the specification language. It may later be decided that function F should be implemented with two subordinate functions, F1 and F2. These subordinate functions are invoked from F, and passed

the data they need. The structure then looks like

$$Y = F(X)$$

$$Y = F2(G) \qquad G = F1(X)$$

Function F has been decomposed into two subordinate functions but can be treated as a single entity if the level of detail provided by F1 and F2 is not needed. The specification language analyzer observes that F1 and F2 do not use variables which cannot be passed to them by their parent F, and that they do not alter data which their parent is not authorized to alter. In addition, the specification analyzer notes that temporary variable G is created by one function and used by another on the same level. The specification conforms to the axioms, and is passed.

At a later point in the design cycle, function F1 is further broken down. The specification is altered to look like this:

$$Y = F(X)$$

$$Y = F2(G) \qquad G = F1(X)$$
$$G = F4(P) \qquad P = F3(X)$$

The analysis is repeated to ensure that the change has not resulted in an improper specification, and the design is complete.

In order for implementation to begin, the designer must describe the functions and the variables to the implementation staff. The logical way to do this is to write the function descriptions and put them in the abstract file for the documentor. Thus, the function and data descriptions are available from the beginning of the project, and are added to the source listings from the beginning of implementation. This helps the programmers keep the purpose of each function in mind as it is implemented.

6.2.7.2 Use of the HOIL

The most important characteristic of the HOIL is that interfaces and variable use are illuminated for analysis by other HOS tools. Therefore, the following illustration shows only sample interface statements. The interface statements required for a real system may be somewhat more complex. This example does not show the process of verifying that data passed between functions is treated consistently. In a real situation, it is also necessary to inform the HOIL of all variable types in order to allow the interface analyzer to verify the use of the variables.

```
ENTRY(F)  INPUT(X)  OUTPUT(Y)
..
CALL(F1)  INPUT(X)  OUTPUT(G)
..
CALL(F2)  INPUT(G)  OUTPUT(Y)
..
Y = ...
RETURN
END(F)

ENTRY(F1)  INPUT(P)  OUTPUT(Q)
..
CALL(F3)   INPUT(P)  OUTPUT(M)
..
CALL(F4)   INPUT(M)  OUTPUT(Q)
Q = ...
RETURN
END(F1)

ENTRY(F2)  INPUT(R)  OUTPUT(M)
..
M = ...
RETURN
END(F2)

ENTRY(F3)  INPUT(L)  OUTPUT(M)
..
M = ...
RETURN
END

ENTRY(F4)  INPUT(M)  OUTPUT(N)
N = ...
RETURN
END
```

Although there are very few functions implemented, and only the interface statements are shown, the collection of listings obscures the control structure. Even though the HOIL interfaces are easy to examine, the control structure is not nearly as easy to see as it was in the original specification.

The data flow is even harder to follow, because the programmers have not used the same names to refer to the same data. In particular, variable name M refers to one datum in function F2 and to an entirely separate datum in functions F1, F3, and F4. Manual analysis is quite difficult for small systems, and impossible for large systems.

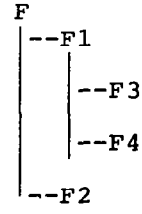
6.2.7.3 Interface Analysis

These HOIL statements are examined by the interface analyzer. The analyzer finds the ENTRYs and CALLs which tell it which variables are involved in the interaction between the modules. The analyzer notes that F has the proper number of inputs and outputs, and that these are passed to the subordinate functions of F. In addition, although this example does not show the data type declaration statements, the analyzer verifies that the data transmitted and received by the various functions are of compatible types.

The programmers responsible for implementing F1 and F2 have chosen different names for their references to variables X, Y, and G. Depending on the managerial philosophy of the project, the interface analyzer can either follow the name change or demand that the subordinate functions use the names defined in the specification. In either case, the analyzer finds that function F calls on two subordinate functions F1 and F2, using variables X, Y, and G, as called for in the specification. This verifies that the implementation of the interfaces between function F and the rest of the system is consistent with the specification. Further analysis is performed on the other functions, and they pass depending on whether or not variable renaming is allowed.

6.2.7.4 Control Mapping

Because the HOIL implementation of the functions has obscured the relationships between functions, the control mapper is used to re-illuminate the control map. In this case, the map looks like this:



This notation clearly illustrates the relationship between the various modules. A more complex system would produce a more extensive map, of course. This notation is somewhat more compact than the original specification, and allows larger control maps to be printed on a single page. The compact format is advantageous because this version of the control map is likely to be generated often as implementation proceeds.

6.2.7.5 Concordance Listing

The HOS concordance shows all data and where they are used. Previous programs have generated global variable reference tables, but they lack the capability of tracing data under different names through module interfaces. In order to highlight the differences between a conventional concordance and the HOS concordance, examples of both are presented. Although a real concordance would provide the locations within each function where data were referenced, these statement numbers are omitted from the examples for clarity. The conventional concordance for this example would look like:

Variable Uses

G	F Stored	F Referenced		
L	F3 Referenced			
M	F4 Referenced	F1 Referenced	F2 Stored	F3 Stored
N	F1 Stored	F4 Stored		
P	F1 Referenced			
Q	F1 Stored			
R	F2 Referenced			
X	F1 Referenced			
Y	F1 Stored			

This conventional concordance is not very useful because programmers have re-named variables at the interfaces to the various subroutines. This makes it impossible to determine which routines manipulate which data. The HOS concordance program has access to the output from the interface analyzer. The analyzer traces data through the various names used in the different modules. The HOS concordance is much shorter, because there are fewer data than names, and the HOS concordance has an extra column, giving the names which refer to the data in each function. This illustrates clearly that variable M in function F2 is not the same variable that is referred to in functions F1, F3, and F4.

Name	Datum Name	Vble References					
G	G	F	Stored	F	Referenced	F2	Referenced as R
		F4	Stored as N	F1	Stored as Q		
P	M	F1	Stored	F3	Stored	F4	Referenced
X	X	F	Referenced	F1	Referenced as P	F3	Referenced as L
Y	Y	F	Stored	F2	Stored as M		

There are fewer variable names in the HOS concordance. This is because there are really only four data- X and Y, which are the input and output variables of function F, and G and P, which are the temporary variables which pass data between the subordinate functions of F and F1. The datum name is obtained from the specification. The variable name is assigned by the highest level function which refers to that datum, no matter what names are used by other functions to refer to the same datum. This concordance makes it possible to follow data use across interfaces which may not use the same names to refer to the same data. This information is vital to maintenance programmers. In order for the programmers responsible for individual modules to be able to tell which datum their names refer to, the narrative documentor adds the system name to the variable list generated for each module.

This simple example illustrates the complexities of tracing data use manually, and makes the 73% interface error figure from APOLLO quite understandable. The HOS data flow tools eliminate these errors by clarifying the effects of variable references and name changes. It might be added in passing that as long as the interface analyzer can isolate references to the original data names, it would be perfectly possible for it to edit programs so that they refer to the common set of names. This might be deferred until after system delivery, in order not to unduly ruffle the programmers.

6.3 Implementation

The HOS rules have been shown to be effective in past software development efforts. It is possible to verify the control structure and module interfaces manually, but this is a costly process. It is much better to automate the required analyses. The proposed interface analyzer and HOIL are implemented as a single tool which is specifically tailored for the fault-tolerant multiprocessor. The specification

language and control mapper are much more general purpose facilities, and are therefore not treated further in this report. The other tools, however, are specific to the multiprocessor and are described at much greater length below.

6.3.1 Implementation of the HOIL and Interface Analyzer

The proposed HOIL is a low cost flexible extension of the assembly language for the processors. The HOIL provides facilities for describing interfaces between subroutines and for specifying how variables are to be used. These interfaces between modules are analyzed to verify that all interfaces are consistent, and that all variables are referenced correctly. These verification facilities and interface specification facilities, when added to the assembler, produce a HOIL and interface analyzer designed specifically for the computer.

The HOIL is implemented as a pre-assembly macro processing step. Macros are written to generate assembly language code for all necessary inter-module interface functions. As each interface macro is written, the analyzer is extended to verify its use. HOIL and interface analyzer development are incremental processes, which can be carried as far as necessary to support a particular application.

Whenever a variable appears in an interface macro, the analyzer checks its type to ensure that the function being called treats it in the same way. All system data used in a function are declared as part of the interface expected by the program, and references to global variables which are not listed are flagged.

In addition, whenever a variable name appears in the program, the analyzer checks to make sure that it is being used in a manner consistent with its type. In order to allow this, macros are written to generate assembly code to perform arithmetic operations. Normally, when variables participate in arithmetic operations, the operational logic is obscured when the assembly language instructions for loading registers, computing, and storing the result are coded individually. However, the macro "=" allows such operations to be coded as:

```
A = B + C
```

The algebraic form of the expression is much easier to understand. Also, the "=" macro processor checks the specifications of variables B and C, and complains if they are not appropriate variables to be added

together.

An advantage of this approach is that if it is later decided to add special purpose hardware or microcode to the computer, the macro definitions may be changed to generate code to take advantage of the new facility. This will not require changing the programs. In fact, if the computer is provided in several versions which have different facilities, the same programs can be assembled for different versions by using different macro libraries..

6.3.1.1 Steps in Development of the HOIL and Interface Analyzer

A HOIL processor generates code for a specific computer, and must be designed for the computer being used. The following steps are needed to develop the HOIL described above:

1. Examine the selected computer architecture and determine how interfaces between modules are to be implemented. These functions include such facilities as CALL, Define Data, Receive Data, Fork, Join, Wait, Post, and the like.
2. Select an appropriate macro processor. There are many suitable macro processors available.
3. Code macros to implement the necessary interfaces.
4. Write a program to scan source code for occurrences of interface macros and check them for consistency.
5. Expand the function of the interface checker to verify the use of variables.
6. Write a program to print the results of the interface analysis as a data concordance.
7. Write a program to print the results of the interface analysis as a control map.

Powerful general purpose macro processors are available at CSDL, and can be used without further development. As it turns out, the developers of the macro processors which are in use at CSDL were greatly concerned with the portability of the macro processors themselves. They wrote the macro processors in such a way that it is easy to install and run them on different machines.

6.4 Justification

This section briefly explains how assemblers and macro processors work in order to justify the above assertions concerning the practicality of developing a HOIL based on macro processors.

6.4.1 Character String Mapping

It is possible to regard the operation of all computer language processors as accepting an input character string and generating an output character string. This process is often referred to as "mapping". A compiler input string is the source program and the output string is the object code which runs in the computer.

It is customary to think of the source program in terms of individual language statements. It is possible instead to think of the program as one long input string which is divided into separate statements according to some convention, such as the end of a card. In PL/1, for example, the string form of the program is made explicit, because a special character is needed to end a statement.

It is also possible to think of the object code as a character string. Computer memory can be subdivided into characters, and a program in core can be thought of as one long string of such characters. The most striking difference between source and object strings is that object code strings do not make sense to most human readers. It is interesting to note that the computer's string is usually much shorter than the human's.

6.4.2 Assemblers

The function of the assembler is to map a string which a human can read into a string which a computer can read. This mapping process is straightforward. Assemblers essentially process three kinds of data: labels, opcodes, and operands. Labels are used to refer to specific points in the program. In order to be able to use the labels, the assembler must first examine the program to learn where they are. As each label is found, it is put in the label table along with the current offset from the beginning of the program. This offset is the value of the label, and the label defining pass is normally referred to as PASS1. After defining the labels, the assembler makes PASS2. This time, it looks up each opcode in a table, and learns the numerical opcode for it and the number of operands to expect. It then looks up the operands in the label table and determines the numberical value of each. Finally,

the assembler places the value of the opcode and the values of the operands in the output string.

6.4.3 Macro Processors

A macro processor implements a somewhat more complex mapping function than the assembler. The number of opcodes in an assembler is usually defined in advance, and the table is built into the assembler. A macro processor, on the other hand, allows different sets of maps to be defined by the user. The definitions of these maps are called macros. The collection of macros defines the overall map from input string to output string.

A macro is defined by first giving a sample of each recognizable input substring followed by the output substring to which it is to be mapped. The macro processor searches the input string for substrings for which a macro has been defined. Input substrings for which there is no match are added to the output string without change, while input substrings for which there is a match are replaced by the specified output substring.

The process of generating the output substring in response to a specific input substring is somewhat complex. The macro definition not only includes characters which are to be added to the output string but instructions to the macro processor itself. These instructions may cause the macro processor to remove additional substrings from the input string and copy them into the output substring in specific places. For example, the assembly language statement for a subroutine call may be GOSUB followed by the name of the subroutine. The macro definition for a simple CALL macro which uses this instruction follows:

```
MACRO
CALL &PARAMETER
GOSUB &PARAMETER
MEND
```

The MACRO and MEND statements delineate the beginning and end of the macro definition. The first line describes the input substring which is to be mapped to the output substring as the characters "CALL" followed by any parameter. This parameter is substituted into the output string whenever it is referenced. The second line tells the macro processor that the output substring to be associated with the input substring "CALL name" is generated by adding "GOSUB" to the output

substring followed by "name". Thus the string "CALL name" is replaced by the string "GOSUB name". Because CALL does not appear in the output string, the assembler does not find it, but encounters GOSUB instead.

A common use of macro processors is to implement extended assembly language functions. A particular computer environment may have a standard set of instructions which are used to pass data between subroutines. It is tiresome to code this process repeatedly, and there is little point in debugging each subroutine calling sequence individually. In this case, the CALL macro would map into the series of assembly language statements needed to pass arguments to the subroutine. This not only saves time and eliminates errors, but makes the program more readable. It is not necessary to scan all the statements in the calling sequence to realize that a subroutine is being called, because the CALL is easily identified.

6.4.4 Interface Checking

It is not possible to check interfaces automatically in most assembly language programs because the interface logic is obscured when individual assembly language statements are coded. When macros are used to implement interfaces, the interfaces are much easier for programmers and interface analyzers to find. Automatic interface checking and variable use verification are made possible by this enhanced program legibility. Because the macros for calling subroutines, accepting data from subroutines, and other interface functions are easily located, they can be checked for errors. In principle, the interface analyzer scans all the programs and checks to see if the interfaces are compatible. However, it is possible to implement the interface analyzer in a considerably more elegant manner, as explained below.

Because the interfaces are specified in the control diagram, it is possible to define them to the macro processor as a special purpose macro. Thus, when the macro processor encounters an interface function macro, it not only generates the code to accomplish the interface function, but also checks the particular interface invocation against its list of interface definitions. Thus, interface validation is performed as each module is assembled. Alternatively, the interfaces to a given module can be defined before the module is coded. This allows the structure to be verified before all of the modules in it are developed. A "pseudo module" containing only interface specifications

can participate in the interface analysis, and the code can be filled in later. This allows some aspects of system integration to be completed before the modules are developed.

6.4.5 Algebraic Expression Evaluation

The mapping between CALL statements and assembly language statements to implement calling sequences is relatively straightforward. It is somewhat more complicated, but not excessively so, to examine algebraic expressions and generate code to implement them. The techniques for scanning a parenthesized expression are very well known, and require only a stack to hold temporary results. After development of the interface macros, the next step is to write the "=" macro, which generates code for arithmetic expressions. The "CALL" macro verifies that the program passing data and the program receiving data agree on the definition of each datum, and the "=" macro makes sure that variables are used properly.

As with the CALL macro, the "=" macro enhances the readability of the program. The algebraic expression may be understood at a glance, because understanding does not depend on the analysis of many assembly language statements. Furthermore, because the macro processor is made aware of what is actually being done to the variables, it verifies that the operations are appropriate. Because there are many algebraic expressions in flight code, this facility is extremely helpful to flight programmers.

6.4.6 Generation of Control Functions

Once the burden of algebraic statements has been removed from the programmer, the major remaining burden is the development of control functions. Recent experience indicates that such statements as IF, THEN, ELSE, CASE, and DO are very useful and provide adequate program clarity. It is possible for a macro processor to generate code for such sequences.

In explaining this process, it is helpful to re-examine the evaluation of parenthesized algebraic statements. A statement of the form

```
2 * (a + b)
```

requires that the multiplication be delayed until evaluation of the expression inside the parentheses. The method for evaluating such

expressions involves the use of a stack to hold data describing deferred operations while more immediate ones are performed. In the example above, when the "(" is encountered, the "2" and the "*" must be pushed on the stack. They are removed when the ")" is encountered.

The process of unraveling a nest of IF's, THEN;s, DO's etc. is essentially similar, provided that the entire source program is regarded as a continuous string. The macro processing the string defining the algebraic expression above pushes the "2" and the "*" when it encounters the "(". The "(" signals that the preceding operation must be deferred. Similarly, the macro processor finding a "DO" statement after a "THEN" stores the "THEN" for processing after it finds a matching "END". The reason the "THEN" must be stored is similar to the reason why operations before a "(" must be stored. When the group of statements bracketed by "DO" and "END" is to be executed, there is no difficulty in passing control directly to the first statement of the "DO". When the statements must be bypassed, however, the code generated in response to the "THEN" must branch around the "DO" group. The conditional test generated by the "THEN" statement branches to a statement label located after the "END" if the test fails. Therefore, the macro processor must remember to generate an appropriate label when it locates the "END".

The key point is that once the program is regarded as a string, it is possible to treat individual statements within it in the same way that individual variables are treated within an algebraic expression. This method of dividing the source code into small pieces closely resembles the process of breaking a system down into functions. The analysis of the source program consists of removing the next "thing" from the input string, and then chopping that "thing" into smaller and smaller "things" until an irreducible "thing" that can be processed is found. "DO" is the equivalent of "(" and causes items to be added to the stack. "END" is the equivalent of ")", and causes removal from the stack. Looked at in this way, the processing of nested DO's etc. reduces to the previously solved case of evaluating algebraic expressions. Just as a "(" defers performance of an arithmetic operation, a "DO" defers assignment of a control label. Processing control statements requires additional "special characters" and they are handled in different ways, but this is a difference of degree rather than of kind.

The point of this argument is that macros can be defined to generate code required to implement IF, THEN, and DO functions. These

macros make it unnecessary for the assembly language programmers to generate individual assembly language statements to perform these functions, just as the CALL macro makes it unnecessary to write the code to pass parameters. This reduces the programming effort needed to implement a given set of software functions. Once these macros are implemented, programs written in the augmented assembly language look surprisingly like programs written in compiler languages.

It must be noted that this does not imply a proposal to implement a specific compiler language such as PL/1 or a PL/1 subset. Existing languages were not designed with HOS structures in mind, and most allow the programmer to express such complicated constructs that compilation is difficult. The HOIL syntax to be developed restricts the use of keywords such as DO and THEN so that the macro processor can easily parse the program. Furthermore, the HOIL syntax is designed to illuminate the interface specifications for the benefit of the automated analyzers. This simplifies the programs for the reader as well as for the macro processor, and make it easier to learn to use the HOIL.

6.5 Conclusion

From the point of view of the user, the augmented assembler language at first sight looks like a compiler language. However, programmers can always use assembly language code when necessary to perform functions which are not supported by the macros, although assembly code is not subject to automatic verification. This provides a great deal of flexibility in the use of the HOIL. The justification for assembler use can be made on a case-by-case basis, rather than condemning the entire software development process to the use of an assembler because occasional routines require it. Furthermore, if a programmer discovers a better code sequence than a macro generates, the macro can be changed to generate the more efficient code, so that all programs benefit, and macros can be written to perform specialized functions needed by the multiprocessor software. This further reduces the need for assembly language.

The macros force interface functions and operations on variables to be specified in a highly visible manner. This makes the programs more readable, which cuts implementation and maintenance costs. The increased visibility also makes it possible to verify interfaces and the use of data in arithmetic expressions automatically, thus eliminating the major cost in final system integration. Because it is not

possible to verify pure assembly language statements automatically, the HOIL flags assembly statements so that the programmer can justify them to management.

The development of the HOIL and the interface analyzer can proceed incrementally. If the initial interface functions are found to be inadequate for a particular application, more macros can be written to implement additional functions. This neither invalidates previous macros nor affects the analyzer's processing of previous functions.

Developing systems according to the HOS axioms leads to lower life cycle costs. System integration costs less because interface errors are eliminated even before modules are coded. Maintenance is cheaper because the system is easier to understand and because the analyzers verify that changes affect only the modules which are changed. If module interfaces change, the analysis can be repeated to illuminate the side effects of the change. All of these facilities can be implemented using existing macro processors. Each macro written in developing the HOIL can be individually tested and debugged. Therefore, the development cost of the HOIL can be measured in man-months rather than the usual man-years or man-millenia required for compiler development.

6.6 Application to Software

The fault tolerant flight computer system can be regarded as a series of environments which support increasingly complex software systems. The lowest level environment is provided by the system hardware itself. The hardware supports a number of microcoded functions which provide common facilities and are used by the rest of the software. The executive software controls task dispatching, resource allocation, and reconfiguration in response to faults. The executive supports the applications software, which is not affected by the fact that the flight computer is tolerant of hardware errors. Each level of software supports the adjacent level, and there are many tradeoffs between levels. For example, the more functions are implemented in microcode, the fewer instructions are needed to perform a given task, but the more microcode memory is required for each processor. One of the major advantages of the HOS approach is that the detailed tradeoff between the various levels of software can be made after the system is designed. This is because a given function is specified in the same way, no matter how it is implemented. Because interfaces between functions are specified using macros, functions may be moved from

software to microcode without affecting the programs which use the functions. The macros must be changed to generate different code, but the source program itself does not change. In order to illustrate the common specification of functions, the following example defines the HOS specification for the task scheduling portion of the multiprocessor executive and part of the reconfiguration routines. It is important to note that the functions are described in the same way, no matter how they are implemented.

6.6.1 Software Example

The scheduling functions in the executive are collectively known as SYSPROM. The main purpose of SYSPROM is to ensure that a scheduled task is performed at the correct time or in response to the appropriate event. The overall scheduling function can be described as

Task Done = f (Task Scheduled)

Because a task is done only after it has completed its function,

Task Done = g (Active Task)

but it does not become active until it is ready, so

Active Task = h (Ready Task)

Likewise, a task must be scheduled before it can become ready, so

Task Ready = j (Task Scheduled).

These specifications are illustrated in Figure 6.1. Note that it is not necessary to define any of the functions f-j. The specification and the control structure are implementation independent.

Interfaces should be checked at this point. Input can be traced down the tree, and output can be traced up. Intermediate outputs and inputs can be traced on the same level. Because the axioms have not been violated, Figure 6.1 represents a valid HOS control structure.

Function j involves some kind of delay mechanism between the scheduling of a task and the time at which the task is to be performed. Thus, j decomposes to

Task Ready = k (Task Waiting) Task Waiting = l (Task Scheduled)

This breakdown of j is shown in Figure 6.2. An alternate decomposition of l is shown in Figure 6.3, which is entirely equivalent to Figure 6.2. It is also possible to express all control structures as binary trees as in Figure 6.4. In general, the binary structure is much

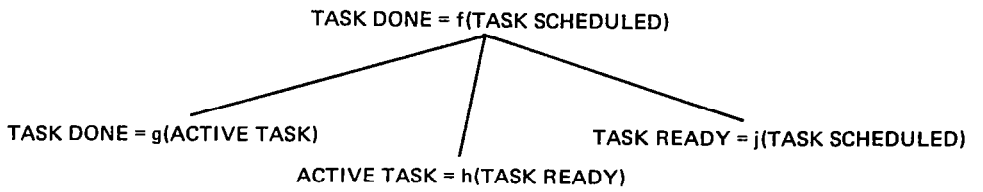


Fig. 6.1 First Level of Decomposition.

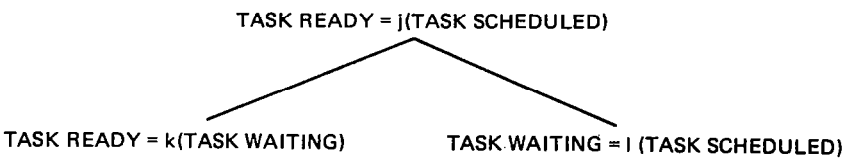


Fig. 6.2 Second Level.

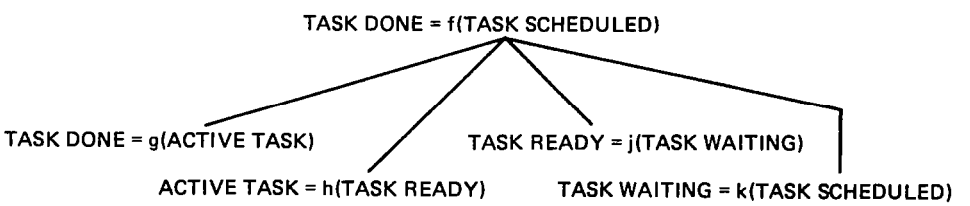


Fig. 6.3 Alternate Structure.

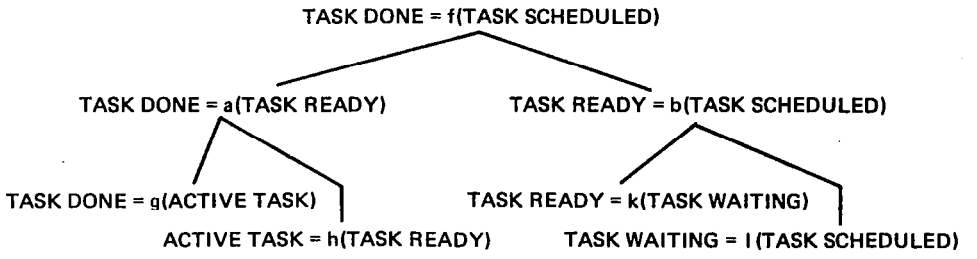


Fig. 6.4 Binary Tree Structure.

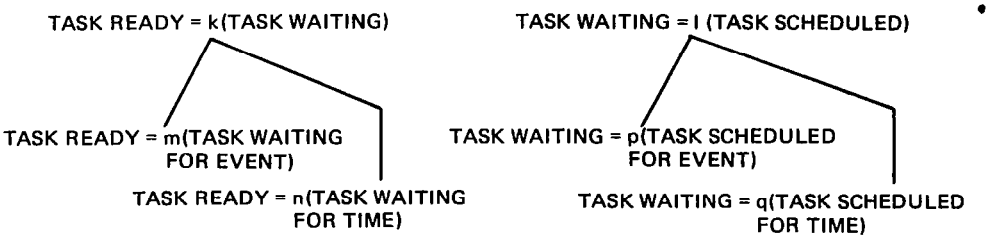


Fig. 6.5 Further Refinement.

clearer and lends itself to greater simplicity of design. The extra levels of refinement do not require additional work during implementation because implementation is performed only at the lowest levels of the tree.

Up to this point, the scheduler diagrammed in Figure 6.4 schedules tasks only as a function of time. The specification is augmented as shown in Figure 6.5 to accommodate event scheduling as well. Examination of the total structure defined by Figures 6.4 and 6.5 shows that the input and output of the specified functions are consistent with the HOS axioms, and that the control structure is also consistent.

The decomposition process is carried to more and more levels until the design reaches a point where the lowest level functions can be implemented easily. The cutoff point for decomposition may be determined by trial and error during implementation. If the implementation of a function proves to be excessively complex, another level of decomposition is called for. On the other hand, if implementation of a function is so simple that the interface statements occupy a major portion of the program, it is wise to implement directly at a higher level.

It is important to note that identical considerations apply whether the system is implemented in hardware or in software or in some combination of the two. The lowest level functions may be implemented as software subroutines or hardware modules without affecting other modules. This allows many implementation decisions to be deferred until the design is complete, and ensures that options are kept open as long as possible.

HOS Axiom Table

This table states the HOS axioms in an informal manner. Derivations of theorems based on the axioms are contained in [5]. These theorems motivate the rules for function relationships which are incorporated in the interface analyzer and specification analyzer.

Axioms, Formal Defintion

1. A function controls the invocation of the set of valid functions on its immediate, and only its immediate, lower level.

2. A function is responsible for its own and only its own output.

3. A function controls the access rights to each set of variables in the output space for each immediate, and only immediate, lower level function.

4. A function controls the access rights to each set of variables in the input space for each immediate, and only immediate, lower level function. Thus, a function cannot alter members of its own input set.

5. A function controls the rejection of invalid elements of its own, and only its own, input set.

6. A function controls the ordering of each tree for the immediate, and only the immediate, lower level.

CHAPTER 7

SEMICONDUCTOR TECHNOLOGY

The multiprocessor architecture has reached the degree of maturity where it becomes appropriate to begin generating specifications and design candidates for evaluation and production. The reduction of system failure rates through redundancy alone has a limited degree of feasibility. In order to reach the level of 10^{-10} failures per flight hour it is necessary to address the many issues for which redundancy is only a partial solution or no solution at all. The preceding chapter dealt with one of these issues, i.e. software, for which redundancy contributes a high probability of correct execution, but can not otherwise contribute to correctness. In this chapter and the next, we consider two primary issues underlying failure rate reduction. The chapter following this one examines packaging constraints and some candidate approaches. Key issues are isolation, logistics, repair, integrity, interconnection reliability, and heat transfer. Heat transfer is possibly the most crucial of these issues, because the thermal density will inevitably be high, and the failure rate of semiconductor components is critically sensitive to the ambient temperature.

This chapter is concerned with several issues related to semiconductor technology: the tradeoffs in design, the component reliability goals, and the acquisition of reliable components.

7.1 Candidate Components

To a large extent, the multiprocessor presupposes several characteristics of its semiconductor components. They must among other things be reliable, producible, available, economical, and efficient in terms of power and volume. The inherent incompatibility of many of these characteristics makes it exceedingly difficult to select among the available technologies and devices, none of which are ideal in all respects.

From a historic viewpoint, the use of massive redundancy in integrated systems has only been considered where the cost of hardware replication has been small in comparison to project cost. The emergence of the LSI microprocessor has done much to foster consideration of a massive redundant architecture for commercial aircraft. It is natural, therefore, to begin with the assumption that the LSI microprocessor is a necessary element of the multiprocessor. This assumption can be challenged on grounds of speed, testability, and failure rate. The LSI processor devices are near today's limits of complexity, and as a result can suffer from anomalous effects such as pattern sensitivities, which stem from their small and irregular geometries. They moreover are difficult to test, because relatively few of their internal signals are accessible. The faster bipolar microprocessors, to some extent, alleviate the problem, because they are partitioned onto a number of chips. Their failure rate remains an open question, however, until such time as their production volume becomes large enough to establish an assessment. The alternative approach is to use more chips in a more nearly conventional arrangement. This approach is limited by considerations of size, power, and cost.

Memory accounts for a large fraction of the multiprocessor's circuitry. LSI memory elements are generally adequate in terms of speed, are moderately testable, and in some cases have proven to have low failure rates. They are therefore viable candidates for all of the memory functions: common memory, cache memory, and microprogram stores. The volatility of semiconductor read-write memory leaves some doubt as to their utility in common memory. Core memory is a potential competitor, but does not appear to be improving as fast as semiconductor memories in cost, density, and power consumption. Moreover, if core memory were used for instruction storage, the initialization of the computer after certain maintenance procedures would require external reloading. A wholly semiconductor common memory could use programmable read-only memory elements for instruction storage and read-write elements for data. When a permanent memory module fails, it could be backed up by a read-write memory spare, although it is conceivable that dedicated spare PROM memory modules might be included for critical permanent modules.

Certain of the multiprocessor's circuits are repeated often enough that they might appropriately be produced as custom integrated circuits. The bus guardians, the bus interfaces, and the clock

receivers are good examples. An equally attractive approach is the use of hybrid sub-carriers. Both of these approaches are expensive initially with the potential of reducing the long-term total costs. Whether or not such implementations are made, the use of discrete transistors and diodes, and SSI and MSI circuitry for certain functions is inevitable. The only question is how much. Again, power, volume, and standardization place limitations on the acceptability of SSI and MSI.

In summary, then, it is anticipated that a variety of semiconductor components will be required, ranging from transistors to LSI elements. The tradeoffs that lead to selection will depend on time frame of procurement, procurement volume, computer performance parameters, and computer functional application.

7.2 Reliability Goals

In order to estimate the reliability requirements of the semiconductor elements, we begin from the premise that the computer failure rate will be of the order of 10^{-10} failures per hour. As a first estimate, we can refer to the Markov reliability model results described in Chapter 5. Table 5.3 lists baseline parameter values used for a representative calculation of failure rate, in which the basic module MTBF was taken to be 10^4 hours. For a complement of 15 processor and 9 memory modules, the computer failure rate was slightly over 10^{-10} per hour. A rough extrapolation to a somewhat larger computer (20 processors, 20 memories) and a slightly lower computer failure rate leads to a nominal module MTBF of 3×10^4 hours. The next step in the estimation process is to assume a certain number of semiconductor devices per module. In the absence of a firm design, we make the rather crude estimate that each module will require roughly two hundred semiconductor devices. If we attribute all failures to these devices, each device would require a failure rate of 0.017 per cent per thousand hours (1.7×10^{-7} per hour). In order to take into account the packaging and interconnection failure rate, we reduce the allocation to the semiconductor devices to 0.01 percent per thousand hours. This is a convenient number to deal with at this stage of the design, because it represents the low end of the range in which commercial production components usually are available after screening.

In order to achieve the 0.01 percent per thousand hour failure rate over all components, special precautions will be necessary,

involving the selection and the acquisition of components. The requirement is clearly one that can be met. Military programs such as Poseidon and Trident have done so. Indeed, Minuteman required a rate one tenth as large, and Apollo achieved one twentieth. Moreover, the technology and management required to do it is widely known. On the other hand, commercial avionics equipment has a history of performing an order of magnitude or more worse than what we desire. How this situation comes about, and how it can be resolved is not totally clear, but some insight may be gained by a brief consideration of some of the relevant issues.

7.3 Component Selection

One of the keys to component reliability is the process of choosing the components to be used. If complete freedom of choice were available, it would be possible to obtain excellent components with a minimum of effort. In practice, however, strong pressures exist to choose components that are not readily procurable with low failure rates. LSI is an example of this, where the high component density available has advantages in all respects except, perhaps, reliability. Given that component selection will necessarily involve the choice of devices that are relatively new, or not produced in large quantity, or for some other reason are not easily qualified a few guidelines are applicable in making such choices.

Table 7.1 indicates many of the failure modes that have been observed in integrated circuits. The fact that some components are available with very low failure rates testifies to the ability in some cases to avoid or surmount these problems. One obvious guideline for component selection is to choose components related to those that have been successful, in terms of process, vendor, and/or procedures for screening and burn-in.

Figure 7.1 shows a family tree of technologies, incomplete for lack of some of the newer developments. The figure helps to visualize the vast number of different processes used by various suppliers today. Each process has its own individual failure modes, each of which may require a different screening environment. The consequence of choosing components from among several of these technologies will be an escalation of the cost of procurement and reliability assurance.

The choice of LSI components is particularly difficult, because many problems are exaggerated by the small pattern geometries. Small

TABLE 7.1

IC INTERNAL PROBLEMS DETECTED PRIOR TO PRODUCTION BUYS

1. Poor metal adhesion to SiO_2
2. Underbonding
3. Overbonding
4. Al rich, Au Al eutectic formation ("purple plague")
5. Nicks and cuts in bonding wires
6. "Necking" at foot of ultrasonic or wedge bonds
7. Critical angle of wire pull (ultrasonic bonds)
8. Overly long bonding leads
9. Leads shorting to edge of chip or package lid
10. Burn out at scratches caused only by electrical testing
11. Shorts by metal smearing
12. Surface instabilities
13. PNPN switching
14. Chip cracking - pyroceram mount, eutetic mount
15. Poor chip orientation
16. Oversized bonds
17. Pinholes in SiO_2
18. Poor SiO_2 Dielectric strength
19. Misplaced bonds
20. Contamination in packages
21. Al corrosion
22. Thinning of Al at SiO_2 steps
23. Al to Si contact resistance
24. Lifted chip
25. Scratches
26. Secondary breakdown
27. Interconnect layout - geometry
28. "Purple plague" at SiO_2 steps - multimetal systems
29. Non interchangeability of "identical" circuits
30. "Disappearing" of aluminum - alloying, metal migration
31. Steeper SiO_2 steps due to photo resist change

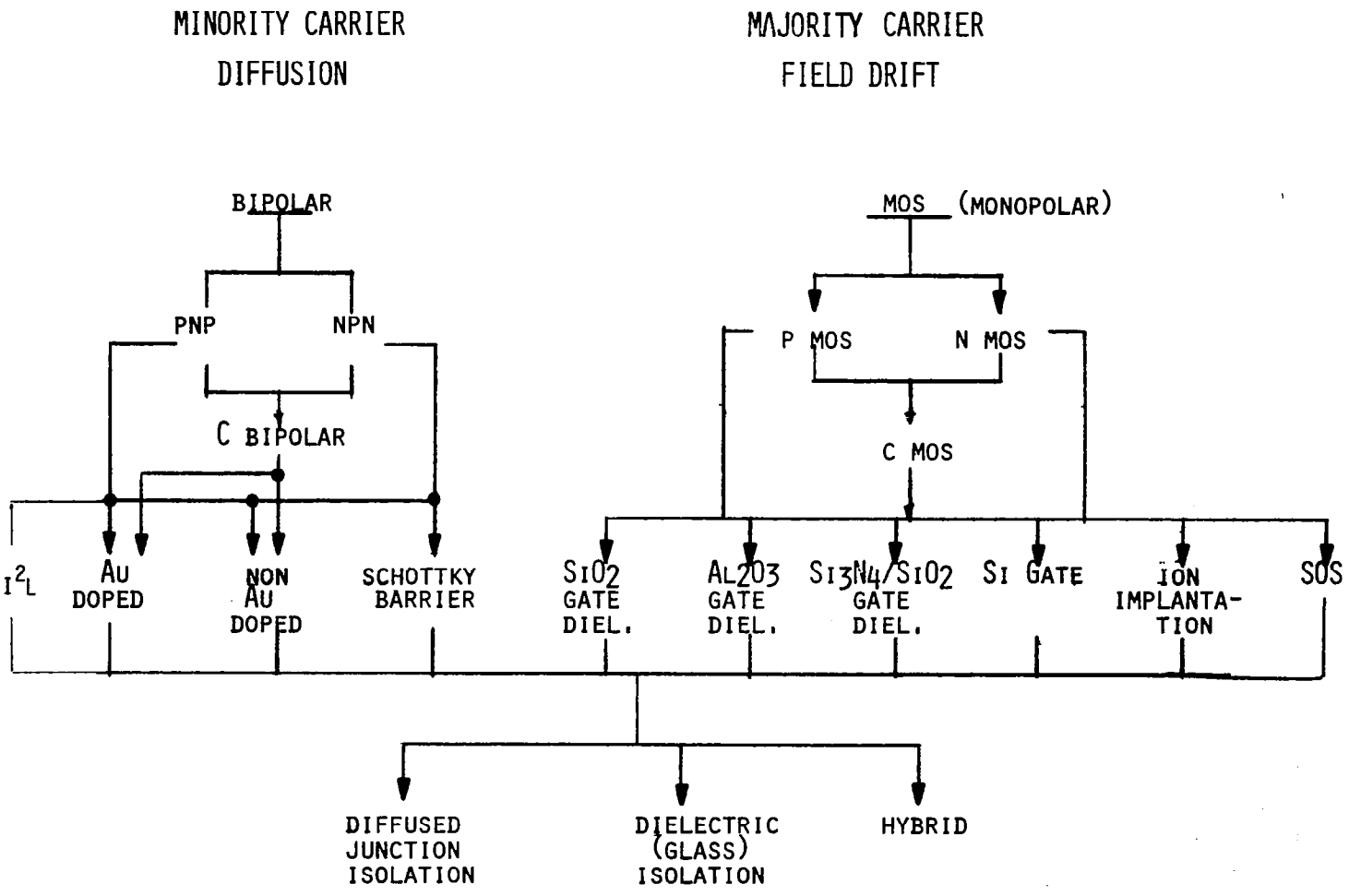


Fig. 7.1. Technology Family Tree.

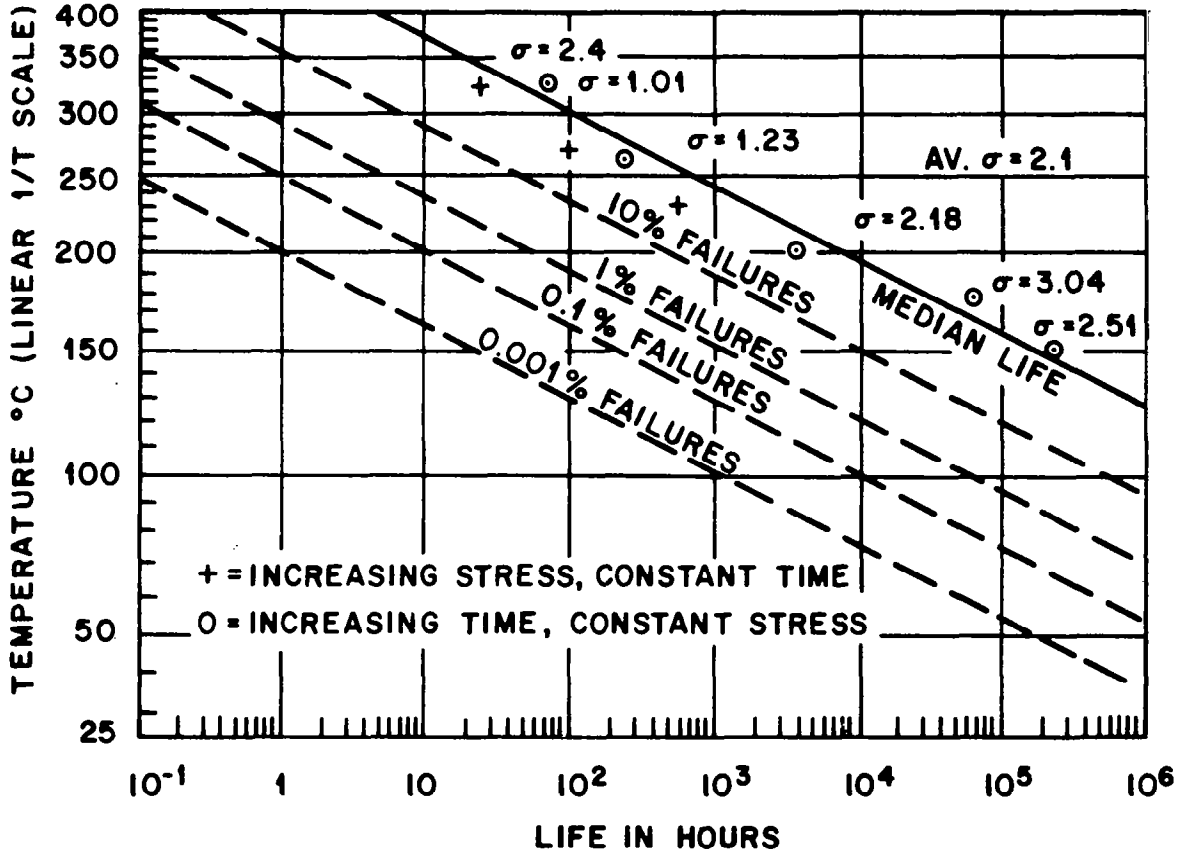
metallizations for interconnection and small spaces between interconnects enhance material migration through high current density and electric fields. Small windows in the thermally-grown oxide make etching more prone to defects, which can instill long-term failure modes. Surface instability is another problem related to the oxide in small geometry circuits. LSI also suffers, as mentioned previously, from testing problems. As a guideline for LSI device selection, one must try to avoid devices that have not matured to the point of large-scale production with relatively good yields. It is worth pointing out that where an LSI device replaces ten other devices, one can in principle tolerate ten times the failure rate. This is only true, however, for random failures. Many of the failure modes are generic, i.e. strongly correlated, which acts to negate the advantages of redundancy. This effectively means that unless LSI devices exhibit failure rates well below their functional share of the circuitry, they are apt to contribute more than their share toward system failure.

It is always beneficial to minimize the number of different components used. Wherever a single component can serve a multiplicity of functions, the degree of standardization is enhanced, and the procurement volume of the part is increased, multiplied by the number of computers times the number of like modules in each computer. This could mean tens of thousands for each instance.

As a final guideline for component selection, we mention once again the issue of temperature. Figure 7.2 illustrates the dependence of component life on component temperature. For different technologies and circuits, the curves are shifted, but the slope remains fairly constant. A difference of a few tens of degrees in temperature produces an order of magnitude difference in failure rate. Aside from motivating a conservative thermal design, these curves also provide strong motivation for minimizing power dissipation of components. This factor tends to favor the newer technologies over the better-established ones, contrary to other guidelines. The resulting dilemma is resolved in this case by making strong concessions in the thermal design of the packages rather than relying on low-dissipation components.

7.4 Reliability Assurance

Component reliability requires a dynamic life cycle rather than just a single initial specification. Inadvertent changes in production processes can introduce new failure modes, detectable only by incoming



LIFE DISTRIBUTION OF EARLY SILICON MESA TRANSISTORS

Fig. 7.2. THE ANALYSIS OF DATA FROM ACCELERATED STRESS TESTS

by

D. S. Peck
Bell Telephone Laboratories
Allentown, Pennsylvania

insepction and test, or by the analysis of field failures. Timely detection and correcting action is crucial in order to minimize the situation of having many devices with generic flaws in the field. The magnitude of the detection problem can be seen by considering the length of time required to obtain 90 per cent confidence of the 0.01 percent per thousand hour failure rate in a sample of a thousand devices. There must be no failures in 2.7 years, there may be one in 4.5 years, and four in 9.3 years. The paucity of allowable failures requires stringent examination of each failed device to determine the failure mode and its potential for screening.

The methodologies of reliability assurance are numerous. No one procedure is ever adequate, owing to the elusive and nearly invisible nature of the problem. Table 7.2 is a listing of several methods that have been used by various programs, often at substantial expense, and with varying degrees of success. Commercial fabrication facilities have been able to adapt some aspects of these methods to enhance reliability while keeping costs low. In some cases it is less expensive overall to use such procedures than not to do so. Certainly, a reliability assurance program can have a cost that is unreasonably high. It has also been shown, however, that programs of moderate cost have produced excellent results. In terms of life cycle costs, where maintenance, logistics, and down time are taken into account, a reliability assurance program of reasonable size can easily be justified. The problem is, how to make it effective.

Success in reliability assurance correlates well with motivation and determination. This follows from the fact that so many potential problems exist, any one of which can result in high failure rates. Every link in the chain must constantly be kept strong. The relationship between airlines and the computer manufacturer can have a favorable impact on the success of the program. A commercial airline application is an environment wherein the computer manufacturer can be involved in the complete life cycle of the equipment. Equipment lifetime warranties are not new to this industry. In some cases, manufacturers guarantee to provide a given number of operational pieces of equipment to an airline for a fixed fee, and handle the repairs and logistics. An extension of this policy into areas of liability, reliability, and cost of ownership would probably be required to obtain the system goal of one failure in 10^{10} hours.

TABLE 7.2

METHODS OF RELIABILITY ASSURANCE

1. AC and DC testing (vendor specs).
2. Acceptable Quality Level (AQL)
   Sampling procedures and tables MIL Std 105D.
3. Use of MIL standard parts.
   Test method for semiconductors MIL Std 750A.
   Test method and procedures for microelectronics
   MIL Std 883.
4. Qualified suppliers list.
5. Qualification testing
   a. Small sample step stressing
   b. Life testing
   c. Screen and burn-in
6. Predicted life from accelerated stress (Bell Laboratories).
7. Lot rejection.
8. Captive line.
9. In-house resident.
10. Fixed processes.

7.5 Recommendation

The failure rate goal of 0.01 per cent per thousand hours for a commercial application can be reached, and has been surpassed in several military and space programs. Nevertheless, the requirement is stringent, and will require out-of-the-ordinary processing. The production cost will be higher than usual, but the increased cost will be counterbalanced by reduced maintenance and down-time costs.

Production startup costs will be high, as solid state component vendors are reluctant to change any of their methods of operation. They do adapt in some cases, and with large production purchases by a customer who understands solid state processing technology and failure modes, they can anticipate returns in the form of increased production yields. The initial costs of special handling, extra documentation, test equipment, burn-in facilities, failure analysis, and engineering evaluation will be considerable. This is one of the areas where component standardization is highly advantageous, as it distributes these costs over many production units.

In the absence of a more specific design, it is not possible to make a detailed recommendation for a reliability assurance program. Some requirements are functions of technology and the environments of manufacturing and purchasing, and must be dealt with at a later time. It is possible, however, to make some specific observations. We have stated before that the desired component failure rate can only be achieved by a dynamic approach in which problems are continuously detected and solved. The reliability assurance program should begin two years before production begins, and should continue for the computer's production life cycle. The following paragraphs roughly outline the elements of such a program.

First, a program of engineering evaluation and qualification testing will be required. Failure modes must be determined, as well as the failure rate for each modes. Environmental tests to trigger each mode must be established, and the level of screenability determined.

From the engineering data generated, screen and burn-in procedures must be designed to monitor and identify component problems. The static and dynamic tests, the cost trade-off between component screen and burn-in vs. module burn-in, and component procurement specifications can then be generated.

Module burn-in for the fault-tolerant computer can be readily performed because of the many fault detection techniques built into the system. The burn-in rack would simply be a large computer motherboard. The biggest cost of module burn-in would be in the cost of the modules themselves. Early failures not picked up at component screen and burn-in could be detected at module burn-in. The degree to which one depends on an extensive burn-in depends on the failure rate of the procured components and the cost trade off.

Failure analysis on field failures and module burn-in failures are required. After the production process is established, this data can be used for corrective action by the vendor and an indication of failure rate trends. This is a long time-constant detection method, but it is capable of indicating minute changes in failure rate. A type of Qualified Suppliers List is required. This, coupled with the component specification requirement, will help to generate the procurement plan. Data from Failure Analysis will update the Suppliers List.

Lot jeopardy or lot rejection, such as employed on the Apollo program, is not required. Some type of in-line quality detection method is necessary, however, to prevent many modules from being built with a bad lot of parts. Screen and burn-in plus module burn-in can detect these problems, but detection at the module burn-in level is costly because of rework or scrapping of modules.

CHAPTER 8

PACKAGING CONSIDERATIONS

This chapter covers the packaging of the fault-tolerant computer from a general packaging philosophy through to brassboard and production packaging schemes. The section on packaging philosophy promotes the idea of physical integration, where an attempt is made to make packaging elements serve 2 or 3 of the basic packaging functions, interconnection, thermal control, and physical integrity. This is followed by a statement of the fault-tolerant computer packaging problem in the form of the computer architecture to be packaged and the operational and environmental constraints which must be considered. A general discussion of some packaging technologies is then followed by specific concepts for the brassboard and production computer packaging.

The brassboard packaging is done using mature technologies. In fact, there is no reason the ultimate reliability goal for the computer cannot be achieved using the design proposed for the brassboard. The production model packaging scheme proposed for a build in the 1980-85 time frame could easily be built now, but with some lack of confidence in reliability. By 1980-85 the technologies used in this design will have achieved a maturity which will improve our confidence in their reliability. Also, we might expect by then a small saving in cost, as well as the large saving in volume and weight this design will provide.

8.1 Packaging Philosophy

The term packaging can be applied to a wide range of activities. It can be used for putting resistors in a carton for shipping, for designing a TO-5 can and putting a transistor chip in it for physical protection, or for the physical design and assembly of a complete electronic system.

In order to focus on the emphasis required for the physical design of systems incorporating components such as LSI and hybrid circuits the term "physical integration" will be used. The term alludes not only to integration in a material sense, but also to integration based on

physical laws applied to the system as a whole. For example, the designer of a system connector must consider not only such things as connector size, resistance, and reliability, but also whether it will contribute to system heat transfer and structural integrity.

The human nervous system is a good example of a physically well integrated system. Of course economics and repairability place limits on building such a system even if we had the technology. Physical integration will require that system designers apply their knowledge of logic, device physics, materials, heat transfer, high-speed data transmission, electromagnetic interference, reliability, repairability, and testing concepts. All these factors must be considered and compromises must be made to achieve future system goals.

The complexity of these tradeoffs is more evident when we realize that a study of heat transfer alone must consider thermal properties of materials, thermal interfaces at mechanical and electrical joints, the distribution of thermal sources and sinks, device dissipation, device temperature rise versus reliability, and cooling methods available.

We may gain some insight into what can be done to physically integrate a system by examining LSI itself to see what makes it so attractive. The immediate and somewhat superficial answer is because many packages are replaced by one package, many interconnections are eliminated, and many of those connections that remain are on the chip itself, which reduces cost and increases reliability. Perhaps more basic to the success of LSI, and indeed the reason for its name, is the fact that all parts of the LSI chip serve multiple basic functions. The components and interconnections in a LSI structure compose only a small part of the volume of a chip. However, the remaining part of the chip contributes to its overall strength and thermal dissipation, the prime functions of the larger part of the otherwise unused silicon. We might conclude that the design of parts of a system to serve two or more basic functions is a desirable guide in working toward a "physically integrated" system.

Physical design, in the most basic terms, consists of designing for the electrical interconnection of components, extraction of heat, and mechanical integrity in a specified environment. One must bear in mind at all times the tradeoffs between these areas while evolving a design. System logic designers must consult with the people responsible for the physical design in the conceptual phase to achieve the goals of physical integration. A good system design can only be achieved if

good compromises are made in the areas of system electrical design, and system physical design.

Interconnections are best when eliminated. If this is not possible, permanent interconnections are preferred to demountable ones. Unfortunately, the need for repair, modification, economy, and electrical test preclude this ideal situation. As a compromise situation we must resort to partitioning a piece of equipment or system into manageable portions which may be interconnected by demountable connections. Good partitioning requires the cooperation of the electrical, LSI, and physical designer. For example, increasing the number of gates in a LSI chip may reduce the lead/gate ratio. However, the number of lead connections which can be put on a chip is a function of area. Of course there are, and always will be, technology limits on how big a chip can be made and packaged. These variables leave considerable room for tradeoffs. As another example of an area for designer compromise, it might be desirable to use serial instead of parallel transmission to reduce interconnections. Such a change may or may not be compatible with system speed requirements.

When analyzing a system from a physical integration point of view it is helpful to consider the levels of interconnection in the system and the method of connection. A complex system may have 4 (or possibly more) levels as listed below:

Level 1 - can be represented by interconnections in a hybrid circuit. Interconnections are usually non-repairable. A failure in a component would require replacement of this level as the unit is not repairable.

Level 2 - can be represented by a multilayer wiring board, or cordwood welded matrix interconnections. Connections are semipermanent. Repairs can be made at the fabrication facility.

Level 3 - can be represented by pin connectors with a wire-wrap field interconnection. Connectors are demountable and therefore a replacement module can easily be installed.

Level 4 - can be represented by a cable and connectors interconnecting subsystems. These connections are demountable and therefore subsystems can be replaced easily.

These defined levels of interconnection aid in dividing the physical integration problem into portions of manageable size at the hazard of neglecting the effect that a decision made in one area will

have in the other areas. Keeping this hazard in mind, the physical designer must consider for each level the items listed below:

1. Heat transfer within each level and to the next level.
2. Type of connection or connector.
3. Type of interwiring.
4. Partitioning for a minimum number of connections, ease of testing, and ease and cost of repair.
5. Ability to stand environmental stresses.
6. Reliability.
7. Volume.
8. Weight.
9. Cost.

The environmental conditions under which a system may be required to operate vary over a wide range. Ground-based computers may be in environmentally controlled rooms isolated from all but the mildest vibrations. Spaceborne computers may be required to operate reliably under severe shock, vibration, and temperature excursion conditions in a vacuum as well as under atmospheric conditions. Care is required in the selection of materials for matched thermal expansion and, if this is not possible, methods of stress relief must be built in. Substantial amounts of material may have to be added for structural rigidity. Everything must be held firmly in place with screws, cement, potting, etc. A physical designer should repeatedly ask himself whether the material added for mechanical integrity can also be made to act effectively to help solve the heat transfer problem and/or double as part of a connector structure.

The heat packing densities possible with LSI imply a significant increase in heat dissipation per unit volume despite the fact that the power per function is reduced. The cooling technique for a particular system must take into account coolants already available. For example, cooling air might easily be bled from a jet-engine compressor. One should design for a minimum weight of the electronics and cooling system taken as a whole, particularly for spaceborne and airborne systems. The ability of the cooling system to remove large amounts of heat from small volumes is important to prevent hot spots. For high power density equipment, cooling by liquid flow or cooling by boiling might be considered. Thermoelectric cooling and heat pipes should be

considered for hot-spot control. It is of course advantageous if material added for thermal control doubles either as a structural member or as part of a connector.

8.1.1 Assumed Computer Architecture for Packaging

The object of this section is to set forth in condensed form the architectural features which are of importance in developing a packaging concept for the computer.

We have assumed that 28 VDC power is available from 5 separate sources. For example for an L1011 these could be the alternators on 3 engines, the auxiliary power unit, and the battery bank. Because the computer may never be without power during flight, it may be necessary to add a dedicated battery system to be used during short time power outages. This is already being done for navigation systems such as the carousel. The need for such a dedicated battery can only be assessed for a particular application and we have assumed for simplicity that it is not needed. The separate power sources are brought to each module where the necessary isolation, conditioning, and overvoltage protection is provided. The total power budget of 28 VDC is 1400 watts. This conservative number reflects the possibility that LSI utilization may be limited by considerations of producibility and testability.

The computer is made up of processor modules, memory modules, clock and I/O access modules, and a suitable bus structure to interconnect the modules. Table 8.1 lists the modules and buses required. More detail about the number of bus conductors is given in Table 8.2. Here we have assumed that each connector will provide a pair of pins to make contact to each bus. Figures 2.2. and 2.3 in Chapter 2 schematically show how the various modules and buses are connected in the computer. The bus guardian units and isolation gates,depicted separately, will actually be in the modules they are associated with.

The architecture implies that one failure must not be permitted to cause another failure. For example a short circuit in one module must not spatter metal on the active section of another module. In addition the system cannot tolerate a serious rash of generic failures in either the electronic components or packaging components. Generic failures are caused by a built in wearout mechanism in a given part type which causes all these parts to tend to fail after a given amount of usage rather than randomly.

TABLE 8.1

BASELINE COMPUTER ARCHITECTURE

MODULES

20 Processors
20 Memories
6 Clock & I/O Access

Each module contains 2 bus guardian units, bus isolation gates, and its own power supply.

MODULE INTERWIRING

5 I/O Buses
5 Memory Buses
5 Power Buses
6 Clock Buses
6 I/O Net Buses

TABLE 8.2

LIST OF MOTHERBOARD WIRING

CLASSIFICATION	NO. OF CONDUCTORS	NO OF PIN PAIRS
POWER (28 VDC)	10	10
CLOCK	12 + GND PLANE	13
I/O BUS	15 + GND PLANE	16
MEMORY BUS	15 + GND PLANE	16
I/O NET	12 PAIRS	24
TOTAL		79

8.1.2 Environmental Constraints

The most serious environmental constraint is thermal. ARINC Report Number 414 suggests a storage range of -65°C to +71°C and a normal operational range of -40°C to +55°C. This is quite a large temperature range for high reliability electronics.

We have assumed that cooling will be done through the medium of ambient air and that the air contains too many contaminants to be passed directly over the electronics. The availability of a vacuum plenum for pulling cabin air across the heat exchange surfaces in the equipment will not preclude the necessity for fans in the equipment in the event the vacuum system fails. The computer will require considerably more than the recommended 6.8 - 13.6 Kgms of cooling air per hour per 100 watts dissipated. The 26°C air temperature rise with a 13.6 Kgms./hour flow is much too high.

Atmospheric pressure ranges from 305 meters below sea level to 13700 meters above sea level must be withstood by the computer. It must also withstand a decompression from 13,700 to 2,130 meters.

The computer must withstand vibration, landing, shocks, and handling shocks. Adequate shielding to protect against EMI, and in particular EMI generated by lightning strikes, must be provided. Much of the detailed information on environmental constraints can be found in ARINC Report No. 414.

8.1.3 Operational Constraints

We have assumed that ATR packaging, although desirable, will only be used if it can be shown that such packaging does not measurably influence the computer's reliability. It is particularly important that the fault-tolerant computer reliability not be compromised by efforts to reduce or by efforts to design it into an unfavorable form factor. This concept is not new in aircraft practice, as we see from ARINC 414 where it says, "The airlines place great emphasis upon reliability and maintainability with only secondary interest in size and weight." A severe failure budget, such as 1 failure in 10^{10} operational hours for the computer, allows little room for compromise.

Maintainance should normally be done on the regular maintainance cycle. Since the computer can pinpoint its own failures to the level of a processor memory, clock, I/O access unit, or bus, it seems reasonable that the modularity of the computer conforms to these functional blocks as much as is practical. Maintainance would then consist of

pulling the failed module and replacing it. The length of operation between scheduled maintenance stops is one of the determinants in fixing the number of spare modules which must be built into the computer.

8.1.4 Aircraft Specifications

Various ARINC specifications and reports were drawn upon in studying the airborne fault-tolerant computer packaging problem. It is anticipated that the guidance of these and similar documents would be followed except where a well defined advantage will result from taking another direction. The documents which significantly contributed to this report include:

"Air Transport Equipment Cases and Racking", ARINC SPEC 404.

"Guidance for Designers of Airborne Electronic Equipment," ARINC Report 403.

"General Guidance for Equipment and Installation Designers," ARINC Report 414.

"Guidance for Designers of Aircraft Electronics Installations," ARINC Report 306.

"Guidance for Aircraft Electrical Power Utilization and Transient Protection," ARINC Report 413.

8.1.5 Packaging Technologies

It is not possible in a few pages to even list, much less fully discuss various packaging technologies which may be applicable to fault-tolerant computer packaging. However, it is felt that a limited discussion which covers technologies we have selected for use in both the brassboard and production designs as well as some we considered and rejected, will give some insight into the selection process.

Starting at the lowest level of interconnection in the system it is logical to first consider component packages. Integrated circuits are generally available in two package styles, flat packs and dual-in-line (DIP) packages. Flat packs are substantially smaller than DIPS and more easily removed for repairs. However, they are more expensive, less available, and not adaptable to wave soldering. Both package types can be of either hermetic or non-hermetic construction. Although recent tests have demonstrated the high reliability of some non-hermetic packages when used with the right semiconductor

metalization and passivation systems, the hermetic package will still offer a reliability edge for many more years. Hybrid circuits can also be made in DIP or flatpack form. Because one hybrid circuit contains a considerable amount of electronics a high number of I/O leads are necessary. The flatpack is more often used because of closer lead spacing (1.27 vs 2.54 mm) and the greater ease of putting leads on 4 sides of a flatpack. Many passive components, lower power resistors, small capacitors, and small inductors, can also be put into the DIP and flatpack package styles. For the brassboard computer the DIP package has been selected. For the production model which would be built in the 1980-85 time frame we believe hybrid circuit technology can be gainfully used. The flatpack will be used for packaging these hybrids.

Hybrid circuit technology provides a means of interconnecting many components in a small space and then protecting them in a common package. Many advantages can accrue. There are fewer connections in the system, and the system is smaller. Smaller size implies lower weight, higher speed, and lower power. The materials and processes used in hybrids are far from standardized today. Of particular importance are differences in semiconductor bonding. Connections can be made using gold or aluminum wires and thermocompression or ultrasonic bonding. IBM has standardized on a flip chip solder collapse bonding technique where the chips are specially made with solder bumps at the bonding pads and the chips are protected by a glass coating. Bell Telephone uses a gold beam leaded semiconductor chip which is protected with silicon nitride. The chip is bonded by a thermocompression process. Neither the IBM or BTL processes have been widely used outside these companies and procuring a line of chips which have been reliably processed in either format is not presently possible. Many other semiconductor attachment processes have been tried over the past fifteen years and all but one have remained in relative obscurity. That one is the so called film bonding or tape automated bonding. In this process the chip is prepared with bumps on the pad areas. A reel of plastic film with sprocket holes in it carries sets of copper fingers contilevered over cavities in the film. A set of fingers is gang bonded to the bumped chip. The next step is to cut the chip with fingers free from the film and gang bond it into a package or hybrid circuit. The keys to the success of the process are the ease of handling and operating on chips in the reel format and the gang bonding. This process is being used by several vendors for DIP packaging and is beginning to find its way into hybrid circuit manufacture.

The base of the hybrid circuit consists of a substrate which supports a wiring pattern. To this wiring pattern active and passive components can be individually attached. Also it is possible to deposit, via batch processors, resistor, capacitor, and inductor elements. The processes for depositing materials, pattern formation, and component adjust are legion. A few of the most important aspects are touched on here.

Wiring patterns can be made in multilayer format by two primary methods. One starts with a tape on the order of .25mm thick made from ground ceramic and plastic. Holes are punched in the tape at appropriate places and are filled with a metal powder in an organic binder. A pattern is also silk screened on the tape using the same metal - organic mixtures. These tapes are then accurately stacked, pressed together and then fired until the ceramic and metal sinters into a solid block. The result is a multilayer wiring structure in a block of ceramic. A second method starts with a fixed ceramic substrate. Alternate layers of metal wiring patterns and dielectric are screened and fixed to again form a multilayer wiring structure.

Resistors can be made by silk screening special glass and metal, or glass and semiconductor formulations at appropriate gaps in the wiring pattern and firing. Capacitors and inductors can be made using the fired conductor material in conjunction with screenable and fireable high dielectric constant and high magnetic permeability materials respectively. Silk screening is a thick film process.

Thin film processes such as vacuum evaporation and sputtering can be used to deposit conductor, resistor, and insulator materials on ceramic substrates. Photolithographic processes can be used to form these materials into patterns to make circuits.

The addition of semiconductor components and discrete passive components to structures made in the aforementioned ways completes the hybrid circuit.

Any product which is made in so many different ways must have some decisively good features. Our problem is to select materials and processes which best lend themselves to control, and to find vendors who are capable and willing to exert the degree of control necessary for the fault-tolerant computer application.

Individual components and hybrid and LSI circuits must be interconnected in the system to form modules. Multilayer ceramic wiring boards made as previously outlined can fulfill this function if the board area required is not too large, say under 160 cm^2. Printed circuit boards made using epoxy glass sheets with copper laminated to one or both sides offer an excellent alternative for larger area boards. According to one recent study these boards become more economical on an area basis up to the 1300 to 2000 cm^2 range. Since most circuits cannot be laid out without extensive use of crossovers a 2 sided board becomes necessary when interconnecting many-leaded components. The most reliable method of electrically connecting the wiring paths on one side to the paths on the other is through the use of plated through holes. Additional wiring layers may be required in the board for 2 primary reasons. It may be necessary to provide isolation or controlled impedance by the use of ground planes. Also the required board area can be reduced by about a factor of 2 in going from a 2-layer board to a 6-layer board.

Interconnections at the module level and mother board level might be provided by two other technologies. Wire wrap, where a solid wire is tightly wrapped around a square post to form a gas tight point has been shown to be extremely reliable in the telephone system. Changes can easily be incorporated in wire-wrap interconnections. Perhaps the single largest drawback is the volume required. Stitch welded wire is less space-consuming than wire wrap, but is also more difficult to change and does not have the long history of reliability that wire wrap has. Connections are made by pressing a teflon coated nickel wire against a stainless steel pin with enough force to break through the teflon. The welder is then fired to make a weld joint.

When selecting module connectors and computer I/O connectors, a myriad of choices is available. There are tradeoffs to be made in the amount and type of plating or inlay to be used. Wear must be factored in. Wear increases with mating force, but adequate mating forces are required for reliable connections. Zero insertion force connectors are connectors which are mated without having the contacts touch. One of the sets of contacts is moved, using a mechanical actuator build into the connector, to touch the other set of contacts. This type of connector offers high contact forces with a minimum of "wiping" action and therefore a minimum of wear, however the need for their use in this application is not anticipated. Perhaps the most important point is

that the connectors must be protected from contamination and mechanical damage when unmated.

Control of materials and processes used in the fabrication of packaging components and in the assembly of the finished equipment requires a substantial effort. Processes must be mature and well documented. For high reliability packaging nothing must be left to chance, there must be no ambiguities in specifications, and as much operator judgement as possible must be removed from the process. There must be constant vigilance that specifications are followed, and the end products must be inspected and tested to insure that the product continues to meet specified standards.

8.2 Packaging the Fault-Tolerant Computer

The two overriding concerns for designing any piece of equipment are: one, will it do the job it is supposed to, and two, can it be done in an economical manner. For the fault-tolerant computer we must first concern ourselves with meeting the reliability goal. Concerns with size, weight, and the economics of a production model will be brought to the foreground upon successful demonstration of the brassboard.

The following sections outline packaging concepts for both a brassboard computer to be flown about 1980 and a production computer for 1985 flight. The brassboard packaging makes use of well-known technologies with a proven capability in high reliability systems. With proper production controls we believe this design could be built in an economical manner to achieve the 10^{10} hour reliability goal. The brassboard is large and heavy (about 227 Kgms.). Although the technology presently exists to reduce the size and weight by a factor of 10, we believe it is not now mature enough to be used for a 1980 flight. We are however, predicting that this size and weight saving can reliably be made in the production model to follow. It should also be noted that a considerable saving in weight is made if the brassboard computer uses substantially fewer than this baseline number of modules.

We have assumed that the computer will make use of the air-conditioned equipment-bay air for cooling. We have also assumed that should the bay become extremely hot or cold during ground parking the computer will only be turned on after the bay is conditioned to a reasonable temperature range, for example 0 to 50°C.

The demonstration of potential reliability in the brassboard model is the important goal at this time. Technology cost tradeoffs are only significant if done a short time before the production design and based on costs and technologies as they exist then. Cost analysis must also consider such things as repair philosophy, operational savings resulting from weight reduction, and the very difficult question of cost/reliability tradeoffs.

8.2.1 Brassboard Packaging Design

Ambient avionics compartment air is to be used for cooling the computer. Since wide temperature extremes can be encountered when a plane is parked unpowered in Arabia (+71°C) or Alaska (-65°C) we have assumed that the avionics compartment will be conditioned in the range of 0 to 50°C before the computer is turned on.

The power dissipation of the brassboard multiprocessor is estimated at 1400 watts. If we assume the air temperature rise of the cooling air is to be limited to 5.6°C then 12.6 m^3/mm of air must be supplied at one atmosphere pressure. The fans used can be designed to increase their speed at cabin pressures below one atmosphere so the weight of air delivered per unit time remains about constant. Of course this air must also cool the fan. This is significant as about 35.3 watts/m^3/mm required which amounts to 445 watts additional dissipation. It should be pointed out that ARINC prefers that forced air not be used. In our opinion, however, the advantages out weight the disadvantages.

Cooling would be most efficient if the air were drawn directly over the component to be cooled. Such cooling does cause a buildup of dust on the components with time, even when the air is filtered. There is also the remote possibility that water or other fluid could come in contact with the electronics if the cooling system were designed this way. The next option is to case each module and conduct heat from the components to the case where it is subsequently removed by convection. Although this approach does add the conduction ΔT to the component temperature, it is felt to be the better choice for this design. The cases will not be hermetic. They will be rugged enough to withstand decompression should it occur. Of course pains will be taken to reduce the conduction ΔT by providing efficient thermal paths from the components to the case.

With a 6.35 mm spacing between modules, the convection coefficient for laminar flow will be about 19.4 watt/cm^2. With about 13 cm^2 of convection area (with no attempt to increase the area with fins or corrugations) the temperature difference between the convective surface and the air is 5.6°C. If fan power is included, the worst case ΔT between module wall temperature and inlet temperature will be 12.8°C. For a 24 pin DIP package the thermal resistance between the package and convective surface will be approximately 16.7°C/watt using the construction set forth in the Module Design section. Therefore 1/2 watt 24 lead packages will operate approximately 22.2°C above intake cooling air temperature.

We have assumed that a bank of 10 fans, each with 10 cfm capacity, will be arranged to feed a single plenum and filter arrangement for the computer. The fans will be fitted with flapper valves to prevent air from blowing back through the fan when it is off. The fans will be arranged in groups of two. Each group will be fed by 5 diode isolated power lines and one fan will run while the second fan is held as backup in case the first fails. The computer will control the switching of fans. With the ambient temperature around 21.1°C the computer should operate without excessive overstress on as few as two fans. Expected fan life can range from 2000 hours to 25,000 hours depending on fan quality. Of course the more reliable models would be selected for this application.

It is recommended that a temperature monitor be placed in each module. This monitor can warn of a single module thermal problem as well as signalling a problem caused by fan failures, dirty filter, high avionics compartment temperature, or a combination of these conditions.

The concept of making each processor, memory, and clock a single module arises from the fact that these are the smallest entities to which the system can isolate faults. It therefore seeems desirable to correct any detected failure by replacing, in a single package, the electronics containing that failure. It is also desirable to have the module protected physically from damage inside and outside the multiprocessor assembly. For example, a failure on one module, such as a short circuit, must not spatter conductive material on the active section of another module, as propagated failures must be rigidly guarded against in this architecture. When the module is removed from the multiprocessor assembly, it is important that the electronics be

protected against handling damage and contamination.

The added material to protect the module does raise some design problems. The primary problem is designing for a reasonable thermal drop between the components and the outside convective surface. This would tend to push the design to a thin planar module to reduce the length of the conductive paths. Other considerations push in this direction also. A different form factor resulting in a thicker module would dictate the interconnection of two or more planar structures inside the module. Such an assembly would likely prove more difficult to repair than a single planar structure, and also would be less reliable. An external consideration that makes the thin module preferable is the interconnecting bus design for the modules. The easiest bus design is simply a collection of straight bus bars. Thick modules would necessarily make the bus longer, so long in fact that the bus would have to be bent to make the multiprocessor in some reasonable form factor.

The planar structure suggests a printed circuit card as the means of interconnection. The dual-in-line (DIP) hermetic package has been selected as the preferred logic package. It would be desirable, where possible, to use this package for passive components and for power supply packaging. For purposes of this study we have estimated that each processor and each memory with their power supplies, bus guardian units, and bus isolation gates will require the equivalent of 180 16-lead dual-in-line packages. In order to interconnect these in a reasonable area the printed circuit card must be of a multilayer construction. We are allowing 6.45 cm^2 of card area per dual-in-line package. This area includes the overhead for fastening the card in the module and for the I/O connector attachment.

Ease of module repair precludes adhesively attaching the multilayer board to one side of the module structure for good heat transfer. We would like a system which would allow access to both sides of the board. If a dual-in-line package is simply soldered into a board, the only conductive thermal contact is through the DIP kovar leads. Radiation and convection do not provide low resistance thermal paths from the package, nor do the kovar leads provide a particularly good heat flow path. To achieve a better thermal path, "2 oz" copper cladding will be used on both sides of the board. The DIP packages will be adhesively attached to the top layer of copper using double sticky tape. Heat will be removed from the board through standoffs coming from both

external faces of the module on a 50.8 mm matrix. The screws through these standoffs will serve triple duty as primary module fasteners, heat transfer paths, and structural members necessary to withstand sudden pressure changes. This thermal scheme can keep the temperature difference between a 16-lead DIP and the modular convective surface below 33.3°C/watt. If necessary, this ΔT can be further lowered by increasing the PC board copper cladding thickness, using a more highly conductive but less repairable DIP adhesive, and by increasing the area and/or number of standoffs.

The multilayer printed circuit card will consist of 4 internal wiring layers plus one ground plane and one power plane. It is not anticipated that additional ground planes will be required between wiring layers.

The connector will consist of 2 rows of 80 pins on 3.96 mm centers. The pins will be used in pairs to provide doubly redundant connections to the various buses. A skirt will be provided completely around the protruding connector pins to provide protection when the module is not plugged in. This skirt is pressed against a gasket material when the module is installed to provide a seal between the exposed connector pins and the convective cooling air. Two guide pins will be provided to guarantee proper alignment during connector mating and that the module is not installed backwards.

Two levers can be provided on the outboard end of the module. These are used to provide the force for both engaging and disengaging the module connector.

Two guide bars are provided on the module which are used to engage guide slots which align the module when it is inserted and withdrawn from the multiprocessor assembly. An external view and cross sectional view are shown in Figures 8.1 and 8.2.

The multiprocessor case must perform several functions, the most important of which are listed below.

1. It must provide for the physical protection and support of the modules placed in it - 20 processors, 20 memories, and 6 modules containing clocks and I/O access electronics.
2. It must provide for the electrical interconnections of the modules to each other and to the external world through module connectors, I/O connectors, and power connectors.

STANDARD MODULE
356x381x25.4mm

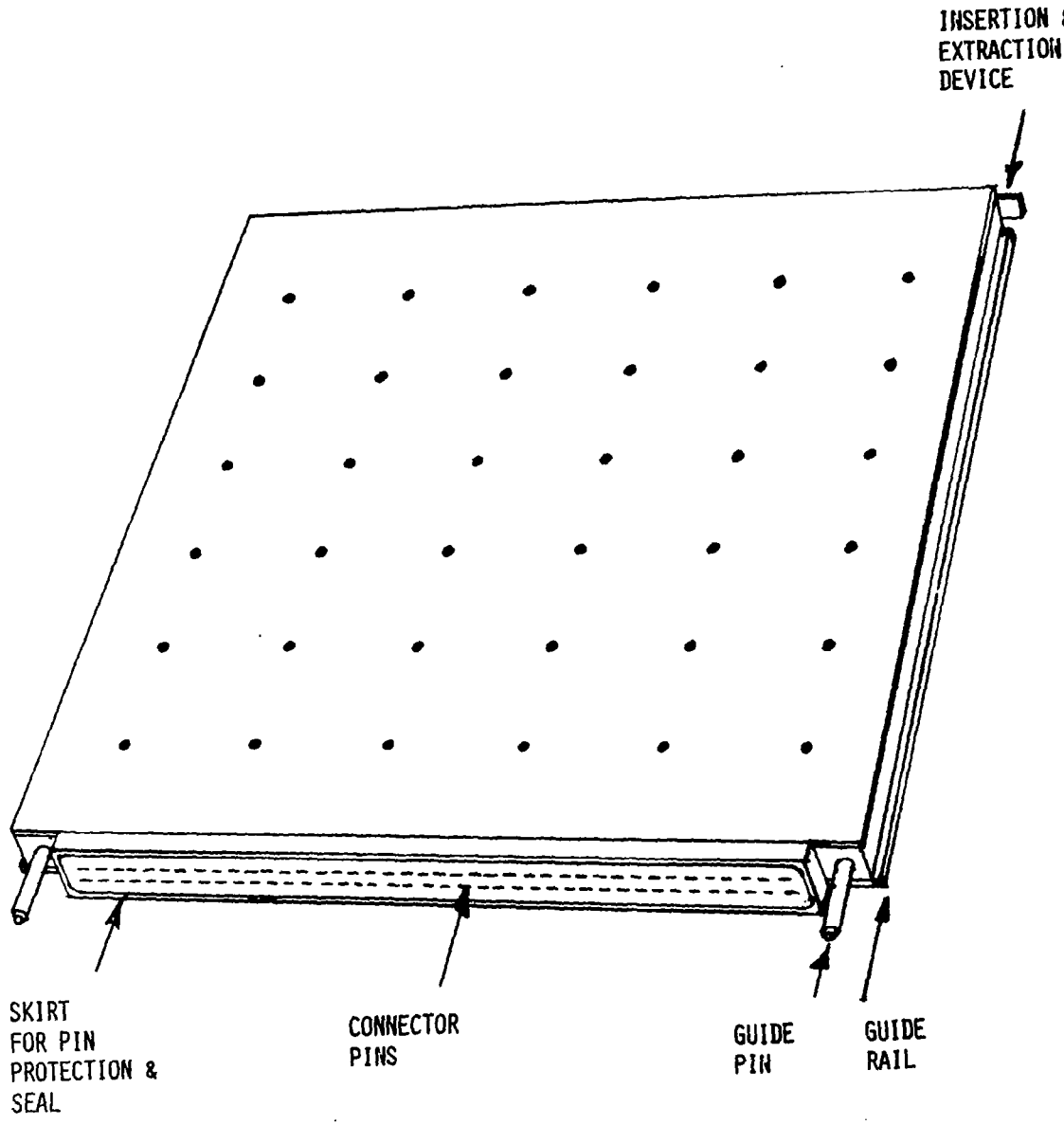
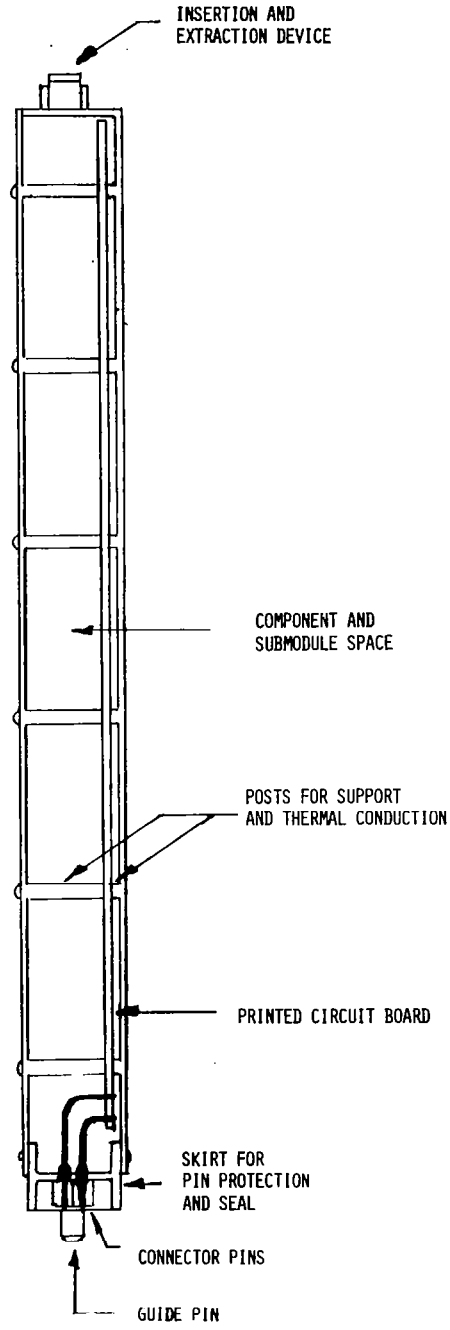


Fig. 8.1.

FIGURE 8.2
BRASSBOARD STANDARD MODULE
356x381x25.4mm



3. It must serve to direct the cooling air for proper cooling.

4. It must assist in providing EMI protection. (Part of this burden falls on the module cases.)

Figure 8.3 is a schematic of a case which provides these functions. The case is 432x457x1245 mm overall outside dimensions. The module slides are placed on 25.4mm centers, which allows 6.35mm between adjacent modules for cooling-air flow.

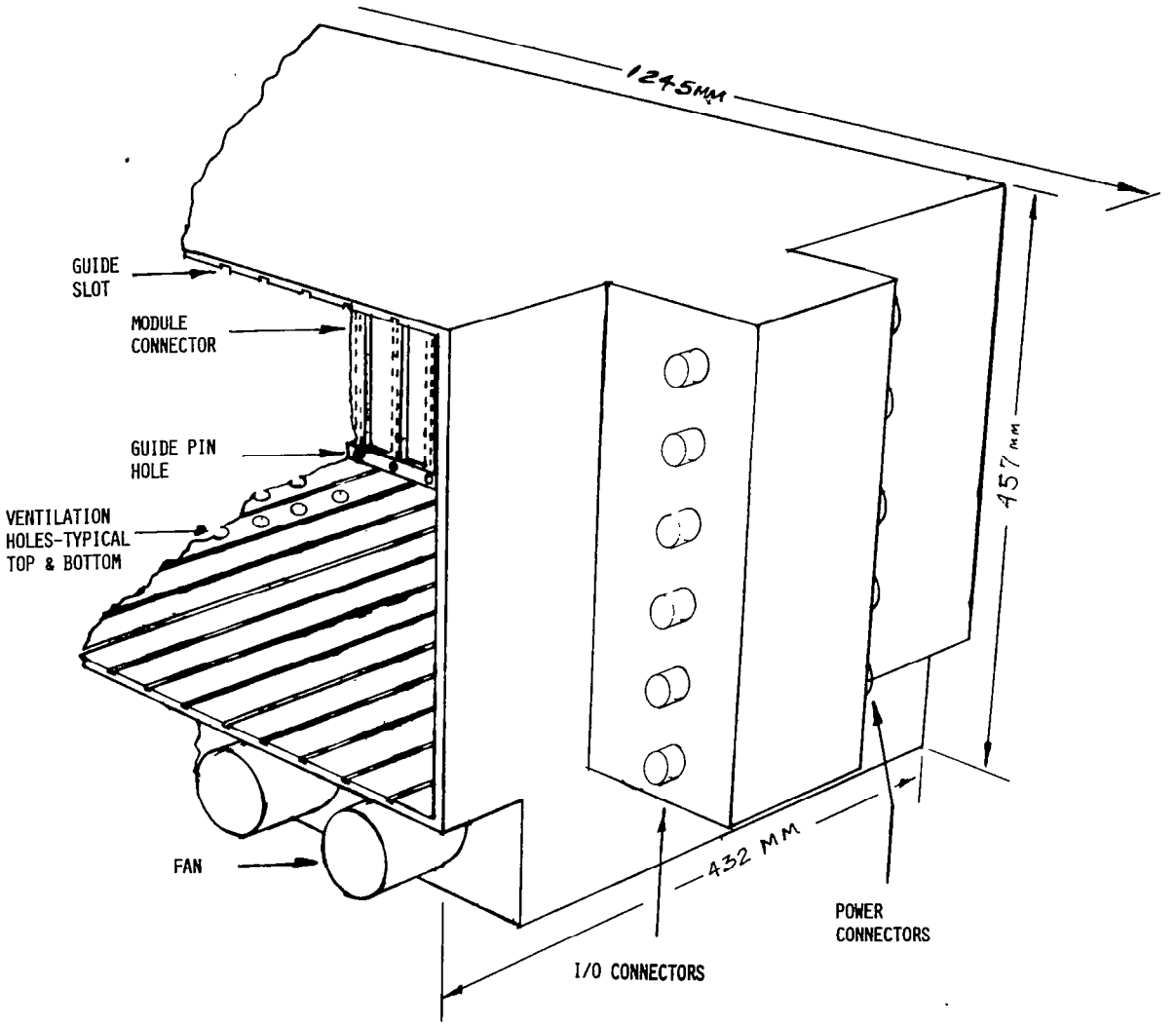
A common plenum, with the 10 fans attached and containing an air filter, communicates with the module section through a series of holes which line up with the spaces between the modules. The air exits through similar holes in the top of the box, which can be protected against falling objects by a baffle plate. If cooling air is available in the avionics equipment racks this design can easily be adapted to the use of this air. The fans can be retained as a backup or removed based on the reliability of the plane's forced air system. Not shown in the figure is a hinged and latching front plate which closes the box.

The modules are guided into the case by module slides, and the final connector mating is guided by two pins on the module connector which fit into two corresponding holes at the ends of the box connector. There are two levers permanently attached to the back of the module which engage slots in the box to provide the force for engaging and disengaging the connector. These levers also serve to lock the modules in place with the box cover providing a backup for this function.

Connections to the outside world are provided through five power connectors and six I/O connectors. These connectors are of the circular, bayonet locking type. The motherboard connector contacts are of the fork type on 3.96 mm centers and are installed with their individual insulators in an accurately drilled matrix of holes in an aluminum plate.

Table 8.2 shows the various signal and power wiring which must be carried by the computer motherboard. The motherboard wiring consists of all straight buses. The power buses must be capable of carrying 50 amps at the nominal 28 VDC input voltage. The internal supplies must be able to operate with a considerable variation in input voltage so that voltage drop along the line is not as important as I^2R losses. We have assumed a copper bus bar of a cross section 2.54x2.54 mm

FIGURE 8.3
BRASSBOARD FAULT-TOLERANT COMPUTER CASE



which will have a resistance of under 5 milliohms for a 1.22 mm This would produce less than 0.02 volts drop at the end of the bus and an I^2R dissipation on the order of 5 watts in the bus. The ground return buses can either be of the same type or can be made common. The bus bars will be laminated to a .79 mm thick piece of G10 epoxy glass for structural integrity and ease of handling and will be separated from the bused signal paths, which will be a second board.

A sample section of the board used for signal paths is shown in Figure 8.4. The I/O net wires are run as signal pairs on opposite sides of the board while the other signal lines are run over a ground plane buried in the center of the board. Wires from the I/O and power connectors are joined to the buses by soldering them to extra through holes made in the buses at one end.

8.2.2 Production Packaging Design

We project that hybrid technology will have achieved adequate maturity to be seriously considered for the production fault-tolerant computer design for 1985 production. Several benefits which will accrue from the use of hybrid technology when compared to the use of multilayer printed circuit boards and discrete components are listed below:

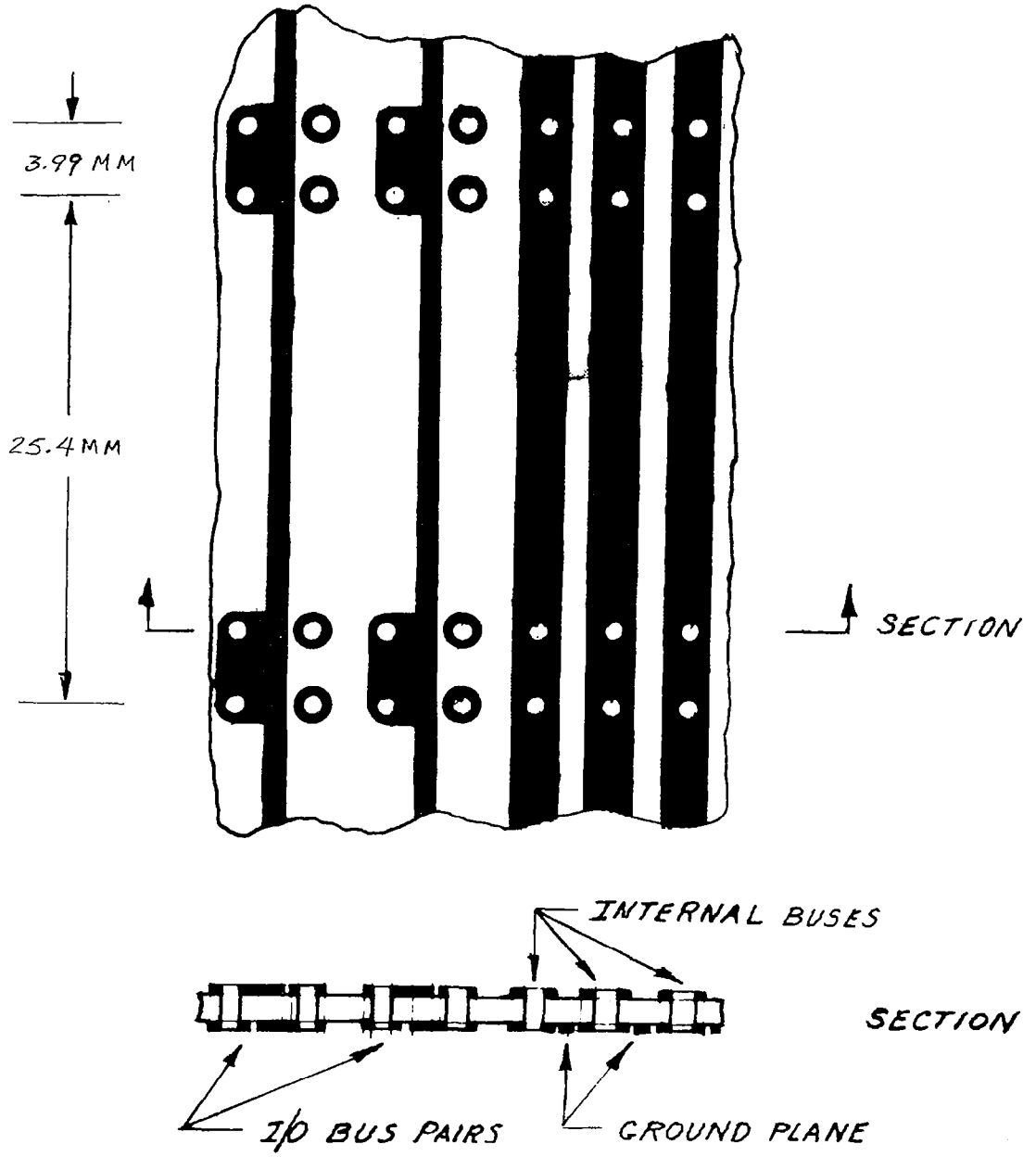
1. A substantial size and weight saving.
2. A small increase in computer speed and a small decrease in power required - assuming the use of the same semiconductor chips.
3. A small saving in cost.
4. The stocking of a small number of hybrid circuit types for repair rather than a large number of discrete component types.
5. Advances in technology can be used to build hybrids which can be used as one for one replacements for old hybrids when repairing the computer.

We further project that semiconductor advances will provide much higher levels of chip integration and a total power requirement of 350 watts vs. 1400 watts for the brassboard.

The overall design concepts used for the brassboard will be maintained in the production design. That is, the modularity, bus

FIGURE 8.4

MOTHERBOARD SIGNAL BUS STRUCTURE



structure, and cooling will be similar. The hybrid circuits will allow the active area of the computer module to be reduced from 381x356mm to 38.1x101.6mm The overall module thickness will be 12.7mm and will include a built in heat exchanger to take care of the increased power density. The reduced size of the module will require that the connector pin spacing be reduced from 3.99-2.54mm so that the required number of contacts can be fitted in the available length. A computer module made using hybrid technology is shown in Figure 8.5. The interconnections between the 4 to 8 hybrids in the module can be made using either a multilayer epoxy glass board or a multilayer ceramic board. The ceramic board can offer a better heat transfer path to the heat exchanger, although it is more difficult to mount than an epoxy glass board because it is more easily cracked by thermally induced stress caused by the difference in expansion coefficient between the board and heat exchanger. This problem can be designed around by making the heat exchanger in such a way that it can provide stress relief even though it is firmly soldered or cemented to the ceramic board.

The production computer case will be made in much the same configuration as the brassboard. The electronics section will be 152.5x165x635mm without the fans, plenum, I/O connectors, and power connectors. When these items are added to the box the overall dimensions are 229x229x660mm compared to the brassboard overall envelope of 432x457x1245mm Note that no space is required between modules, except clearance, because cooling is done by directing air through the module heat exchanger rather than between modules as done on the brassboard. See Figure 8.6.

It is also possible to package the computer in a one and one-half ATR box measuring 496mm long by 390mm wide by 19.35mm high as shown in Figure 8.7. There are some drawbacks to this package which must be considered. Probably most important is the more complex bus structure dictated by the requirement that the modules be placed in 2 rows because of the box length limitation. Also, normal racking would require that the entire computer case be unplugged each time a module is removed.

FIGURE 8.5

PRODUCTION MODULE

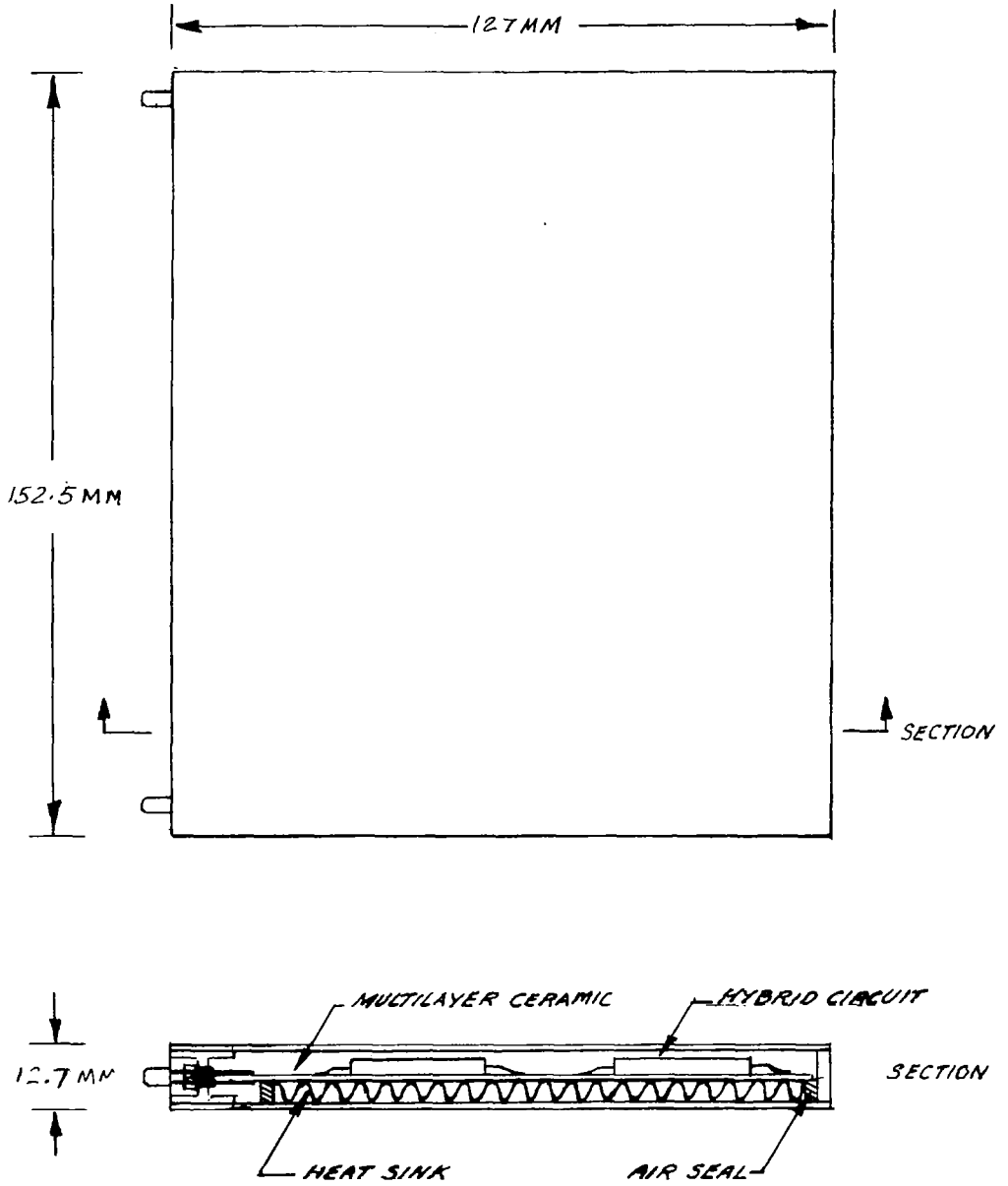


FIGURE 8.6

SCHEMATIC SECTION OF PRODUCTION CASE

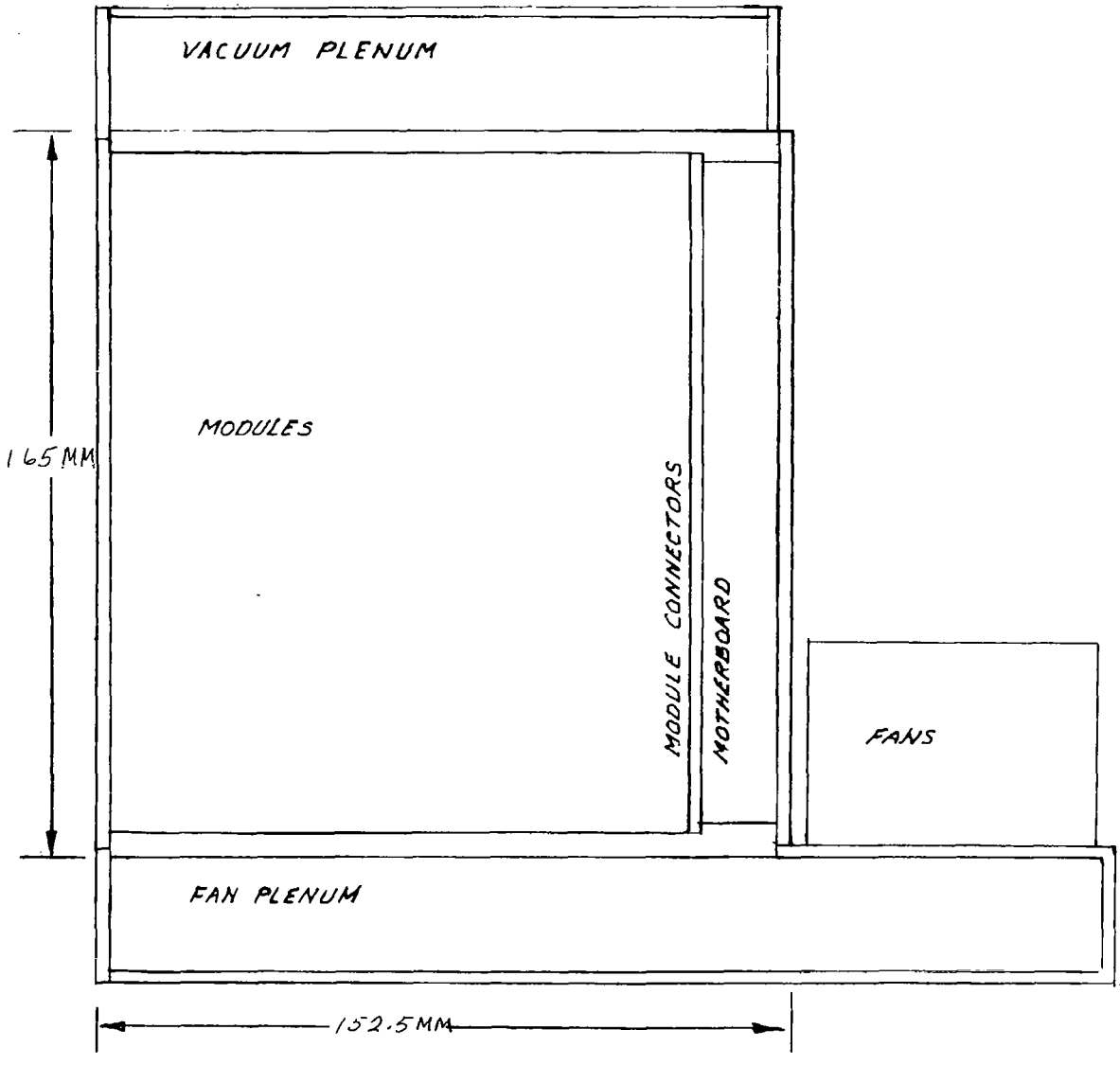
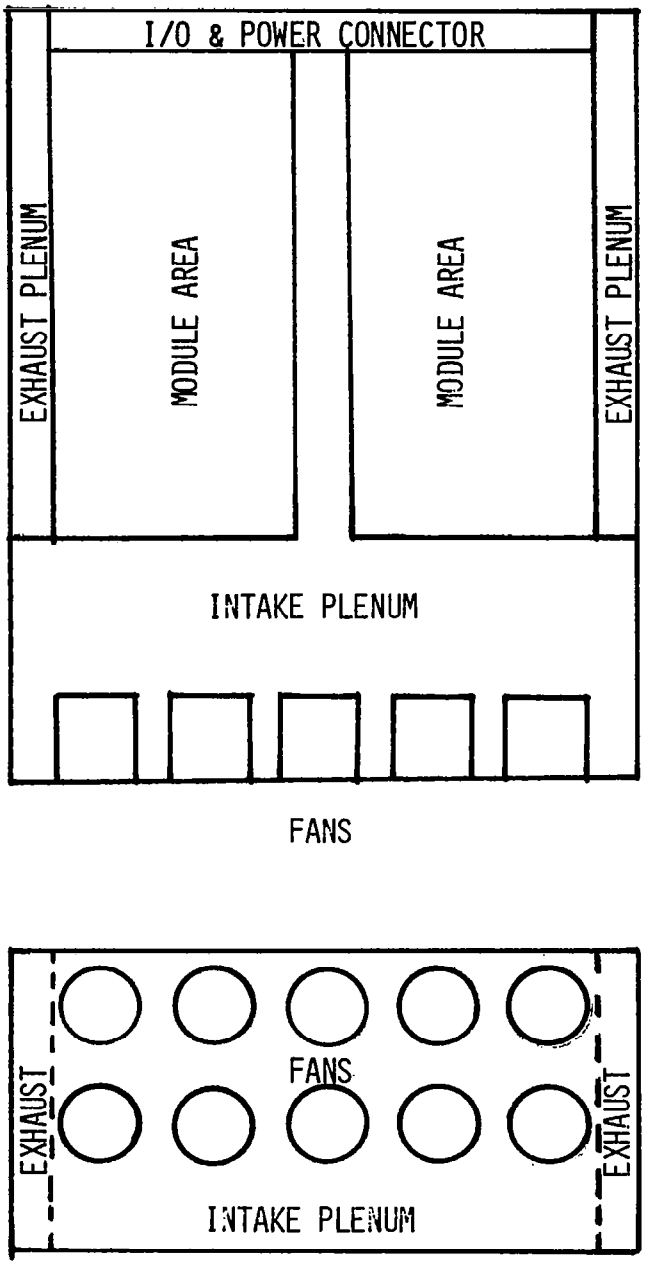


FIGURE 8.7

PRODUCTION CASE DESIGN USING ONE AND ONE-HALF ATR BOX FORMAT (496 x 390 x 19.35 mm) (TOP & FRONT VIEW SCHEMATICS)



CHAPTER 9

MULTIPROCESSOR SIZING STUDY

The applications study reported in the Chapter 3 has been useful in identifying the functional makeup of some of the important alternative system implementations that can be foreseen. The study reported in this chapter concerns the parameters of the multiprocessor that are impacted by its functional charter, including its survivability.

It is generally expedient to characterize a computer in terms of its major resources, which are processing speed, memory capacity, and input/output bandwidth. Speed is particularly difficult to quantify in the absence of benchmark programs. A common approach is to assume that the architecture is reasonably well matched to the application and simply characterize the speed in terms of operations per second (OPS). An important shortcoming of this approach is that not all operations are equally fast. Ratner et al. [4] have largely surmounted this shortcoming in their requirement study by separate designations of "short" OPS and "long" OPS. Their approach is also followed here.

In the case of a multiprocessor, the memory access bandwidth assumes a substantially larger importance than it does in a single-processor computer. The use of a serial memory access bus is of particular concern, as it can easily become the limiting factor on speed. We therefore adopt memory bus bandwidth as a primary parameter for this multiprocessor, along with the other primary factors of processing speed, memory capacity, and input/output bandwidth already mentioned.

Before proceeding to a discussion of specific parametric relationships, we shall first treat the major partitioning issues that relate the parameters of the multiprocessor to the overall functional requirements for the system. It is clear that the parameters of the multiprocessor are not uniquely determined by the system functional requirements. Following sections introduce parameters that describe internal mechanisms and functions, and exhibit approximate models for the determination of processing speed and memory bus bandwidth. The

chapter concludes with a discussion of architectural consequences of the sizing study.

9.1 System Partitioning

Chapter 1 describes the system context for the multiprocessor. The element of that discussion most important to the present subject is the significance of digital processing local to, and dedicated to, the individual subsystems. A certain amount of information processing is present in most contemporary subsystems, though most of it is analog rather than digital. With the development of integrated avionics and a reliable central computer, one could conceive of a move to centralize all information processing. To do this, however, would work against overall economy and dependability. The reasons are indicated in the chapter just cited.

On the other hand, there is nothing to prevent the use of the multiprocessor in a highly centralized system if circumstances warrant it. One example might be an experimental deployment of an integrated system in a past - generation aircraft, in which local information processing is minimal. Another example might be a future - generation aircraft whose functional scope is small enough to fit comfortably in a single computer.

Looking in the opposite direction, it is evident that the more sophisticated future applications can benefit substantially from distributed information processing. Numerous examples come to mind, ranging from the preprocessing of data from sensors such as inertial navigation units, collision avoidance receivers, and microwave landing receivers, to closed loop high frequency control of each of a redundant set of small control surfaces on control-configured aircraft. In large-scale applications such as these, the employment of distributed processing will not only offload the central computer, but will enable the inclusion at low cost of additional functions, relating to subsystem monitoring and diagnosis, that would otherwise not be cost-effective to implement. The central computer's function in a highly distributed application, although by no means negligible, would gravitate toward outer loop control and coordination, with the result that its performance capacity may not need to be any larger than that of a fully centralized system of modest scope.

According to the foregoing argument, the functional requirements for an aircraft do not translate simply into performance parameters for the multiprocessor. Moreover, the converse is also true; the performance parameters of a central computer do not per se govern the functional capacity of the system. It is therefore important to consider the impact of a range of functional requirements on the central computer's performance requirements. For this purpose, a parametric model has been developed for the two performance measures most crucial to the multiprocessor architecture: processing speed and memory bus bandwidth. Memory capacity and input/output bandwidth are more simply related to the functional requirements. The next two sections explain the origins of the parameters of the model.

9.2 Program Execution Parameters

A predominant characteristic of the multiprocessor is its retention of instructions and data in a common memory. In order to execute programs, processors must acquire instructions and data from this common memory and must transfer computed data to it. The use of cache memories local to processors tends to reduce the volume of common memory data transfers for a given function for two reasons. First is the option of storing permanently some fraction of the overall program in all of the cache memories. Second is the option of executing instructions more than once after they are brought from the common memory. The first involves the cost of replication of instructions among all processors. The second option is to allow the program segment to remain in temporary cache until it has executed some number of times. If this number is very large, the result is more of a multicomputer than a multiprocessor, and the more dedicated each processor becomes. If the program segment is executed only once before being overwritten, some repetition will still occur due to the repeated use of some of the instructions in program loops.

9.2.1 Macros

In this architecture, a certain amount of basic procedure must be stored local to the processors in order to be able to operate at all. It might be pointed out that the instructions stored permanently in cache memory act as macro extensions of the microprogram. Indeed, it is likely that certain cache macros will be moved into or out of micro-program as they become more or less utilized over the evolution of

the system. For the sake of this parametric sizing study, however, it has been assumed that the only microprogrammed macros relevant to application programs are those apt to be found in conventional computers, for example floating-point multiply and divide. To represent the performance impact of other macros, located in cache memory, we define the parameter MACRO, which is a bus bandwidth compression factor due to the use of macros. It is the ratio of the number of instructions from common memory needed to execute a program with macros, to the number needed without macros.

9.2.2 Looping

Another facet of the architecture is the principle of job step execution. Job steps are relatively short programs that generate sample-time updates of control variables. They are executed from cache memory, which is loaded from common memory for the purpose. At the end of the job step, its instructions are no longer retained, in order that the processor can seek a new job step to perform. This produces a cost in memory bus bandwidth, because cache memory must be reloaded for each job step. The main advantages in doing so are minimal reconfiguration penalty and convenient memory and processor resource sharing. During the execution of a single job step, there is a probability that some of the instructions brought into cache will be executed more than once, as in the case of a loop. A parameter called LOOP is used to represent this probability. LOOP is a compression factor like MACRO, and is the ratio of instructions fetched from common memory to the number of executions among them.

9.2.3 Bus and Cache Memory Overheads

The use of a loadable cache memory, in addition to its advantages, involves some overhead in managing the loading and storing operations with common memory. For each instruction loaded into cache memory, there is an average number of instruction times, of the order of one, used for cache management and loading delays. This number is represented by the parameter CACHE. Other overheads associated with the memory bus arise from delays involved in address transmission and data handling. These are represented by the parameters ADDRESS and DELAY. ADDRESS is a multiplier on memory bus bandwidth, and DELAY is the delay per fetched data word measured in short-instruction times.

9.2.4 Queuing

Another parameter of significance in sizing time shared resources represents excess or unused resources over and above those needed for serving useful functions. Both instruction execution and memory bus transfers are subject to queuing delays, which become excessive as the utilization approaches one hundred per cent. We therefore define a parameter QUEUE, which is the fraction of available processing resources that can be allocated to useful work,and another parameter QUEUEBUS, which is the corresponding fraction of the bus bandwidth.

9.2.5 Executive Overhead

The executive program is run each time a job step is dispatched. The program will be mostly or entirely available as a macro in permanent cache memory, but the executive data bases, such as the job queue, event queue, etc., reside in common memory. The resources consumed by the executive are dependent on queue length, the complexity of the executive function, and the number of job steps executed per unit time. The parameter SAMP represents the gross sampling rate of the computer, and is defined as the average number of job steps dispatched per second.

9.3. Function Parameters

The preceding section identified a number of parameters related to the multiprocessor architecture. This section deals with a functional breakdown of the multiprocessor's operation into four categories: application programs, executive activity, periodic test, and system redundancy management.

9.3.1 Applications

There are several dimensions to the subject of applications. They include time frame of aircraft manufacture and logevity, size and mission of aircraft, aircraft design dependence on integrated system criticality, accompanying ground aid development and its operational constraints, and flight regime (takeoff, cruise, etc.). We find a broad range of potential applications, ranging from non-critical to wholly critical. Interestingly, over this broad range, the impact on the resource parameters of the fault-tolerant system is primarily external to the central computer. This is partly explained by the fact that a substantial fraction of the central resources exist purely to establish system fault tolerance. Secondarily, the resources required

for aircraft control tend to move from the central computer out to local processors in the more sophisticated of the projected systems. All indications are that applications will nevertheless be the predominant consumers of central resources. The estimates of Ratner et al. [4] have been found to be reasonable in comparison with other analyses and estimates, when adjustments are made for overheads and decentralization. The actual numbers depend on specific applications, but one can identify likely ranges in order to establish bounds for sizing. The parameters that characterize programs are first the operations per second, which are divided into two categories for short and long operations. SHORTS and LONGS are the corresponding parameters and are measured in instructions per second. The execution time ratio of long to short operations is called R. For applications programs specifically, we use ASHORTS and ALONGS. Another functional parameter is the number of data words per unit time, for which we use DATA in words fetched or stored per second, and ADATA for applications programs.

The required memory bus bandwidth is sensitive to the parameter AMACRO, the bus compression factor for applications programs due to the use of macros. The diversity of the various applications functions and their relatively large program volume poses some challenging tradeoff problems for the multiprocessor. The basic tradeoff is between bus bandwidth and permanent cache memory. Short procedures with high usage are good candidates for inclusion as cache macros, where long, infrequent procedures are not. The dividing line depends on several factors. First, the cost of the memory bus is a nonlinear function of bandwidth. At low bandwidths, it is relatively insensitive, whereas it seems to rise steeply at a point that is estimated to be in the area of ten to twenty megabits per second. At this point, the cost of permanent cache memory in each processor becomes less severe in comparison to bus cost. Next, one considers the number of processors involved. This number multiplies the cost of permanent cache memory, whereas in common memory the cost multiplier is three for the principal memory triad in which the procedure is stored, plus any spare modules. Finally, the inclusion of procedures in permanent cache as macros inevitably reduces flexibility by increasing the penalty for changes.

Two main criteria suggest themselves for the selection of macro procedures. The first is utility, examples of which are seen in such procedures as vector and matrix manipulations, trigonometric and transcendental functions, and software module interface operations. The

second is tasks with very high sampling rates. Such tasks are expected to be relatively rare in a central computer. Additional procedures may gravitate into macro form as they become fixed for the foreseeable remainder of the life cycle.

The situation with looping is somewhat different from that of macros, in that any tradeoffs must be made in the process of generating applications programs, and are out of the system designer's hands. The looping parameter ALOOP for applications programs, then, is almost purely an *a priori* prediction of looping behavior.

9.3.2 Executive

Executive program activity has been roughly characterized in Section 9.2. The program's operation is characterized by the following parameters. The number of equivalent short operations in one dispatch is called EXEC. The production of EXEC and SAMP, the overall sampling rate, gives ESHORTS, the number of equivalent short operations per second for the executive. Most or all of the executive program will be in permanent cache memory, so that its macro compression factor, EMACRO, will be relatively small in value. The looping factor, ELOOP, will likewise be small, as the executive is a highly repetitive program. The data access is characterized by the parameter EXDAT, which is the average number of words accessed for each job dispatch. The product of EXDAT and SAMP gives EDATA, the total number of data accesses per second for the executive. The size of EXDAT is more or less proportional to the number of different tasks that the computer handles. The job and event queues need to be ordered or searched for every job step dispatch. The number of words handled per insertion or search depends on dispatch statistics and search strategy, as well as the number of tasks. Because of these complexities, and because this data volume is not a dominant factor in the total picture, no more detailed modeling is attempted of this operation at this time. We rather treat EXEC and EXDAT by estimating on the basis of past experience.

9.3.3 Test and Redundancy Management

Test programs are run periodically in processor triads to expose latent faults in all modules and buses. The frequency of testing is not directly dependent on the applications programs, but rather depends primarily on the target failure rate of the multiprocessor and the predicted module failure rates. The parameters of testing programs

are these: Each triad requires an average number of short operations denoted TEST. If there are T triads, a total of TxTEST short operations are used per test. The test frequency, TFREQ, multiplied by TxTEST gives TSHORTS, the equivalent short operation rate due to testing. The parameters TMACRO, TLOOP, and TDATA are defined similarly to their counterparts described earlier. TDATA is TxTSTDATxTFREQ.

Redundancy management programs are analogous to test programs, but serve primarily to maintain the integrity of the external system. Because of the relative simplicity of this category of programs, we will deal directly with the major parameters RSHORTS, RMACRO, RLOOP, and RDATA.

9.4 Parametric Models

In the last two sections, we have defined a family of parameters that lead to approximate models for processing speed and bus bandwidth. In this section these models are exhibited and discussed.

9.4.1 Instruction Execution Rate

The total instruction execution rate of the multiprocessor is the product of the number of processor triads and the execution rate of one processor. It will be convenient to measure execution rates in SHORTS PER SECOND.

The available execution rate must be sufficient to allow for all computation functions and overhead functions. Allowance must be made for all data accesses, during which it is assumed that no processing takes place. The total instruction execution rate is approximated roughly as follows:

$$\text{SHORTS/SEC} = \frac{1}{\text{QUEUE}} \text{ [EXECUTION RATE + DATA DELAYS]}. \qquad (1)$$

An expression for the first term is as follows:

$$\begin{aligned}\text{EXECUTION RATE} = {} & \text{ASHORTS (1 + CACHE x AMACRO x ALOOP)} \\ & + \text{ALONGS (R + CACHE x AMACRO x ALOOP)} \\ & + \text{ESHORTS (1 + CACHE x EMACRO x ELOOP)} \\ & + \text{TSHORTS (1 + CACHE x TMACRO x TLOOP)} \\ & + \text{RSHORTS (1 + CACHE x RMACRO x RLOOP)} \qquad (2)\end{aligned}$$

The terms of the form (1 + CACHE x MACRO x LOOP) represent direct executions plus a cache management overhead for those instructions that

are not already resident in cache memory. The term multiplying ALONGS is similar, except that the direct execution of a long operation takes R times as long as that of a short one. Equation (2) can be further broken down by means of parameters defined in the preceding sections:

$$\begin{aligned}\text{EXECUTION RATE} = {} & \text{ASHORTS } (1 + \text{CACHE} \times \text{AMACRO} \times \text{ALOOP}) \\ & + \text{ALONGS } (R + \text{CACHE} \times \text{AMACRO} \times \text{ALOOP}) \\ & + \text{EXEC} \times \text{SAMP } (1 + \text{CACHE} \times \text{EMACRO} \times \text{CLOOP}) \\ & + \text{TxTEST} \times \text{TFREQ } (1 + \text{CACHE} \times \text{TMACRO} \times \text{TLOOP}) \\ & + \text{RSHORTS } (1 + \text{CACHE} \times \text{RMACRO} \times \text{RLOOP}) \end{aligned} \quad (3)$$

The second term of (1) can be expanded as:

$$\text{DATA DELAYS} = (\text{ADATA} + \text{EDATA} + \text{TDATA} + \text{RDATA})\ \text{DELAY}, \quad (4)$$

and further expanded as:

$$\text{DATA DELAYS} = (\text{ADATA} + \text{EXDAT} \times \text{SAMP} + \text{TxTSTDAT} \times \text{TFREQ} + \text{RDATA})\ \text{DELAY} \quad (5)$$

9.4.2 Memory Bus Bandwidth

The memory bus bandwidth is measured in bits per second. Since the parameters defined thus far are word-oriented, it is convenient to model first WORDS PER SECOND. This must be adequate to deliver instructions and data to the cache memories. Therefore

$$\text{WORDS/SEC} = [\text{BUS INSTRUCTION WORDS} + \text{BUS DATA WORDS}]\ \frac{\text{ADDRESS}}{\text{QUEUEBUS}} \quad (6)$$

where ADDRESS is the bus addressing overhead factor, and QUEUEBUS is the associated queuing factor. The first term can be expanded as follows:

$$\begin{aligned}\text{BUS INSTR WORDS} = {} & (\text{ASHORTS} + \text{ALONGS})\ \text{AMACRO} \times \text{ALOOP} \\ & + \text{ESHORTS} \times \text{EMACRO} \times \text{ELOOP} \\ & + \text{TSHORTS} \times \text{TMACRO} \times \text{TLOOP} \\ & + \text{RSHORTS} \times \text{RMACRO} \times \text{RLOOP} \end{aligned} \quad (7)$$

The expression could be further expanded by replacing ESHORTS and TSHORTS as in (3), but this will be omitted here for the sake of brevity.

The second term, assuming all data is the same length as an instruction word, is

$$\text{BUS DATA WORDS} = \text{ADATA} + \text{EDATA} + \text{TDATA} + \text{RDATA} \quad (8)$$

The expansions for EDATA and TDATA, as in (5), are omitted.

9.4.3 Sensitivities

Given that the independent variables of the parametric models are known only to an approximation, some of the terms in the equations are difficult to predict because they are products of several parameters. For example, one term in (7), when further expanded, becomes

$$\frac{\text{ADDRESS}}{\text{QUEUEBUS}} \times \text{EXEC} \times \text{SAMP} \times \text{EMACRO} \times \text{ELOOP}. \qquad (9)$$

Suppose each parameter is estimated to within plus or minus 20%. The uncertainty range of the term is between 0.8^6 and 1.2^6, or between 0.26 and 2.98, or greater than an order of magnitude. On the other hand, the only factors common to many terms are QUEUEBUS, CACHE, ADDRESS, and DELAY. Of these, the last two are essentially hardware design parameters. CACHE is a parameter of the system software, which will be determined at a relatively early development stage. Queuing parameters are sensitive to the executive program duration, the average job step duration, and permissible dispatch delays. This issue is further discussed in Section 9.5.

9.4.4 Parameter Ranges

Table 9.1 lists some approximate ranges for the parameters defined earlier in this chapter. These ranges are based on various estimates, and the purpose of this subsection is to provide a degree of substantiation for them.

QUEUE and QUEUEBUS: (0.5, 0.7, 0.9). A good deal remains to be learned about these parameters because of their relationship to delays. This is further discussed in the next section. The ranges given are felt to be sufficiently broad to cover any probable value.

R: (5, 10, 20). The ratio of long to short instruction execution times is a hardware design parameter, unless a highly integrated processor is chosen in which these operations are previously constrained. The data format is also a factor. We presently assume a floating point format. Long instructions account for a large enough fraction of the processing task to be worth a reasonable hardware investment for the sake of speed.

CACHE: (1.0, 1.5, 2.0). The CACHE memory loading overhead depends on the bus speed, the bus delay, and the number of words to be fetched at one time. The fewer words fetched at a time, the greater the overhead. If many words are fetched, this overhead tends to

TABLE 9.1

MINIMUM, MEDIUM, AND MAXIMUM ESTIMATES FOR SPEED AND BANDWIDTH PARAMETERS

	MIN.	MED.	MAX.
QUEUE	0.5	0.7	0.9
QUEUEBUS	0.5	0.7	0.9
R	5	10	20
CACHE	1.0	1.5	2.0
ASHORTS	100K	200K	400K
ALONGS	25K	50K	100K
AMACRO x ALOOP	0.2	0.4	0.6
ADATA	10K	20K	40K
EXEC	100	200	400
SAMP	200	600	1000
EMACRO x ELOOP	0.05	0.1	0.2
EXDAT	30	60	90
TEST	2K	4K	8K
T	4	8	12
TFREQ	0.5	1	2
TMACRO x TLOOP	0.05	0.1	0.2
TSTDAT	50	100	200
RSHORTS	1K	2K	4K
RMACRO x RLOOP	0.05	0.1	0.2
RDATA	200	400	800
ADDRESS	1.2	1.4	1.6
DELAY	1	2	4

diminish until queuing delays begin to become appreciable. It is presently assumed that the typical fetch will consist of the order of eight words, and that the cache overhead will be fractionally greater than the bus latency alone. This overhead is compensated by the bandwidth reductions made possible by the cache memories.

ASHORTS: (100K, 200K, 400K operations per second). The application study by Ratner et al. [4] previously cited suggests a value for this parameter just under 200K short operations per second. This figure is based on the inclusion of some unlikely functions and the exclusion of some likely ones, in the light of more recent investigations. We feel that we can project with reasonable confidence to within a factor of two or so what the nominal sizing of the central computer should be. Estimates have been made assuming that the following capabilities will be available in the central computer:

1. Flight control, including stability augmentation, load alleviation, mode stabilization, ride improvement, and flight envelope limiting.
2. Navigation and guidance, including area navigation, inertial navigation, autoland, autopilot, and hybrid navigation.
3. Air data processing.
4. Collision avoidance processing.
5. Display data management.
6. Centralized engine management.
7. Integrated aircraft systems management, including management for fuel, weight, cabin environment, power, caution and warning, and maintenance data.

The net result is that 200K short OPS remains a reasonable estimate for an ambitious future application, assuming that local processing is available for displays, engine controls and various other dedicated subsystem support functions. An upper limit of double this figure represents a case where a larger load results either from greater complexity or from a more centralized system. A lower limit could be quite a bit smaller, especially for an experimental application, but for sizing purposes a lower limit of half the expected value is used.

ALONGS: (25K, 50K, 100K operations per second). This range is chosen by the same rationale as the previous one. The high end of the range represents a very substantial computing load, which would have a hardware impact in size and/or power if it were to be accommodated. This parameter is one that can effectively be offloaded by local processing.

AMACROxALOOP: (0.2, 0.4, 0.6). With increasing control sophistication, the benefits of macros tend to increase, in the areas of matrix, vector, trigonometric, and transcendental operations for navigation, guidance, control, and display. Rather little is known about the looping parameter despite some prior experimentation. As more macros are used, the incidence of looping will tend to become less, because highly repetitive operations tend to be the good candidates for macros. For this reason we have chosen to estimate the combined parameters, instead of each separately. The given range is a rough estimate based on Appendix B of reference [4], and on limited experimental experience.

ADATA: (10K, 20K, 40K words per second). The range given is based on the instruction execution rate. The data rate applies to those quantities that are stored and accessed from one job step to another, which will probably be an order of magnitude below the instruction rate.

EXEC: (100, 200, 400 operations per sample). The number of short operations per second required for the executive for each sample period of each application program in a current experimental implementation is of the order of one hundred. The given range is based on the expectation of a more complex job dispatch strategy.

SAMP: (200, 600, 1000 samples per second). Although the number of samples per second could be quite low in an experimental situation, a full-scale application might easily range up to the order of a thousand. At this magnitude, it could have a significant sizing impact depending on the degree to which efficient queue management strategies are developed.

EMACROxELOOP (0.05, 0.1, 0.2). Owing to its high usage and inherent repetition, the executive involves rather little overhead in cache management and memory bus traffic. It resides mostly in permanent cache memory.

EXDAT: (30, 60, 90 words per sample). This parameter is sensitive to search strategies as well as the number of functions served. The range reflects an extrapolation of current experience.

TEST (2K, 4K, 8K operations per triad iteration). This parameter represents the average number of instructions needed to conduct one test iteration on one processor or memory triad. The range reflects experience with self-test programs in the past. It may be assumed that no one iteration will test every possible element. Rather each iteration will be designed to test most elements, the remaining ones to be tested on an occasional basis. In the case of memory triads, each iteration would test a fraction of the total contents.

T: (4, 8, 12 triads). This range stems from a minimum system to the large system, including processor triads and memory triads.

TFREQ: (0.5, 1, 2 cycles per second). The test frequency is related to recovery speed, which is an important reliability parameter. This range is estimated in part through the results of reliability modeling. It will depend on module MTBF to some extent. The frequency can in principle be dropped for peak load periods of short duration, so that the figure used here could represent a minimum level, while the average frequency is boosted during the periods of lower activity. For modeling purposes, the reciprocal of TFREQ can be treated as a mean time-to-test.

TMACROxTLOOP:(0.05, 0,1, 0.2). Whether or not the test programs reside in permanent cache memory, they involve much repetition, which keeps this factor small.

TSTDAT: (50, 100, 200 words per second). Testing data from common memory should be virtually negligible.

RSHORTS: (1K, 2K, 4K operations per second).
RMACROxRLOOP: (0.05, 0.1, 0.2).
RDATA: (200, 400, 800 words per second).

These three parameters are foreseen to be small at this time, owing to the network management experience cited in Ref. 7.

ADDRESS: (1.2, 1.4, 1.6 words times per word). This memory bus overhead factor depends on the number of words transferred per access. The estimate given assumes a moderate number of the order of ten, which may be a reasonable average in order to control queuing delays.

DELAY: (1, 2, 4 operations per data word). This is a measure of processing overhead to obtain or store data from/to common memory, and is analogous to the cache memory overhead for instruction fetching. Because of queuing, this quantity is sensitive to the average number of instruction words fetched at a time, as well as the number of data words at a time.

9.5 Problems of Resource Sharing

It is well established that the sharing of resources among multiple users with random arrivals produces rapidly-mounting delays as the total utilization approaches one hundred per cent. The delays are also related to the durations of each use of the resources. The delay times can be reduced by making the resources faster, by removing users from the system, or by making each use shorter while increasing the number of uses. The latter amounts to job partitioning, which is valid if overheads are neglected.

In sizing the multiprocessor, it is assumed that users can not be removed from the system. (The use of priorities in allocating resources is to some extent an approximation of user removal, in that it favors those users whose needs are greatest). The use of job partitioning is possible, but resource access overheads must be taken into account, which acts as a limitation. It is primarily through the adjustment of resource speeds that delay criteria are met. The queuing parameters, QUEUE and QUEUEBUS, used in the preceding section, are resource speed adjustments to meet delay criteria.

The processing and memory bus bandwidth resources are both subject to queuing delays, but are substantially different from one another. The bus is a single resource, whereas there are several processing resources operating simulataneously. The bus may be used for short, fixed time periods or alternatively for periods distributed over a wide range. This factor is determined by cache and data management algorithms. The processors are used for periods determined largely by the applications programs, where the distribution is apt to have a relatively narrow range. The computer's performance must satisfy near worst-case conditions for processing resource delays, and average conditions for bus delays.

The problem of multiple processing resources with low-variance time periods has been studied recently by Lala [6] as a doctoral thesis project. One of the interesting results from this work is that near

worst-case delays are not as sensitive to system load as average delays are. This has both good and bad consequences. In order to reduce delays, the processors must be made substantially faster, which can be viewed as bad. Once this is done, however, the computer can probably be loaded with additional jobs without severe impact on delay. This is a complex issue, made more complex by the inclusion of overheads such as the executive program, priorities, and the variety of sampling rates involved. Until this problem is studied further, we must anticipate that dispatch delays of at least several times the average job step length will occur often if the computer is reasonably well loaded.

The bus problem more nearly resembles the classical single-server queuing problem, except that the usage durations can be affected by management algorithms. Pending further study, it is assumed that the bus access delays will be minimized by assigning an upper limit to the number of words that constitute a single transaction. This number must be a compromise between delays caused by long transactions and delays caused by access overheads for short transactions. The average delay is expected to follow the classical pattern, and should amount to a fraction of the transaction length.

9.6 Three Size Estimates

Table 9.2 summarizes sizing estimates for three variants of the multiprocessor architecture, corresponding to minimum, medium, and maximum applications loads, respectively. In all cases, a conservative estimate is made with respect to queuing and cache management.

9.6.1 Small System

For the small system, it is assumed that relatively little emphasis need be placed on enhancing macro and looping operations, so that the corresponding factors will be maximal for the sake of economy. Testing and redundancy management activities are assumed to be relatively light. The long-instruction ratio is taken to be medium, representing rather efficient microprograms. The data delay parameter is also assumed to be the medium value, by virtue of a microprogram-supported bus interface and a relatively short limitation on bus access times. About five percent of the processing resources are attributable to executive, test, and redundancy management activity. The remainder is more or less equally divided between applications short and long instructions. The modest bus bandwidth is largely used by applications

TABLE 9.2

THREE MULTIPROCESSOR SIZE ESTIMATES

	SMALL SYSTEM	BASELINE APPLICATION	LARGE OR CENTRALIZED APPLICATION
ASHORTS, ALONGS, ADATA, SAMP, T	MIN	MED	MAX
EXEC, EXDAT	MIN	MED	MED
TEST, TSTDAT, RSHORTS, RDATA	MIN	MED	MAX
TFREQ	MIN	MED	MED
ALL MACRO x LOOP	MAX	MED	MIN
R	MED	MED	MIN
ADDRESS	MED	MED	MED
QUEUE, QUEUEBUS	MIN	MIN	MIN
CACHE	MAX	MAX	MAX
DELAY	MED	MED	MIN
SHORT MOPS REQUIRED	1.1	2.4	3.1
BUS KWORDS/SEC REQUIRED	270	483	610
BUS M BITS/SEC REQUIRED @ 16 BITS PER WORD	4.3	7.7	9.8
COMMON MEMORY CAPACITY	16K	32K	64K
PERMANENT CACHE CAPACITY	4K	6K	8K
WRITABLE CACHE CAPACITY	1K	2K	2K
INPUT/OUTPUT BANDWIDTH M BITS/SEC	0.5	0.5	1.0

short instructions. The memory and I/O estimates given reflect a modestly sized application such as a digital flight control system for a conventional short-haul aircraft.

9.6.2 Baseline Application

For a reasonably ambitious integrated distributed aircraft system, we use the medium applications sizing parameters. A greater emphasis is now placed on macros and looping, as reflected in the larger cache memories. Greater criticality is assumed here, and therefore a higher test frequency. No change, however, is assumed in the data delay and the long-instruction ratio. The sizing in all respects is roughly double that of the small system with essentially no hardware design differences other than module count and memory bus bandwidth. The processing resources are now becoming heavily expended in long instructions, and an increase in their speed would significantly reduce processing requirements.

9.6.3 Large or Centralized Applications

In the absence of significant local processing in the baseline application, or for a larger application than presently anticipated, we assume the maximal applications parameters. In order to accommodate this additional load without unduly stressing the memory bus bandwidth, we assume quite extensive macros. We further assume microprograms and hardware as needed to reduce the long-instruction ratio. Concessions are also assumed in hardware and software to minimize data access delays. At this point, executive, test, and redundancy management account for almost a quarter of the processing resources. The remainder are once again balanced between applications longs and shorts. The memory bus bandwidth is dominated by applications shorts, executive data, and applications data, in that order.

9.7 Summary

By a process of estimation and approximate analysis, we have arrived at rough projections for the desired sizing of a multiprocessor family. We anticipate that a production version would be designed to support the baseline application, although at the present time, the option is open to place greater emphasis on the long-instruction ratio, by providing hardware aids to multiplication, division, and normalization, for example. Likewise the memory bus interface might be enhanced

by dedicated hardware.

The new baseline processor executes a short instruction from cache memory in about two microseconds, and has a cache memory capacity of about 8K fixed and about 2K writable words of 16 bits. The bus interface is quite efficient, able to perform the equivalent of a direct cache memory access in about two microseconds.

The baseline memory module would contain 8K or 16K words, according to the total capacity required and the state of semiconductor memory technology at the time of design.

The baseline memory bus bandwidth is about ten megabits per second. Work is currently in progress to better define the speed-cost tradeoffs in this speed range. The nominal I/O bandwidth is about one megabit per second, which is not expected to pose any significant problems.

We can also begin to visualize the size of an experimental model, whose mission would be oriented to the small system. The processing speed would be about the same as the baseline, but the long-instruction ratio and the memory bus interface delay would probably not be pushed to their eventual limits. The cache memory would be somewhat smaller than the baseline. The experimental version would, if feasible, utilize the full baseline bus bandwidths, in order to demonstrate the feasibility of expanding the computer capability by the addition of more modules.

CHAPTER 10

DEMONSTRATION OF AN EXPERIMENTAL FAULT-TOLERANT MULTIPROCESSOR EMULATION

This contract had as part of its goal the construction of hardware to serve as a test bed for the architectual concepts of the system. An existing experimental fault-tolerant multiprocessor emulator was used, and certain other test equipment was built for this purpose. Tests were conducted demonstrating many of the design features and also pointing to some design problems. Experimental goals, results and conclusions are discussed in this chapter.

10.1 Experimental Goals of the Demonstration

The experimental design and the exercises performed were chosen to test specific aspects of the system design. It was desired that the experiments be of an integrated nature, that is, one apparatus closely patterned after the projected final product, capable of testing design concepts much as they would be integrated in the final product. Despite this, certain parts of the demonstration were separated out for isolated technical feasibility demonstration.

10.1.1 Synchronous Processor Triad Operation in a Multiprocessor Environment

Simultaneous operation of several triads of processors, each processor of a triad being synchronized with its partners, is a design requirement of the proposed design. This problem involves initial synchronization of three processors to start a triad, maintenance of synchronization over the long term, replacement of a processor with a spare, and bringing the new member into synch with its new partners. Additionally, mechanisms for operating multiple triads, such that there is no hardware interference, needed to be tested. Finally, the mechanisms for assigning a processor to a triad and to a particular bus needed to be tested.

10.1.2 Synchronous Memory Triad Operation

Simultaneous operation of several triads of memory is similarly a design requirement of the proposed design. Synchronous operation of three memory modules needed to be tested. The ability to assign modules to different portions of the address space also required testing. Finally, the ability to assign a memory module to a specific bus required verification.

10.1.3 High Speed Redundant Serial Bus

The high speed redundant serial bus is the heart of the proposed architecture. For this bus to be viable, several features needed to be demonstrated:

1. High speed synchronous transmission on three parallel buses from the three different elements of a triad with transmitters located at different points along the buses.
2. The passive arbitration scheme for allocating the use of the bus to particular triads for short periods of time needed to be tested within the context of extremely high bit rates. Additionally, the impact of the arbitration scheme on processor triad efficiency needed to be studied.
3. The ability to successfully mask bus errors from a single bus of three active buses needed exploration.
4. The ability to successfully protect the bus system from errant processors and memories required verification.

10.1.4 Demonstrate Diagnostic Capabilities for Hardware Faults

The proposed system includes hardware devoted to fault detection and isolation. It is of course desirable that this hardware be in some sense minimal or as little as possible while at the same time providing complete coverage. It was desired that the demonstration show the adequacy of placing error detection devices in only the processor modules. Additionally, it is necessary that the experimental system exhibit the ability to detect and recover from any fault of a large class of faults with minimal knowledge of the different mechanisms which contribute to that larger class of faults. For example, in order for the system to be viable, it is necessary that any processor submodule fault be covered and that this be done without an exhaustive

list of all possible processor submodule faults and their contributing mechanisms. Detection and recovery from both hard failures and transient failures needed to be demonstrated.

10.1.5 Demonstrate Software Framework for a Fault-Tolerant System

In order for the proposed system to function correctly it is necessary that the core software be perfect. It was important to show that the proposed system is sufficiently simple and clear that the required perfect software can probably be produced.

10.1.5.1 Multiprocessor Executive

A simple executive for handling task dispatch in a multiprocessor environment needed to be developed. This executive should assure that all of the processor triads are able to work together without distructive interference or deadlock, while at the same time being simple, easy to verify, and reasonably efficient.

10.1.5.2 Automatic Configuration Control

The philosophy for managing the available redundancy is based on the desire that all redundancy mangement, error detection, configuration control and recovery be invisible to the applications program. Automatic configuration control, operating invisibly while application code runs, even in the presence of faults in the hardware, must be demonstrated.

10.1.5.3 Diagnostic and Latency Testing

Diagnostic code is required to pinpoint the sources of known errors and to unearth or discover latent faults. This code should run in conjunction with automatic configuration control to provide a fault-free environment for the applications code to run in and to provide completely invisible error detection and recovery from faults.

10.1.5.4 Automatic Use of Cache Memory

Because of the relatively slow serial bus between common memory and processor modules each processor is equipped with a small private cache memory. As with the cache of large machines, the use of this cache should be as transparent as possible to the applications code.

Unlike large machines, a completely hardware-based scheme for cache management cannot be used, due to cost and complexity. A software-based cache management technique is required which is invisible to the user within the overall system software structure, while being reasonably efficient at the same time.

10.1.5.5 Applications Program Support

The final software structure in which an applications programmer must operate is bounded by various constraints imposed by the requirements dictated from the above. It needed to be demonstrated (1) that a programmer could work with these constraints given normal software support in the form of assemblers and various utilities, and in the form of executive supported macro functions, and (2) that the programmer would perceive these constraints not as awkward difficulties to be worked around but as a natural framework in which to work and in which the task of software generation has been made substantially easier.

10.1.6 Typical Applications Environment

In order to demonstrate fully all of the features discussed in the preceding section, within the context of a reasonably integrated experimental apparatus, it is necessary that some of the effort be devoted to creating a simulated application environment which might be typical of projected situations for the final product. Within this context it should be possible to verify the predicted impact of subjective design judgements on the ease of use of this type of system, and to unearth unforeseen operational problems.

10.1.7 Fault-Tolerant Clock

A fault-tolerant clock or time base is required by the system. An experimental demonstration of a highly redundant, extremely accurate clocking system capable of serving this function is required.

10.1.8 Redundant Power

Reliable power is required by any electronic assembly hoping to demonstrate high reliability itself. A demonstration of a viable approach to redundant power management should be made.

10.2 Experimental Results

In order to demonstrate and validate as many of the design concepts as possible, an experimental apparatus was used which was able to emulate many of the design features of the proposed system. This demonstration was of an integrated nature in that the experimental setup duplicated much of the environment which a final product of this nature might encounter, and was therefore able to verify not only the separate design pieces forming the whole, but was also able to confirm predicted interactions between disjoint pieces, and in some case unearth unexpected interactions.

The basic experimental apparatus consisted of a fault-tolerant multiprocessor, modeled along the lines of the proposed system. The multiprocessor served as the control computer for a Boeing 707 aircraft as provided in simulation by the Draper Laboratory's hybrid computer. The fault-tolerant multiprocessor consisted of 14 National Semiconductor IMP-16 based processor modules, seven common memory modules of 2K x 16 words, two I/O ports, and six I/O nodes. There was additionally a monitor processor for observing the operation of the system. The processor modules included 1K RAM/1K ROM cache memory storage. With the 14 processor modules it was possible to operate up to 4 triads of processors simultaneously. With the seven RAM modules it was possible to operate two memory triads. The redundant data busing system was triply redundant and each attached module had two Bus Guardian Units associated with it for protecting the bus system. An I/O node remote from the fault-tolerant multiprocessor and local to the hybrid computer provided A/D and D/A interfacing to the simulated aircraft as shown in Fig. 10.1.

This equipment was designed and constructed under an Independent Research and Development program at the Draper Laboratory, and most of the core software algorithms were developed under a National Science Foundation Grant. The digital autopilot software and its attendant system integration were activities of this contract.

10.2.1 Processor Triad Operation

Synchronous operation of three processors as a triad was achieved early, with little difficulty. The ability to rapidly replace and resynchronize a failed or deactivated processor within a triad was also demonstrated with little difficulty. The mechanisms for arbitra-

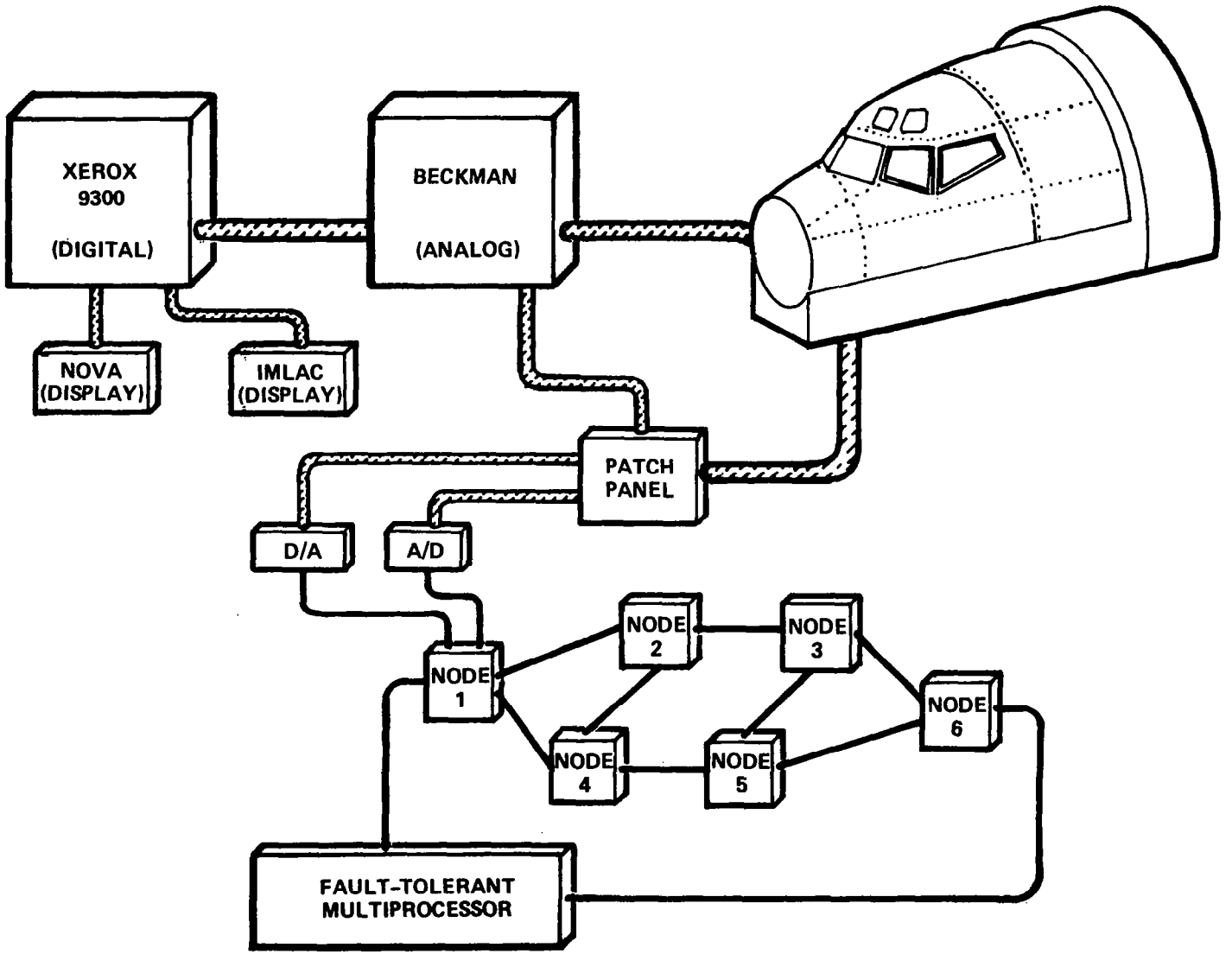


Figure 10.1. Demonstration System Diagram.

ting which processor triad was to receive control of the bus system was refined. No unexpected interferences between processors were observed except for those associated with the competitive poll. The experience with the competitive poll is discussed in greater detail in Section 10.2.3.

10.2.2 Memory Triad Operation

Synchronous operation of three memory modules as a triad was also achieved early. Memory modules of a triad were automatically synchronized, and the mechanism worked flawlessly and without difficulty. No difficulty was encountered in assigning modules to serve different portions of the common memory address space or in assigning modules to particular buses.

10.2.3 High Speed Redundant Serial Bus

Synchronous transmission and reception of identical signals from three different sources located at different points along the bus was demonstrated within the integrated system for data rates up to 2 MHz. Bus length in the experimental equipment was roughly 2 meters, about three times the bus length for the proposed breadboard. Data bits were placed on the data bus on the rising edge of the system clock and strobed on the falling edge. The clock was a square wave. Most of the propagation delay from transmitter to receiver was associated with active circuit device delays. Because of these delays data bits were barely stable at the receiver output when they were strobed, thus limiting bus speed to roughly 2 MHz. They remained stable well past the next rising edge of the system clock. The proposed specification moves the strobe to a full bit period after a data bit transmission begins, thus allowing an additional 1/2 bit period for permissible propagation delays. Additionally, better device selection can reduce propagation delays by a factor of 2. Thus, by combining both improvements it should be possible to operate the serial bus at 8-10 MHz. An electrical model of a 10 MHz bus was constructed and tested at up to 15 MHz to verify proper electrical performance of the higher speed bus. These tests indicated that an 8-10 MHz bus is feasible.

Since propagation delay along the short bus did not account for the major portion of the transmission delays, the siting along the bus of the 3 transmitters had no effect on the results.

The passive arbitration scheme for allocating the use of the bus to particular processor triads worked as intended. However, the competitive poll number of the experimental system did not include a dynamic priority adjustment to boost a triad's priority if it lost a poll. The competitive poll number included only a field to assign a nominal fixed priority to a triad and a unique identifier field. When several processor triads were operating with the same nominal priority, it was found that there was a tendency for the triads with high identifier numbers to lock out those with lower numbers. For this reason it was necessary to add a dynamic priority adjustment field to the competitive poll number in the specification. This field value is boosted by 1 each time a triad loses a poll competition to another triad of equal assigned priority, saturating at a maximum value. It is reset to zero when the triad wins a poll. Thus, a triad which wins has its dynamic priority field set to zero. If it re-enters competition for the bus before other triads of equal nominal priority have been serviced, they are assured of service before that triad is again granted access to the bus.

All receiving elements of the system used two-out-of-three voters to mask bus errors. The error masking system worked as intended with no unexpected results.

The Bus Guardian approach to bus protect performed according to projection. Failed modules could be easily and reliably separated from the bus sytem.

10.2.4 Fault Diagnostic Capabilities

Each processor module of the experimental system included special circuitry for noting and recording disagreements among the three copies of each bus line. All other modules or receving elements had only error masking circuits.

The error detection circuitry functioned as expected. Most faults manifest themselves as bus errors, and were therefore easily detected. Certain classes of latent faults were detected by diagnostic programs which basically forced bus errors if a latent fault existed. Records kept by diagnostic programs and fault isolation procedures enabled the successful test of techniques for locating both transient and hard failures.

Most faults were detectable as one of a large class of faults. For example, all processor failures were detected at the bus without

the aid of special diagnostic code to test the processor or knowledge of the fault mechanism. Some special attention to specific failure modes and effects was required to devise latent fault detection programs. While code was not written for unearthing all possible latent faults, sufficient latent testing code was written so as to establish considerable confidence that all latent faults could have been tested and detected.

The bus isolation mechanism served as intended and was able to isolate all processor failures from the bus system.

This integrated systems demonstration illustrated all significant aspects of the fault-tolerant computer architecture. It demonstrated the hardware capability to mask faulty unit outputs in the short run, and the capability to detect the fault, isolate the unit, and to reorganize so as to restore system health, all concurrent with normal program activity.

10.2.5 Software Experience

The software that was provided for the demonstration consisted principally of executive or system software and applications software. Executive or system software was written and debugged by staff throughly familiar with the experimental hardware and design objectives. The applications software was provided by a team which was briefed only in general terms as to the nature of fault recovery mechanisms and the overall system architecture. The applications software team was provided with detailed explanations of the executive-to-applications interfaces and executive services, as well as a reasonably short list of programming constraints.

10.2.5.1 Multiprocessor Executive

The multiprocessor executive provided a simple task dispatch mechanism. Tasks awaiting their time of execution were organized in a queue sorted by scheduled start time. As processor triads became free (having finished a previous task) they would consult this list and take the next scheduled job. Jobs could be inserted into any relative position of the time queue as long as it remained properly sorted. Executive functions provided for the routine iterative scheduling of the same job step, as might be required for an autopilot iteration for example. Alternatively, any job, by a call to the executive, could

insert a job into the time queue. The executive also handled the removal of a job from the queue when it was taken up for execution.

In addition to the time queue, the executive handled an event queue. Jobs in the event queue have their execution blocked waiting for a particular event to occur. When the event did occur, the affected job was moved from the event queue to the top of the time queue. Jobs could be inserted into the event queue by any job, through a call to the executive. Events could be signaled by the executive or by another job through a call to the executive.

The executive also provided interfaces for all I/O traffic, common memory to/from cache data transfers, real time clock, and for other relatively simple functions commonly thought of as executive-related.

Critical to the success of the demonstration were the executive functions which provided for automatic error logging and recovery. Executive functions performed all common memory to/from cache transfers, and all I/O. During these functions any errors that might occur will become visible. The executive handled the proper logging of the error, scheduled recovery action, and, via voting, masked the error for the applications task which was using the executive function. Thus, to the applications task, error handling is completely invisible. Additionally, since hardware monitoring is used, error checking, error masking and majority voting do not impact the applications execution speed.

The executive schedules error diagnostics, latent test routines, and error recovery routines, using basically the same mechanisms used to schedule applications tasks. These executive tasks, running concurrently with the applications tasks, but in different processor triads, maintain the system, repairing faults, searching out latent failures, configuring processor triads and memory triads, and starting and stopping triads as required. Thus in the background, behind the system application, continuous activity is in progress to maintain the integrity of the system, an integrity that assures faultless and error-free execution of applications software.

An executive providing these functions was written for the experimental test hardware. Although it was not a complete executive, in that only representative latent faults were tested, the executive did provide the basic facilities for providing error free execution of both executive and applications code. The software framework for

latent test procedures was fully developed, although it was only sparsely populated. Error detection and recovery from all classes of faults was demonstrated in the simulated environment without interfering with the appications tasks.

10.2.5.2 Cache Memory Management

The experimental hardware and the proposed future system both have a common memory shared by all processor triads and private cache memories which are part of the processor modules. Programs are executed exclusively out of a processor's cache memory. Clearly, the burden of program loading from common memory, program overlaying, and other functions associated with bringing sections of code from common memory to the cache for execution could not be placed on the applications coding.

In the experimental computer, a software cache-memory management system was provided as part of the executive. At the subroutine call interface, conventions were adopted that provided for the automatic loading of called routines. A last used, first out algorithm cleared space in the cache if unused space was not available. If a calling routine was dropped from the cache to make room for loading of the called routine, it is reloaded by the subroutine return interface.

The efficiency of this process of loading instructions into the cache before execution will depend a great deal on the number of times an instruction is executed each time it is brought from common memory. Each word brought from common memory will take 5.25 μsec in the proposed sytem. Thus one triad executing 190K instructions per second could completely fill the bus capacity. In the experimental system, it was found that the applications programs executed between 10 and 40 instructions for every instruction brought from common memory. If an overall average of 20 can be maintained in the proposed system, a processor triad now projected to have a raw computing power of 300K instructions per second would load the bus with 15K instruction fetches per second. With reasonable allowances made for data transfers and queueing overheads, this suggests a maximum capacity of 4 or 5 processor triads before saturating the memory bus.

10.2.5.3 Application Software

The application which was coded for this experiment involved an autopilot function for a Boeing-707-like aircraft and various instrument redundancy management tasks. The autopilot function performed was control wheel steering. This included rate stability augmentation as well as attitude hold loops. The redundancy management tasks involved various analytic redundancy management calculations so as to detect and identify simulated failures in dual sensors. Figures 10.2, 10.3 and 10.4 show flow diagrams for the autopilot functions used.

The application programming team was familiarized with the overall operating system and the details of the interfaces between applications software and the executive. Particular attention was given to an explanation of the cache memory management techniques and the resulting constraints on the applications software. The programming team was familiar with real-time control and therefore accepted as normal the automatic scheduling of iterative tasks and the overall iteration/job step orientation of the executive.

Surprising little detail was given concerning the triple-redundant nature of the processing taking place, or as regards the fault location and isolation techniques used by the executive. Indeed some of the programmers had virtually no acquaintance with the fault handling procedures. Since no effort was made to hide these details from the applications programmers, their deletion from the overall programmer orientation can probably be explained by the press of time and the fact that except for personal curiosity they had no real reason to know how fault tolerance was achieved. The goal had been to provide a system in which fault-tolerant mechanisms were invisible to the applications coder, and to the extent we succeeded there was no need to explain the mechanisms to the programmer.

The programmers had no greater difficulty generating the applications code for this machine than they would have had for simple machines of similar instruction sets. The visible parts of the executive did appear as positive assistance and enabled them to produce the multi-loop sampled data control algorithms with little difficulty. The use of the cache memory management techniques chosen did not seem to hinder or to make easier the task of coding these algorithms.

Once written, the applications code ran concurrent with executive fault-detection and isolation code in systems of two, three, or

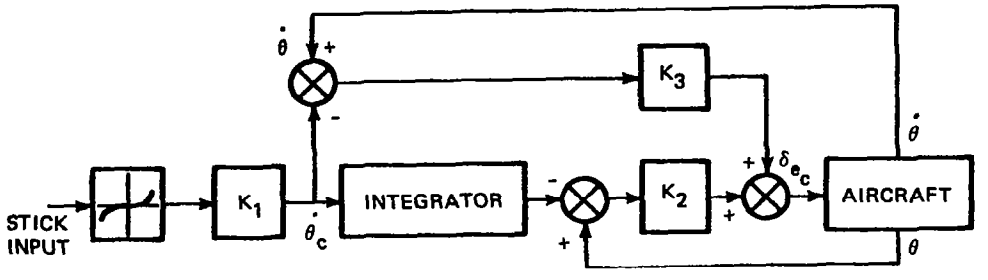


Fig. 10.2 Pitch Control.

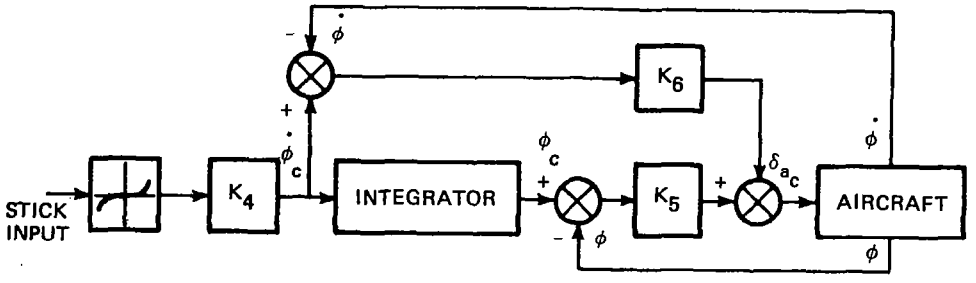


Fig. 10.3 Roll Control.

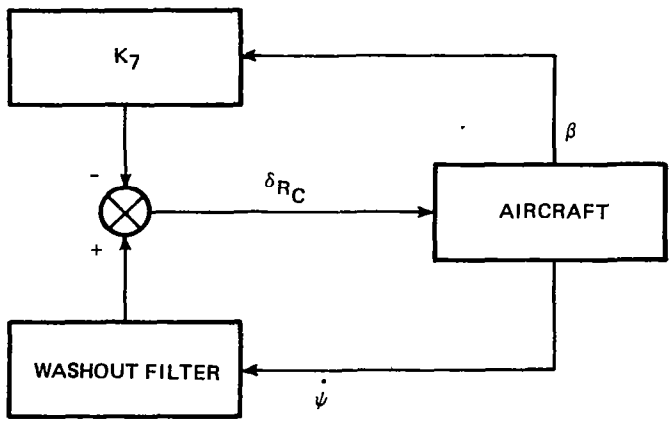


Fig. 10.4 Yaw Damping.

four triads of processors. Fault detection and recovery enabled the system to tolerate multiple serial errors without impact on the application. Indeed, the system's ability to run despite hardware faults provided a system with virtually 100% availability.

The minimum system capable of executing all applications tasks and the background systems management routines was a three-triad system. The computer could tolerate failures of 5 of the 14 processors before impacting the performance. Below three triads, the autopilot function was maintained at real time, and other functions slowed to conserve computational resources. At one triad, all but the autopilot functions stopped. The control wheel steering and stability augmentation were then maintained at real time and consumed all of the system computational power.

10.2.5.4 Software Reliability Experience

The difficulty in obtaining reliability in the demonstration software can be thought of as indicative of the problems associated with producing reliable software for a future system. It is of course true that a full system will be more complex, but it is also true that software verification aids will be better.

The basic core executive and configuration controller was roughly 3K words of coding. This was produced in assembly language and debugged largely in simulated situations in the monitoring processor and not even in the actual experimental multiprocessor. Because of poor visibility into the functioning of the individual processors, the system needed to be virtually bug-free before installation in the multiprocessor. The individual processors could not be single stepped, for example, and there were no indications of internal states to debug from.

Experience showed that, in the code produced and throughly tested before installation, there were three undetected program errors. Of these errors two were discovered within the first days of operation. The remaining error in an unused module of the executive remained undetected for a month. Since that time no new errors have been discovered over the 6 months of operation.

It can be projected from this experience that it is not impractical to expect that the core software can be produced error-free with reasonable investments in verification and testing.

The applications code has not been as error free as the core software. Most of the errors have been algorithmic in nature, with incorrectly chosen gains for example. This type of error is more difficult to weed out without full operation in the simulation. All errors in the application programs were corrected after less than a week of introduction involving three or four simulator sessions. Again the proposed system will clearly be more complex, but extrapolating from present experience suggests that error-free applications programming is not unobtainable.

10.2.7 Fault-Tolerant Clock Experience

The system test bed did not provide for testing of clocking concepts. Consequently, these have been tested so far in stand-alone configurations and on the CERBERUS multiprocessor built and used in connection with earlier fault-tolerant architectural experiments at the Draper Laboratory. For a discussion of the present status of experience, refer to Volume II.

10.2.8 Redundant Power Experience

No redundancy of power conversion or distribution was attempted in the test bed. The present approach is therefore still largely conceptual, except that a protective module power interface has been built and preliminarily tested. Power redundancy is further discussed in Volume II.

10.3 Demonstration Conclusions

The experimental apparatus constructed and tested served as a basic validation tool for most of the design concepts of the proposed system. While certain problems or unexpected results were encountered, these were largely minor and easily corrected. The overall agreement between projected performance and actual operation of the experimental hardware further serves to bolster confidence in the design methodology.

Operation of the integrated emulation of the hardware in the 707 applications environment demonstrated the overall suitability of this class of designs to aircraft control situations.

APPENDIX I

The following program, 'states', solves the multiprocessor Markov model described in Section 5.3. To recapitulate briefly, there are 146 states in the model. Each state is uniquely defined by 8 parameters: the number of undetected and detected failures in processor, memory, bus and bus guardian units respectively. The system starts out in 'All Good' state. The final state is the 'System Fail' state. Therefore initially the 'All Good' probability is unity and all other state probabilities are zero. The program, first of all, computes all the non-zero state transition rates. Having computed the state transition matrix which has the dimensions 146 by 146 it computes the state probability vector iteratively as a function of time by multiplying the transition matrix Q, the old probability vector SP and a time-step 'DELTA'.

The program accepts a number of variables as input. These are defined below:

Numloop:	Number of times the program has to be repeated with a different set of inputs each time.

The following set of input variables have to be defined 'Numloop' times.

Iter & Delta:	Iteratively update the state probability vector 'Iter' times using a time-step 'DELTA' milliseconds.
NSKIP:	Print the results every 'NSKIP' iterations.
P,M,B,G:	Initial system configuration defining number of processor, memory bus, and bus guardian units.
FP,FM,FB,FG:	Failure rates per hour of the respective units.
FGD,FGE:	Failure rate per hour of BGU in the disable and enable modes respectively.

RP,RM,RB,RG: Recovery rates per hour of the same units.

The program outputs the following state probabilities:

PGOOD: Probability of the 'All Good' stae.

PFAIL: Probability system has failed.

PSUCCESS: 1 - System Fail Probability.

P1,P2,P3: Probability of having exactly one two or three (detected plus undetected) failures.

PNM: Probability of having N undetected and M detected failures for various N and M combinations.

A listing of the program "States" follows.

SOURCE LISTING

```
NMBR LEVEL NEST

 100              STATES2:  PROCEDURE OPTIONS(MAIN);
       1
 200   1                    DECLARE STATE(145) CHAR(8) INITIAL(
       1
       1                       '00000000', '00000001', '00000002', '00000003', '00000010', '00000011',
       1                       '00000012', '00000020', '00000021', '00000030', '00000100', '00000101',
       1                       '00000102', '00000110', '00000111', '00000120', '00000200', '00000201',
       1                       '00000210', '00000300', '00001000', '00001001', '00001002', '00001010',
       1                       '00001011', '00001020', '00001100', '00001101', '00001110', '00001200',
       1                       '00002000', '00002001', '00002010', '00002100', '00003000', '00010000',
       1                       '00010001', '00010002', '00010010', '00010011', '00010020', '00010100',
       1                       '00010101', '00010110', '00010200', '00011000', '00011001', '00011010',
       1                       '00011100', '00012000', '00020000', '00020001', '00020010', '00020100',
       1                       '00021000', '00100000', '00100001', '00100002', '00100010', '0010001I',
       1                       '00100020', '00100100', '00100101', '00100110', '00100200', '00101000',
       1                       '00101001', '00101010', '00101100', '00102000', '00110000', '00110001',
       1                       '00110010', '00110100', '00111000', '00200000', '00200001', '00200010',
       1                       '00200100', '00201000', '01000000', '01000001', '01000002', '01000010',
       1                       '01000011', '01000020', '01000100', '01000101', '01000110', '01000200',
       1                       '01001000', '01001001', '01001010', '01001100', '01002000', '01010000',
       1                       '01010001', '01010010', '01010100', '01011000', '01100000', '01100001',
       1                       '01100010', '01100100', '01101000', '02000000', '02000001', '02000010',
       1                       '02000100', '02001000', '10000000', '10000001', '10000002', '10000010',
       1                       '10000011', '10000020', '10000100', '10000101', '10000110', '10000200',
       1                       '10001000', '10001001', '10001010', '10001100', '10002000', '10010000',
       1                       '10010001', '10010010', '10010100', '10011000', '10100000', '10100001',
       1                       '10100010', '10100100', '10101000', '11000000', '11000001', '11000010',
       1                       '11000100', '11001000', '20000000', '20000001', '20000010', '20000100',
       1                      '20001000');
       1
2800   1                    DECLARE WORD CHAR(8);
       1
2900   1                    DECLARE 1 NUM(145),
       1                              2 U FIXED DECIMAL(2) INITIAL
       1                                   ((35)0, (15)1, (5)2, (15)1, (10)2, (15)1, (15)2, (15)1, (20)2),
       1                              2 D FIXED DECIMAL(2) INITIAL
       1                                   (0,(2)(1,2,3),2,3,(2)(3,1,2,3,(2)(2,3,3)),
       1                                   3,2,(4)3,0,1,2,1,2,2,(2)(1,(3)2),2,0,(4)1,0,1,2,1,2,2,(2)(1,(3)2),
       1                                   2,(2)(0,(4)1),0,1,2,1,2,2,(2)(1,(3)2),2,(3)(0,(4)1),0,1,2,
       1                                   1,2,2,(2)(1,(3)2),2,(4)(0,(4)1));
       1
3700   1                    DECLARE (JP,JM,JB,JG,KP,KM,KB,KG) FIXED DECIMAL(1);
3800   1                    DECLARE (ZJP,ZJM,ZJB,ZJG,ZKP,ZKM,ZKB,ZKG) CHAR(4);
3900   1                    DCL Q(146,146) FLOAT DECIMAL(16);
4000   1                    DCL SP(146) FLOAT DECIMAL(16);
4100   1                    DCL PX(146) FLOAT DECIMAL(16);
4200   1                    DCL (A,PGOOD,PSUCCESS) FLOAT DECIMAL(16);
```

```
NMBR LEVEL NEST

 1          DCL MDX FLOAT DECIMAL(6);
 1          DCL IFAIL FIXED DECIMAL(3) INITIAL(146);
 1          DCL (PU,MU,BU,GU,PD,MD,BD,GD) FIXED DECIMAL(1);
 1          DCL (P,M,B,G) FIXED DECIMAL(2);
 1          DCL (WORD1,WORD2,WORD3,WORD4) CHAR(8);
 1          DCL (ZPU,ZMU,ZBU,ZGU,ZPD,ZMD,ZBD,ZGD) CHAR(4);
 1          DCL (DELTA,TIME,ITER) FIXED DECIMAL(10);

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                                 ;
;            /* THE FOLLOWING DEFINES ALL POSSIBLE SYSTEM STATES SUBJECT TO                       ;
;                                                 U<=2 AND U+D<=3.                                ;
;            I = 1;                                                                               ;
;            DO JP = 0 TO 2;  && X400 $$                                                          ;
;              DO JM = 0 TO 2;  && X300 $$                                                        ;
;                 DO JB = 0 TO 2;  && X200 $$                                                     ;
;                    DO JG = 0 TO 2;  && X100 $$                                                  ;
;                          DO KP = 0 TO 3;  && X40 $$                                             ;
;                             DO KM = 0 TO 3;  && X30 $$                                          ;
;                                DO KB = 0 TO 3;  && X20 $$                                       ;
;                                   DO KG = 0 TO 3;  && X10 $$                                    ;
;                        IF I>145 THEN GO TO X400;                                                ;
;                                   IF(JP+JM>2) THEN GO TO X300;                                  ;
;                                   IF(JP+JM+JB>2) THEN GO TO X200;                               ;
;                                   NUM.U(I) = JP+JM+JB+JG;                                       ;
;                                   IF(NUM.U(I)>2) THEN GO TO X100;                               ;
;                                   IF(KP+NUM.U(I)>3) THEN GO TO X40;                             ;
;                                   IF(KP+KM+NUM.U(I)>3) THEN GO TO X30;                          ;
;                                   IF(KM+KP+KB+NUM.U(I)>3) THEN GO TO X20;                       ;
;                                   NUM.D(I) = KG+KP+KM+KB;                                       ;
;                                   IF(NUM.U(I)+NUM.D(I)>3) THEN GO TO X10;                       ;
;            ZJP = JP;                                                                            ;
;            ZJM = JM;                                                                            ;
;            ZJB = JB;                                                                            ;
;            ZJG = JG;                                                                            ;
;            ZKP = KP;                                                                            ;
;            ZKM = KM;                                                                            ;
;            ZKB = KB;                                                                            ;
;            ZKG = KG;                                                                            ;
;              STATE(I) =SUBSTR(ZJP,4)||SUBSTR(ZJM,4)||SUBSTR(ZJB,4)||SUBSTR(ZJG,4)|              ;
;                    SUBSTR(ZKP,4)||SUBSTR(ZKM,4)||SUBSTR(ZKB,4)||SUBSTR(ZKG,4);                  ;
;             && PUT EDIT (I,STATE(I))  (SKIP(1),F(3),X(3),A(8));                                 ;
;                PUT EDIT (ZJP,ZJM,ZJB,ZJG,ZKP,ZKM,ZKB,ZKG) (A(4));$$                             ;
;                                 I = I+1;                                                        ;
;                                 X10: END;                                                       ;
;                               X20: END;                                                         ;
;                             X30: END;                                                           ;
;                           X40: END;                                                             ;
;                         X100: END;                                                              ;
;                       X200: END;                                                                ;
;                     X300: END;                                                                  ;
;                   X400: END;   */                                                               ;
;                                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
PL/I OPTIMIZING COMPILER          STATES2: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

                                     /* READ IN PARAMETER VALUES */
         1
    9500 1                          GET DATA(NUMLOOP);
    9600 1                          PUT DATA(NUMLOOP);
         1
    9800 1                          DO N=1 TO NUMLOOP;
         1  1
   10000 1  1                         GET DATA(ITER,DELTA,NSKIP);
   10100 1  1                         PUT DATA(ITER,DELTA,NSKIP);
   10200 1  1                         GET DATA(P,M,B,G);
   10300 1  1                         PUT DATA(P,M,B,G);
   10400 1  1                         GET DATA(FP,FM,FB,FG,FGD,FGE,RP,RM,RB,RG);
   10500 1  1                         PUT DATA(FP,FM,FB,FG,FGD,FGE,RP,RM,RB,RG);
         1  1
         1  1                            /* THE FOLLOWING DEFINES THE ELEMENTS OF THE TRANSITION INTENSITY X Q *
         1  1
   10700 1  1                         MATRIX      */
         1  1                            Q = 0;
   10900 1  1                         DO I = 1 TO 145;            /*  X1000  */
   11000 1  12                          IF NUM.D(I) = 3 THEN GO TO S_FAIL;          /* ONLY TRANSITION FROM D=3 IS (FAIL) */
   11100 1  12                          #                                             TO S(FAIL)      */
         1  12                                    /* OBTAIN THE PRESENT STATE'S PARAMETERS */
         1  12                           PU = SUBSTR(STATE(I),1,1);
   11400 1  12                          MU = SUBSTR(STATE(I),2,1);
   11500 1  12                          BU = SUBSTR(STATE(I),3,1);
   11600 1  12                          GU = SUBSTR(STATE(I),4,1);
   11700 1  12                          PD = SUBSTR(STATE(I),5,1);
   11800 1  12                          MD = SUBSTR(STATE(I),6,1);
         1  12
   11900 1  12                          BD = SUBSTR(STATE(I),7,1);
   12000 1  12                          GD = SUBSTR(STATE(I),8,1);
   12100 1  12                          PDX = GD*P/(P+M);
   12200 1  12                          MDX = GD*M/(P+M);
   12300 1  12                          IF (B-BD)>3 THEN BX=B-BD; ELSE BX=3;
         1  12
         1  12                            /* ANOTHER FAILURE IN ANY OF THE FOLLOWING CASES KILLS THE SYSTEM */
         1  12
   12500 1  12                          IF(NUM.U(I) = 2) THEN GO TO S_FAIL;
   12600 1  12                          IF(NUM.U(I)+NUM.D(I)=3) THEN GO TO S_FAIL;
         1  12
         1  12                            /* FIND THE STATES TO WHICH TRANSITIONS TAKE PLACE FROM STATE I */
         1  12
   12800 1  12                          PU = PU+1;
   12900 1  12                          MU = MU+1;
   13000 1  12                          BU = BU+1;
   13100 1  12                          GU = GU+1;
   13200 1  12                          ZPU = PU;
   13300 1  12                          ZMU = MU;
   13400 1  12                          ZBU = BU;
   13500 1  12                          ZGU = GU;
   13600 1  12                          PU = PU-1;
   13700 1  12                          MU = MU-1;
```

```
PL/I OPTIMIZING COMPILER          STATES2: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

   13800   1  12                     BU = BU-1;
   13900   1  12                     GU = GU-1;
   14000   1  12                     WORD1 = SUBSTR(ZPU,4)||SUBSTR(STATE(I),2,7);
   14100   1  12                     WORD2 = SUBSTR(STATE(I),1,1)||SUBSTR(ZMU,4)||SUBSTR(STATE(I),3,6);
   14200   1  12                     WORD3 = SUBSTR(STATE(I),1,2)||SUBSTR(ZBU,4)||SUBSTR(STATE(I),4,5);
   14300   1  12                     WORD4 = SUBSTR(STATE(I),1,3)||SUBSTR(ZGU,4)||SUBSTR(STATE(I),5,4);
   14400   1  12                     I1 = SNUM(WORD1);
   14500   1  12                     I2 = SNUM(WORD2);
   14600   1  12                     I3 = SNUM(WORD3);
   14700   1  12                     I4 = SNUM(WORD4);
           1  12
           1  12                        /* THE FOLLOWING 4 TRANSITIONS ARE FROM STATES U=0 */
           1  12
   14900   1  12                     IF NUM.U(I) = 0 THEN DO;
   15000   1  123                        Q(I,I1) = (P-PD-PDX)*FP;
   15100   1  123                        Q(I,I2) = (M-MD-MDX)*FM;
   15200   1  123                        Q(I,I3) = (B-BD)*FB;
   15300   1  123                        Q(I,I4) = 2*(P-PD-PDX+M-MD-MDX)*FG;
   15400   1  123                        PUT DATA(Q(I,I1),Q(I,I2),Q(I,I3),Q(I,I4)) SKIP(1);
   15500   1  123                        END;
           1  12
           1  12                        /* THE FOLLOWING 20 TRANSITIONS ARE FROM STATES U=1 */
           1  12
   15700   1  12                     IF NUM.U(I) = 1 THEN DO;              /*  X500  */
   15800   1  123                        IF PU = 1 THEN DO;
   15900   1  1234                          Q(I,I1) = (P-PD-PDX-3)*FP;
   16000   1  1234                          Q(I,I2) = (M-MD-MDX)*FM;
   16100   1  1234                          Q(I,I3) = (B-BD-2)*FB;
   16200   1  1234                          Q(I,I4) = 2*(P-PD-PDX+M-MD-MDX-3)*FG + 4*FGE;
   16300   1  1234                          Q(I,IFAIL) = 2*FP+2*FB+4*FGD;
   16400   1  1234                          END;
           1  123
   16500   1  123                        IF MU = 1 THEN DO;
   16600   1  1234                          Q(I,I1) = (P-PD-PDX)*FP;
   16700   1  1234                          Q(I,I2) = (M-MD-MDX-3)*FM;
   16800   1  1234                          Q(I,I3) = (B-BD-2)*FB;
   16900   1  1234                          Q(I,I4) = 2*(P-PD-PDX+M-MD-MDX-3)*FG + 4*FGE;
   17000   1  1234                          Q(I,IFAIL) = 2*FM+2*FB+4*FGD;
   17100   1  1234                          END;
           1  123
   17200   1  123                        IF BU = 1 THEN DO;
   17300   1  1234                          Q(I,I1) = (1-2/BX)*(P-PD-PDX)*FP;
   17400   1  1234                          Q(I,I2) = (1-2/BX)*(M-MD-MDX)*FM;
   17500   1  1234                          Q(I,I3) = (B-BD-1)*FB - 6*FB/BX;
   17600   1  1234                          Q(I,I4) = (2-4/BX)*(P-PD-PDX+M-MD-MDX)*FG + (4/BX)*(P-PD-PDX+M-MD-MDX)*FGE;
   17700   1  1234                          Q(I,IFAIL) = (3/BX)*(2*FB+2*(P-PD-PDX)*FP/3 + 2*(M-MD-MDX)*FM/3 +
           1  1234                                       4*(P-PD-PDX+M-MD-MDX)*FGD/3);
   17900   1  1234                          END;
           1  123
   18000   1  123                        IF GU = 1 THEN DO;
   18100   1  1234                          Q(I,I1) = (P-PD-PDX-P/(P+M))*FP;
   18200   1  1234                          Q(I,I2) = (M-MD-MDX-M/(P+M))*FM;
   18300   1  1234                          Q(I,I3) = (B-BD-2)*FB;
   18400   1  1234                          Q(I,I4) = 2*(P-PD-PDX+M-MD-MDX)*FG - (4*FGE+FGD);
   18500   1  1234                          Q(I,IFAIL) = 4*FGD + FGE + P*FP/(P+M) + M*FM/(P+M) + 2*FB;
```

```
PL/I OPTIMIZING COMPILER          STATES2: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

   18600    1  1234                          END;
            1  123
   18700    1  123                        PUT DATA(Q(I,I1),Q(I,I2),Q(I,I3),Q(I,I4),Q(I,IFAIL))  SKIP(1);
            1  123
   18800    1  123        X500:           END;
            1  12
   18900    1  12                      IF NUM.U(I)>0 THEN GO TO RECOVER; ELSE GO TO X1000;
            1  12
            1  12                        /* THE FOLLOWING ARE TRANSITIONS FROM STATES U=2 OR D=3
            1  12
   19100    1  12                      OR U+D=3 TO  SYSTEM FAIL STATE   */
            1  12
            1  12        S_FAIL:       Q(I,IFAIL) = (P-PD-PDX-PU)*FP + (M-MD-MDX-MU)*FM + (B-BD-BU)*FB
            1  12                        + 2*(P-PD-PDX-PU+M-MD-MDX-MU)*FG;
   19400    1  12                      PUT DATA(Q(I,IFAIL)) SKIP(1);
   19500    1  12                      IF NUM.U(I)>0 THEN GO TO RECOVER; ELSE GO TO X1000;
            1  12
            1  12                        /* THE FOLLOWING TRANSITIONS TAKE PLACE WHEN A FAULT IS DETECTED
            1  12                        AND THE SYSTEM RECONFIGURES */
            1  12
   19800    1  12        RECOVER:      IF PU>0 THEN DO;
   19900    1  123                        PU = PU-1;
   20000    1  123                        PD = PD+1;
   20100    1  123                        ZPU = PU;
   20200    1  123                        ZPD = PD;
   20300    1  123                        PU = PU+1;
   20400    1  123                        PD = PD-1;
   20500    1  123                        WORD1 = SUBSTR(ZPU,4)||SUBSTR(STATE(I),2,3)||
            1  123                                SUBSTR(ZPD,4)||SUBSTR(STATE(I),6,3);
   20700    1  123                        I1 = SNUM(WORD1);
            1  123
   20800    1  123                        Q(I,I1) = PU*RP;
   20900    1  123                        PUT DATA(Q(I,I1)) SKIP(1);
   21000    1  123                        END;
            1  12
   21100    1  12                      IF MU>0 THEN DO;
   21200    1  123                        MU = MU-1;
   21300    1  123                        MD = MD+1;
   21400    1  123                        ZMU = MU;
   21500    1  123                        ZMD = MD;
   21600    1  123                        MU = MU+1;
   21700    1  123                        MD = MD-1;
   21800    1  123                        WORD2 = SUBSTR(STATE(I),1,1)||SUBSTR(ZMU,4)||
            1  123                                SUBSTR(STATE(I),3,3)||SUBSTR(ZMD,4)||
            1  123                                SUBSTR(STATE(I),7,2);
            1  123
   22100    1  123                        I2 = SNUM(WORD2);
   22200    1  123                        Q(I,I2) = MU*RM;
   22300    1  123                        PUT DATA(Q(I,I2)) SKIP(1);
   22400    1  123                        END;
            1  12
   22500    1  12                      IF BU>0 THEN DO;
   22600    1  123                        BU = BU-1;
   22700    1  123                        BD = BD+1;
   22800    1  123                        ZBU = BU;
```

```
PL/I OPTIMIZING COMPILER          STATES2: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

   22900   1  123                        ZBD = BD;
   23000   1  123                        BU = BU+1;
   23100   1  123                        BD = BD-1;
   23200   1  123                        WORD3 = SUBSTR(STATE(I),1,2)||SUBSTR(ZBU,4)||
           1  123                                SUBSTR(STATE(I),4,3)||SUBSTR(ZBD,4)||
           1  123                                SUBSTR(STATE(I),8,1);
   23500   1  123                        I3 = SNUM(WORD3);
   23600   1  123                        Q(I,I3) = BU*RB;
   23700   1  123                        PUT DATA(Q(I,I3)) SKIP(1);
   23800   1  123                        END;
           1  12
   23900   1  12                       IF GU>0 THEN DO;
   24000   1  123                        GU = GU-1;
   24100   1  123                        GD = GD+1;
   24200   1  123                        ZGU = GU;
   24300   1  123                        ZGD = GD;
   24400   1  123                        GU = GU+1;
   24500   1  123                        GD = GD-1;
   24600   1  123                        WORD4 = SUBSTR(STATE(I),1,3)||SUBSTR(ZGU,4)||
           1  123                                SUBSTR(STATE(I),5,3)||SUBSTR(ZGD,4);
   24800   1  123                        I4 = SNUM(WORD4);
           1  123
   24900   1  123                        Q(I,I4) = GU*RG;
   25000   1  123                        PUT DATA(Q(I,I4)) SKIP(1);
   25100   1  123                        END;
           1  12
   25200   1  12        X1000:          END;
           1  1
           1  1                        /* MAKE THE DIAGONAL ELEMENTS OF Q SUCH THAT ROWS OF Q SUM TO UNITY */
           1  1
   25400   1  1                      Q = Q*DELTA/3600000.;              /*DELTA IS IN MILLISECONDS*/
   25500   1  1                      DO I = 1 TO 146;
   25600   1  12                       A = 0;
   25700   1  12                       DO J = 1 TO 146;
   25800   1  123                        A = A + Q(I,J);
   25900   1  123                        END;
           1  12
   26000   1  12                       Q(I,I) = 1 - A;
   26100   1  12                       PUT DATA(Q(I,I)) SKIP(1);
   26200   1  12                       END;
           1  1
           1  1                        /*  INITIALIZE P VECTOR  */
           1  1
   26400   1  1                      SP = 0;
   26500   1  1                      SP(1) = 1;
   26600   1  1                      TIME = 0;
   26700   1  1                      NSKIP1 = NSKIP;
   26800   1  1                      DO K = 1 TO ITER;
   26900   1  12                       TIME = TIME+DELTA;
           1  12
           1  12                          /* COMPUTE NEW P VECTOR */
```

```
PL/I OPTIMIZING COMPILER          STATES2: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

   27100   1  12                      PX = 0;
   27200   1  12                      DO I = 1 TO 146;
   27300   1  123                       DO J = 1 TO 146;
   27400   1  1234                        PX(I) = Q(J,I)*SP(J) + PX(I);
   27500   1  1234                        END;
   27600   1  123                       END;
           1  12
   27700   1  12                      SP = PX;
   27800   1  12                      NSKIP1 = NSKIP1-1;
   27900   1  12                      IF(NSKIP1>0) THEN GO TO X2500;
   28000   1  12                      NSKIP1 = NSKIP;
   28100   1  12                      PGOOD = SP(1);  PFAIL = SP(146);  PSUCCESS = 1-SP(146);
   28200   1  12                      P01 = 0; P10 = 0;
   28300   1  12                      P20 = 0; P11 = 0;
   28400   1  12                      P02 = 0;
   28500   1  12                      P12 = 0; P21 = 0; P03 = 0;
   28600   1  12                      DO I=2 TO 145;
   28700   1  123                       IF NUM.U(I)=1 & NUM.D(I)=0 THEN P10 = P10+SP(I);
   28800   1  123                       IF NUM.U(I)=0 & NUM.D(I)=1 THEN P01 = P01+SP(I);
   28900   1  123                       IF NUM.U(I)=2 & NUM.D(I)=0 THEN P20 = P20+SP(I);
   29000   1  123                       IF NUM.U(I)=1 & NUM.D(I)=1 THEN P11 = P11+SP(I);
   29100   1  123                       IF NUM.U(I)=0 & NUM.D(I)=2 THEN P02 = P02+SP(I);
   29200   1  123                       IF NUM.U(I)=1 & NUM.D(I)=2 THEN P12 = P12+SP(I);
   29300   1  123                       IF NUM.U(I)=2 & NUM.D(I)=1 THEN P21 = P21+SP(I);
   29400   1  123                       IF NUM.U(I)=0 & NUM.D(I)=3 THEN P03 = P03+SP(I);
   29500   1  123                       END;
           1  12
   29600   1  12                      P1 = P10+P01;
   29700   1  12                      P2 = P20+P11+P02;
   29800   1  12                      P3 = P21+P12+P03;
   29900   1  12                      PUT EDIT('SYSTEM RELIABILITY STATISTICS AT TIME = ',TIME,'  MSEC.,  STEP = ',
           1  12                               DELTA,'  MSEC.')  (PAGE,LINE(2),A,F(10),A,F(10),A);
   30100   1  12                      PUT DATA(PGOOD) SKIP(4); PUT DATA(PFAIL) SKIP(2); PUT DATA(PSUCCESS) SKIP(2);
   30200   1  12                      PUT DATA(P1) SKIP(2); PUT DATA(P2) SKIP(2); PUT DATA(P3) SKIP(2);
   30300   1  12                      PUT DATA(P10) SKIP(4); PUT DATA(P01) SKIP(2);
   30400   1  12                      PUT DATA(P20) SKIP(4); PUT DATA(P11) SKIP(2); PUT DATA(P02) SKIP(2);
   30500   1  12                      PUT DATA(P21) SKIP(4); PUT DATA(P12) SKIP(2); PUT DATA(P03) SKIP(2);
           1  12
   30600   1  12          X2500:      END;
           1  1
   30700   1  1                     IF N=NUMLOOP THEN DO;
   30800   1  12                        PUT DATA(SP);
   30900   1  12                        END;
   31000   1  1                     END;
           1
           1                        /* FUNCTION STATE NUMBER  */
           1
   31200   1            SNUM:     PROCEDURE(WORDX);
           2
   31300   2                      DCL WORDX CHAR(8);
           2
   31400   2                      J = 0;
```

```
PL/I OPTIMIZING COMPILER          STATES2: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

   31500   2                      DO J1 = 128,64,32,16,8,4,2,1;
   31600   2  1                     IF J+J1<145 THEN
           2  1                                 IF WORDX>=STATE(J+J1) THEN J=J+J1;
   31800   2  1                     END;
           2
   31900   2                      RETURN(J);
   32000   2                      END SNUM;
           1
   32100   1                      END STATES2;
```

APPENDIX II

The following program, 'Cards5', solves the multiprocessor Markov model described in Section 5.4. This program is similar to "States. The only major difference is that it solves a simpler 11-state model. In addition, a variable time-step is used to update the state probability vector. The program accepts the following additional input variables that are different from those used by the preceding program.

N1,N2,NSKIP1,NSKIP2: The program, Cards5, first updates the state probability vector N1 times using a time-step Delta milliseconds and prints the results every NSKIP1 iterations.

Then it updates the vector N2 times using a time-step 4 Delta milliseconds and prints the results every NSKIP2 iterations.

The other input and output variables for Cards5 are same as those for "States". A listing of the Cards5 program follows.

```
PL/I OPTIMIZING COMPILER          CARDS5: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

                    SOURCE LISTING

NMBR LEVEL NEST

  100               CARDS5:   PROCEDURE OPTIONS(MAIN);

                              /*   THIS PROGRAM SOLVES A CARDS MARKOV MODEL  FOR LACK OF        */
                              /*   PERFECT COVERAGE.   SYSTEM STATE PROBABILITIES ARE           */
                              /*   COMPUTED AS A FUNCTION OF TIME. THIS MODEL IS AN 11-STATE    */
                              /*   MODEL USED FOR HARD FAILURE ANALYSIS.                        */
         1
         1                    /*                    DEFINITIONS                                */
         1
         1                    /*   Q  = STATE TRANSITION MATRIX (WITH EACH ROW SUMMING TO UNITY */
         1
         1                    /*   SP = STATE PROBABILITY VACTOR                                */
         1
         1                    /*   PX = INTERMEDIATE STATE PROB. VECTOR (WORK SPACE)            */
         1
         1                    /*   L  = DIMENSION OF Q, SP AND PX                               */
         1
         1                    /*   DELTA = INITIAL TIME STEP FOR ITERATIVE SOLUTION             */
         1
         1                    /*   TIME  = TIME OF SOLUTION                                     */
         1
         1                    /*   N1    = #ITERATIONS FOR WHICH TIME-STEP IS DELTA             */
         1
         1                    /*   N2    = #ITERATIONS FOR WHICH TIME-STEP IS DELTA*MM          */
         1
         1                    /*   NSKIP1 = SKIP NSKIP1 ITERATIONS BETWEEN PRINTING RESULTS FOR */
         1
         1                    /*                                          FIRST N1 ITERATIONS*/
         1
         1                    /*   NSKIP2 = SKIP NSKIP2 ITERATIONS BETWEEN PRINTING RESULTS FOR */
         1
         1                    /*                                          NEXT N2 ITERATIONS */
         1
         1                    /*                    ALGORITHM                                  */
         1
         1                    /*   SP(N+1) = Q*SP(N)*DELTA                                      */
         1
 2200    1                 DCL Q(11,11)           FLOAT DECIMAL(16);
         1
 2300    1                 DCL SP(11)             FLOAT DECIMAL(16);
 2400    1                 DCL PX(11)             FLOAT DECIMAL(16);
 2500    1                 DCL (A,PGOOD,PSUCCESS) FLOAT DECIMAL(16);
 2600    1                 DCL (P,M,B)            FIXED DECIMAL(2);
 2700    1                 DCL (DELTA,TIME)       FIXED DECIMAL(10);
```

```
PL/I OPTIMIZING COMPILER          CARDS5: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

    2800   1                     DCL (N1,N2)            FIXED DECIMAL(6);

                                    /* READ-IN DATA */

           1
    3200   1                     GET DATA(NUMLOOP);
    3300   1                     PUT DATA(NUMLOOP);
           1
    3500   1           NUM:      DO N = 1 TO NUMLOOP;
           1  1
    3700   1  1                   PUT EDIT('N = ',N) (PAGE,LINE(2),A,F(2));
    3800   1  1                   GET DATA(P,M,B);
    3900   1  1                   PUT DATA(P,M,B)    SKIP(2);
    4000   1  1                   GET DATA(FP,FM,FB);
    4100   1  1                   PUT DATA(FP,FM,FB) SKIP(2);
    4200   1  1                   GET DATA(RP,RM,RB);
    4300   1  1                   PUT DATA(RP,RM,RB) SKIP(2);
    4400   1  1                   GET DATA(N1,N2,DELTA,NSKIP1,NSKIP2,MM);
    4500   1  1                   PUT DATA(N1,N2,DELTA)      SKIP(2);
    4600   1  1                   PUT DATA(NSKIP1,NSKIP2,MM) SKIP(2);
           1  1
    4800   1  1                   Q = 0;
    4900   1  1                   L = 11;

                                    /*  FAILURE TRANSITIONS  ------------ RECOVERY TRANSITIONS */
           1  1
    5300   1  1                   Q(1,2)  =  P*FP;                 Q(2,1)  =  RP;
    5400   1  1                   Q(1,3)  =  M*FM;                 Q(3,1)  =  RM;
    5500   1  1                   Q(1,4)  =  B*FB;                 Q(4,1)  =  RB;
           1  1
    5700   1  1                   Q(2,5)  =  (P-3)*FP;             Q(5,2)  =  2*RP;
    5800   1  1                   Q(2,6)  =  M*FM;                 Q(6,2)  =  RM;
    5900   1  1                   Q(2,10) =  (B-2)*FB;             Q(10,2) =  RB;
    6000   1  1                   Q(2,11) =  2*(FP+FB);
           1  1
    6200   1  1                   Q(3,6)  =  P*FP;                 Q(6,3)  =  RP;
    6300   1  1                   Q(3,7)  =  (M-3)*FM;             Q(7,3)  =  2*RM;
    6400   1  1                   Q(3,8)  =  (B-2)*FB;             Q(8,3)  =  RB;
    6500   1  1                   Q(3,11) =  2*(FM+FB);
           1  1
    6700   1  1                   Q(4,8)  =  (1.-2./B)*M*FM;       Q(8,4)  =  RM;
    6800   1  1                   Q(4,9)  =  (B-1.-6./B)*FB;       Q(9,4)  =  2*RB;
    6900   1  1                   Q(4,10) =  (1.-2./B)*P*FP;       Q(10,4) =  RP;
    7000   1  1                   Q(4,11) =  (2./B)*(P*FP+M*FM+3*FB);/*(3/B)((2/3)*P*FP+(2/3)*M*FM+2FB*/
           1  1
    7200   1  1                   Q(5,11) = (P-2)*FP + M*FM + B*FB;
    7300   1  1                   Q(6,11) = (P-1)*FP + (M-1)*FM + B*FB;
    7400   1  1                   Q(7,11) = P*FP + (M-2)*FM + B*FB;
    7500   1  1                   Q(8,11) = P*FP + (M-1)*FM + (B-1)*FB;
    7600   1  1                   Q(9,11) = P*FP + M*FM + (B-2)*FB;
    7700   1  1                   Q(10,11) = (P-1)*FP + M*FM + (B-1)*FB;
```

```
)L/I OPTIMIZING COMPILER          CARDS5: PROCEDURE OPTIONS(MAIN);
)RAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

           1  1                        /* INITIALIZE P VECTOR */
           1  1
    8000   1  1                      SP = 0;
    8100   1  1                      SP(1) = 1;
    8200   1  1                      TIME = 0;
           1  1
    8400   1  1                      Q = Q*DELTA/3600000.;          /*DELTA IS IN MSECS*/
           1  1
    8600   1  1          L2:         DO K2 = 1 TO 2;
    8700   1  12                       IF K2>1 THEN IF N2=0 THEN GOTO L5;
    8800   1  12                       IF K2>1 THEN DO;          /* INCREMENT STEP-SIZE MM-FOLD SECOND TIME*/
    8900   1  123                          Q = Q*MM;
    9000   1  123                          END;
           1  12
    9100   1  12                       PUT EDIT(' ') (SKIP(4),A);
    9200   1  12                       DO I = 1 TO L;
    9300   1  123                        Q(I,I) = 0;
    9400   1  123                        A = 0;          /* COMPUTE DIAGONAL ELEMENTS OF Q SUCH THAT ROWS SUM TO 1*/
    9500   1  123                        DO J = 1 TO L;
    9600   1  1234                         A = A+Q(I,J);
    9700   1  1234                         END;
           1  123
    9800   1  123                        Q(I,I) = 1.-A;
    9900   1  123                        PUT DATA(Q(I,I)) SKIP(1);
   10000   1  123                        END;
           1  12
           1  12                         /* PUT DATA(Q) */
           1  12
   10200   1  12          L5:         IF K2=1 THEN DO;
   10300   1  123                          M1 = N1;
   10400   1  123                          NSKIP = NSKIP1;
   10500   1  123                          END;
           1  12
   10600   1  12                       ELSE DO;
   10700   1  123                        M1 = N2;
   10800   1  123                        NSKIP = NSKIP2;
   10900   1  123                        END;
           1  12
   11000   1  12                       NSKIP3 = NSKIP;
   11100   1  12                       J1 = 1;
           1  12
   11300   1  12          L1:         DO K1 = 1 TO M1;
   11400   1  123                          TIME = TIME+DELTA;
           1  123
           1  123                           /* COMPUTE NEW P VECTOR */
           1  123
   11600   1  123                          PX = 0;
   11700   1  123                          DO I=1 TO L;
   11800   1  1234                           DO J=1 TO L;
   11900   1  12345                            PX(I) = Q(J,I)*SP(J) + PX(I);
   12000   1  12345                            END;
   12100   1  1234                           END;
```

```
PL/I OPTIMIZING COMPILER          CARDS5: PROCEDURE OPTIONS(MAIN);
DRAPER LAB LISTING FORMATTER        CHARLES STARK DRAPER LABORATORY, INC.    CAMBRIDGE, MASSACHUSETTS

   NMBR LEVEL NEST

   12200   1  123                        SP=PX;
   12300   1  123                        NSKIP3 = NSKIP3-1;
   12400   1  123                        IF (NSKIP3>0) THEN GOTO X100;
   12500   1  123                        NSKIP3 = NSKIP;
   12600   1  123                        PGOOD = SP(1);
   12700   1  123                        PFAIL = SP(11);
   12800   1  123                        PSUCCESS = 1.-SP(11);
   12900   1  123                        P1 = SP(2)+SP(3)+SP(4);
   13000   1  123                        P2 = SP(5)+SP(6)+SP(7)+SP(8)+SP(9)+SP(10);
           1  123
   13200   1  123                        J1 = J1-1;
   13300   1  123                        IF (J1>0) THEN GOTO X99;
   13400   1  123                        J1 = 12;                          /* PRINT 12 SETS OF DATA ON A PAGE  */
           1  123
   13600   1  123                        PUT EDIT('STEP-SIZE =',DELTA,' MSEC,   N = ',N) (PAGE,LINE(2),A,F(10),
           1  123                                                                          A,F(2));
           1  123
   13800   1  123        X99:
           1  123                        SECONDS = TIME/1000;
   14000   1  123                        IHOURS = SECONDS/3600;
   14100   1  123                        IMINUTES = (SECONDS-3600*IHOURS)/60;
   14200   1  123                        ISECONDS = SECONDS-3600*IHOURS-60*IMINUTES;
   14300   1  123                        PUT EDIT('TIME = ',IHOURS,' HR ',IMINUTES,' MIN ', ISECONDS,
           1  123                         ' SEC   PGOOD = ',PGOOD,' PFAIL = ',PFAIL,' PSUCCESS = ',PSUCCESS)
           1  123                         (SKIP(2),A,F(2),A,F(2),A,F(2),A,E(11,5),A,E(11,5),A,E(11,5));
   14600   1  123                        PUT EDIT('P1 = ',P1,' P2 = ',P2) (SKIP(1),COLUMN(32),A,E(11,5),
           1  123                                                                          A,E(11,5));
           1  123
   14800   1  123        X100:           END L1;
           1  12
   14900   1  12                      DELTA = DELTA*MM;
   15000   1  12                      PUT DATA(SP);
   15100   1  12                      END L2;
           1  1
   15200   1  1                     END NUM;
           1
   15300   1                     END CARDS5;
```

REFERENCES

1. Smith, T.B., "A Damage-and-Fault-Tolerant Input/Output Network," IEEE Transactions on Computers, Vol. C-24 No. 5, May, 1975.

2. Hopkins, A.L. and T.B. Smith, "The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor," IEEE Transactions on Computers, Vol. C-24 No. 5, May, 1975.

3. Pierce, W.H., Failure-Tolerant Computer Design, Academic Press, New York, 1965, pp. 78-114.

4. Ratner, R.S., E.B. Shapiro, H.M. Zeidler, S.E. Wahlstrom, C.B. Clark, and J. Goldberg, "Design of a Fault-Tolerant Airborne Digital Computer, Vol. II - Compuational Requirements and Technology." NASA CR-132253, 1973.

5. Hamilton, M., and S. Zeldin, "Higher Order Software - A Methodology for Defining Software," IEEE Trans. Softw. Eng. Vol. SE-2, No. 1, March, 1976.

6. Lala, J., "Performance Evaluation of a Multiprocessor in a Real Time Environment," MIT Doctoral Thesis. Draper Laboratory Report T-622, February, 1976.

7. McKenna, J.F., "Demonstration and Evaluation of a Fault-Tolerant Input/Output Network," C.S. Draper Laboratory, Inc. Report No. R-918, September 1975.

☆U.S. GOVERNMENT PRINTING OFFICE:1978-735-078/17