NASA Technical Memorandum 84546

NASA-TM-84546 19830002536

THE COST OF SOFTWARE FAULT TOLERANCE

Gerard E. Migneault

SEPTEMBER 1982

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

THE COST OF SOFTWARE FAULT TOLERANCE

Gerard E. Migneault
NASA Langley Research Center
Hampton, Virginia

SUMMARY

This paper proposes the use of software fault tolerance techniques as a means of controlling the cost of software in avionics as well as a means of addressing the issue of system unreliability due to faults in software.

Observations are first made about the problem of escalating budgets for software and about the nature of some of the causes of the increased costs, the nature of possible actions and methods proposed--and not proposed--for addressing the problem. An experiment in the measurement of software "reliability" is briefly mentioned in order to support the construction of a simple model relating the cost of a software module to the effect upon the reliability of systems containing the module.

Attention is then paid to schemes for using dissimilar redundancy in software to obtain a degree of tolerance to software faults in systems which must achieve high levels of reliability. Another simple model is developed--expressing the relationship of a "fault tolerance" scheme to system reliability. The model serves to discuss and question the customarily expected benefit, an increase in system reliability, to be obtained from fault tolerance schemes.

Finally, the simple models are combined to develop a system level view of the relationships among cost, redundancy and reliability. The view suggests the strategy, unconventional in the software world, of deliberately choosing to develop less reliable, dissimilarly redundant software modules in order to lower total software costs and increase the credibility of the estimates of their "reliability."

INTRODUCTION

The assertion that the costs related to software have become significant, even dominant, factors in budgets for the acquisition and use of digital systems is widely accepted. Consequently, more attention is being devoted to understanding and developing methods for forecasting and controlling or reducing the costs. No adequate complex of methods appears to have yet come into use, however, and the growth in total cost continues--seemingly unconstrained when compared to the decreasing costs of associated hardware.

In general, the techniques available or proposed for abating the costs of software have had two common characteristics of particular interest. They have been conventional in the sense of being variants of quite general notions which are commonly, perhaps uncritically, believed to have beneficial effects upon costs. Additionally, they have been nonspecific. That is, with the exception of the customary prerogative of management to control the level and duration of utilization of resources, the techniques have provided no means by which arbitrary, but specific, amounts of costs could be exchanged for equally specific amounts of alternative consequences. For example, the concept of a "chief programmer team" is a particular application of the notion that the structure of an organization affects the quality of its product, an extrapolation of the aphorism that the structure of a system is a determinant of its behavior. Consequently, the utility of the technique is not questioned in principle and a priori, although the relation between marginal benefits and costs is nebulous and fractional application of the technique is clearly not an option.

Software related costs have grown for a number of reasons. The most readily obvious factor is undoubtedly inflation in the general economy. While an increase in costs due to inflation does not represent a real increase in the use of resources, it does indicate that the mix of resources utilized has become less optimal. This suggests that in order to counter the effect of inflation a successful cost reduction or control scheme should implicitly, if not explicitly, address the redistribution and replacement of costlier resources with less costly. Thus schemes which are intended to provide greater visibility and control of the existing development procedures are not likely to be very successful. The concept of a "chief programmer team" appears to be in this category, as are schemes to increasingly formalize documentation requirements and change configuration control procedures.

Perhaps a more significant cause of increased costs is the simple increase in the total amount of software required as digital systems with embedded software replace older technology. Not only do the costs increase in proportion to the increased amount of software, but, as the discipline of economics teaches, in the absence of equally rapid technological progress, more and thus less efficient resources must be used. In the case of software, a people-intensive activity, this means a lowering of the average level of capability of the personnel, technical and managerial, in both the development and maintenance phases of the software life cycle. This suggests that schemes which would be successful in countering this cause of increases in cost must lessen the need for people in the software life cycle. This can be accomplished either by eliminating activities in the software life cycle or by increasing the productivity of the people involved in the activities. If such schemes require an excessive investment of capital or the introduction of costlier resources in other areas of the software activity, they may not be successful as cost savers. The introduction of "programmer workbenches" and "higher order languages" are such productivity improvement schemes. They require considerable investment of capital and continued use of the more knowledgeable personnel. Moreover, they are selective schemes in that they affect the implementation stages of program development more than testing and maintenance stages. Considered as a cost control scheme "correctness proving" would appear to be in the category of schemes which eliminate an activity. That is, if a program were absolutely correct, then there would be no need for its maintenance. Presently, however, the technique requires more expensive resources, in terms of personnel and computer time, than it releases. Also, there is currently no agreement that the technique ever will generate the desired "perfect" software.

Paper presented at the AGARD Avionics Panel Fall '82 Meeting on "Software for Avionics" held in The Hague, Netherlands, September 6-9, 1982.

Increased costs of software have also been caused by more demanding requirements for digital system performance, requirements which have been achieved by means of ever more sophisticated software. To some extent these costs are the result of the very success of digital systems in providing computational capabilities which were not previously available. The increased complexity of the software requires either more knowledgeable, and thus more expensive, personnel in the development and maintenance phases or causes an increase in maintenance activity. This suggests that to counter the effect of demands for increased performance and sophistication, schemes should be sought which reduce the complexity inherent in software. This seems to be the goal of "structured programming." However, it seems to require additional rather than less training of personnel.

The notion advanced in this paper is that in an avionics context, and possibly in other contexts in which there is an appropriately demanding requirement for reliability and maintainability, techniques of software fault tolerance utilizing redundant modules of software can be used to control costs. They do the "good" things previously cited. The level of redundancy, a parameter usually considered only for design purposes, becomes available to management as a parameter for controlling costs. But more important, from the point of view of inhibiting acceptance, the notion is unconventional--in the world of software. It is unconventional because software fault tolerance techniques have been developed to enhance reliability and are considered to be more, not less, costly and therefore less, not more, desirable. Indeed, the suggestion for this paper arose as a reaction to a statement which expressed a consensus in a "working meeting" and was unchallenged in light of its apparent logic. The statement was that, for the purpose of defining requirements, the use of a quantitative measure of the "reliability" of software should be shunned since it would necessitate the use of software fault tolerance and redundancy techniques which would, in turn, increase costs. Hence because of the less conventional nature of the technique proposed and the need for its justification, this paper appears somewhat polemic.

MODELING COST

In order to express a relation between the cost of generating software and its reliability requirement, we borrow from a recent experimental study of software reliability (Nagel, 1982). Data from the study support the assertion that

> after k faults have been corrected in a program, the probability of error during each succeeding execution of the program can be approximated by the constant $e^{-(a+bk)}$, where the parameters a and b depend upon the program and the statistical distribution of the input data and can be estimated from data obtained during a controlled process of uncovering the faults.

The terms "fault" and "error" used in the preceding statement are not synonomous. A program is understood to be simply an embodiment of an abstract relation between variables which is usually defined by a specification, implied by a requirement, etc. Thus a program, the embodiment, has structure which is not part of the abstract relation. The program can be created with a structure which, for some inputs, generates outputs which are not those implied by the abstract relation. "Fault" refers to such a flawed structure; it can be remedied, presumably when its presence is signaled by the occurrence of an error. No statement is made here about the process by which faults are generated. "Error" refers to output data which, while consistent with the structure of the software which generated them, differ from the values implied by the original abstract relation. During operation, it is the error which propogates through a system; it is the error which can be detected and can signal the presence of a fault. Whereas faults have "always" existed, errors "occur" and thus correspond to events which have rates of occurrence. An execution of a program refers to the generation of an output data set in response to an input data set. The time period of an execution is assumed to be small compared with the use period, the many executions, of a program. While it might be possible in the future to estimate the parameters a and b from descriptive information about the program at the completion of a standardized acceptance test, for the present discussion it is sufficient that they can be estimated by a controlled process of repetitive trials beginning after a standardized acceptance test.

One conclusion to be drawn from the referenced study is that a software module in a system can be considered to be a component having a constant error rate during the time it is in operation (which is assumed to be a fraction of the elapsed time of system operation). Thus, conventional notions of reliability can be discussed if errors from software modules are considered to be causes of digital system failures. Of course, not all software errors would likely result in digital system failure; what will constitute a failure will depend upon the application. Therefore, equating one with the other is a conservative assumption--which will be considered again below. With this assumption, computations of mean time to system failure due to software module error have some meaning. Conversely, a reliability budget for the various components of a digital system can assign a maximum allowable error (failure) rate to a software module.

Thus, assuming that a meaningful input data stream can be obtained or generated and assuming the successful accomplishment of the process of uncovering and removing k faults from a software module, and the estimation of the parameters a and b in the process, the error (failure) rate of a module during its subsequent operation can be expressed as

$$\lambda_k = 3600\, m\, e^{-(a+bk)}$$

where m is the number of executions per second required by the application. The expected amount of time taken during the controlled process to discover and remove the k faults, that is, to "debug" a module to a criterion λ, can be expressed as

$$MTTD_\lambda = r \sum_{i=0}^{k-1} \frac{1}{\lambda_i}$$

$$= \frac{mr}{c(1-e^b)} \left(\frac{1}{\lambda} - \frac{e^a}{3600m} \right)$$

where 1/c is the time per execution, r denotes a number of repetitions required during the "debugging" process in order to gather the data from which a and b are estimated. Note that the term "debugging" is here used for the controlled process of testing an "accepted" software module to a required λ level.

Additionally, in the assertion above there is an implied ordering of faults in term of their contribution to error rates of software modules. This means that each fault will require an increasingly long time to be uncovered. If a module is not reliable at the end of "acceptance" testing, data for estimating the parameters a and b will be relatively easily accumulated and a long "debug" phase will be forecast. If a module is relatively reliable at the end of "acceptance" testing, then it will take a correspondingly longer time to accumulate the data for estimating the parameters. In either case, ensuring that the probability of subsequent errors appearing during operation of the software module will be arbitrarily small will be lengthy activity.

On the assumption that a conventional development procedure consisting of a requirements development phase, a program design phase, a program coding phase, and a standard functional and "acceptance" testing phase can be selected, and that "debugging" to criterion λ proceeds from the point of "acceptance", figure 1 depicts the cost profile for "developing" a software module to a λ critierion.

Let $\$_0$, represented by the area under the large hump in figure 1, be the cumulative cost of developing the module through the standard "acceptance" testing point as discussed in various software cost models appearing in the literature, for example (Putnam, 1978). It might be well to note that available software cost models do not appear to be very accurate forecasters and must be calibrated for each software development environment (Thibodeau, 1981). Here, as will be seen later, it is sufficient to note that two modules developed (to the point of acceptance testing) from the same functional requirement would be expected to have similar total costs--as forecast by the available models.

Assuming the "debugging" activity to be a constant rate activity, let $\dot{\$}$ be the cost per unit time of "debugging" the software module further to its specified criterion λ. $\dot{\$}$ is assumed to include the continuing cost of generating the input data sets (stream). The cost of "debugging" is simply represented as

$$\dot{\$} \times MTTD_\lambda$$

Recalling the expression for $MTTD_\lambda$ above, we express the expected cost of a software module as a function of its λ criterion as

$$\$_\lambda = \$_o + \dot{\$} \frac{mr}{c(1-e^b)} \left(\frac{1}{\lambda} - \frac{e^a}{3600m} \right)$$

The values of $\$_0$ and $\dot{\$}$ will, of course, depend upon the complexity of the software module and the size of the staffs required to develop and "debug" it, and upon the particular software development environments. In figure 2 the ratio

$$\frac{\$_\lambda}{\$_o}$$

is plotted versus λ for various values of the parameter

$$\frac{\dot{\$}\, mr}{\$_o c(1-e^b)}$$

A glance at the figure suffices to indicate that any significant reliability requirement implies a considerable increase in development costs over software with unstated reliability. Consequently, in light of the demanding reliability requirements associated with avionics, it is unrealistic not to expect an increase in software costs if system reliability is to be achieved by extending the reliability of conventional software modules. The task is to minimize the increase.

Of course, if the λ criterion truly reflected the reliability required of the software module and if the criterion were achieved, then an occasional occurrence of a system failure due to a malfunction of the software module after operational deployment of the system would not occasion any maintenance activities. By definition the occasional failure would be acceptable. Indeed maintenance action would be suspect unless it included a repetition of the "debugging" process described. In this case, maintenance costs would be negligible, consisting principally of an accounting system to verify that the occasional errors (failures) were no more occasional than forecast. In this sense, the extent to which maintenance activities are in response to component failures is a gauge of the extent to which reliability requirements are not truly being imposed on today's systems--either by oversight or by deliberate decisions made to exchange the costs of obtaining the desired reliability for costs at a later time, the maintenance costs. Such decisions should not, of course, be within the purview of the digital system developers alone, and certainly not of the software developers, since they must (should) consider the costs resulting from the unavailability of systems--a consideration to be left to the users of the system.

FAULT TOLERANCE

The concepts of fault tolerance and redundancy are hardly new in engineering. An automobile's spare tire is a mundane witness to this fact. Nor is the idea of building more reliable systems from less reliable components by means of redundancy and passive fault tolerance new in electronics and computers (Moore, 1956). The cost benefits in terms of reduced maintenance and outages of hardware systems with internal redundancy have also been addressed (Moreira de Souza, 1981). What is novel is the notion that fault tolerance schemes can be devised to prevent system failure (or unavailability) due to design flaws (Anderson, 1981). Software faults are just such design flaws, and software fault tolerance schemes have been proposed in the past decade. The Recovery Block scheme and N-version programming are perhaps the two most widely known schemes.

Consider one "stage" of an N-version programming scheme variant as represented in figure 3. For convenience, the stage consists of an odd number, N, of dissimilar versions, P_j, of a program module which each receive input data from one of N dissimilar voter modules, V_j. Each voter module performs a majority vote on the set of inputs, Y_j, which it receives from each of the dissimilar modules of a prior stage. The outputs, X_j, of the N program modules, P_j, in turn provide inputs to the voters of one or more subsequent stages. Thus the majority value of the set of outputs, X_j, defines the output, X, of the stage. It will be correct if a majority of the X_j are correct, and erroneous otherwise. In a similar fashion, the majority value of the Y_j defines the input, Y, to the stage. In all likelihood, even if all are correct the Y_j will differ in some small amount due to the dissimilarity of the modules generating them. What this means is that the voters will contain some complex logic to account for such legitimate variations. In effect the probability of errors in the execution of these modules cannot be dismissed as insignificant.

Consistent with the previous discussion, with each program module and voter module there is associated a probability that an execution of the module will produce an erroneous output, and the probabilities can be determined, and indeed made equal, by means of the "debugging" process. Then the probability of error, q, in an execution of a program-voter pair is simply

$$q = q_p + q_v - q_p q_v$$

where q_p and q_v represent the program and voter execution error probabilities. In the context of its application, a program-voter pair will appear to have an error (failure) rate (per hour)

1) $$\lambda = 3600\, m\, q$$

The question of the independence of execution errors in "independently" developed and tested software modules is a troublesome matter. On the one hand, the study of the "reliability" of software has not progressed beyond primitive models of individual software modules. On the other hand, studies of fault tolerance and redundancy have usually been focused upon the mechanisms of the schemes, reflecting in part a less than unanimous and enthusiastic belief in the credibility of current software "reliability" assessment methodology in the computer science community. There are exceptions, of course - for example, a study of the feasibility of the application of the recovery block scheme in an avionics application (Hecht, 1978). We shall return to the question later, but here assume that errors in module-voter pairs occur independently of errors in other module-voter pairs. With that assumption, the probability of an erroneous stage execution output can be expressed as

2) $$P_{bx} = (1-q)^N \sum_{i=\frac{N+1}{2}}^{N} \binom{N}{i} \left(\frac{q}{1-q}\right)^i$$

and a relationship between the stage error (failure) rate

3) $$\lambda_s = 3600\, m\, P_{bx}$$

and the component program-voter error rate, λ, established. The relation is plotted in figure 4 for various values of the parameters m and N. Because of the questionable assumption of independence of errors, the plotted curves represent bounds on what is achievable.

Two "fault-tolerant" computer systems, SIFT and FTMP, have been developed under NASA contract and reported in the literature (Goldberg, 1981)(Hopkins, 1978). The design goal of the systems was to achieve, at some reasonable cost, systems of very high reliability for avionics applications. Neither of the systems utilized software fault tolerance schemes. However, as can be seen from figure 5, the architecture of the SIFT computer lends itself admirably to the N-version scheme described above, and can be used to describe how the scheme would actually be implanted into hardware. Simply, each program-voter module pair would reside in a separate processor. With more processors available than required by an N-version stage, in the presence of a hardware failure, the hardware reconfiguration algorithm of the SIFT computer can assign another processor to be in the stage, thus maintaining the stage redundancy at a constant N.

COST VERSUS REDUNDANCY

The relations developed above may be combined to express the cost of software fault tolerance in terms of its level of redundancy, the reliability which it is intended to provide (actually the error (failure) rate which it is not to exceed) and the cost parameters. Recalling the expression for the expected cost, $\$_\lambda$, of a software module as a function of its lambda criterion, we represent the cost of an N-version stage as

$$2\, N\, \$_\lambda$$

reasoning that the N program modules will have the same expected cost as a result of having been "developed" to the same requirement. This is not inconsistent with the accuracy of the current software cost estimation models, as was noted previously. The factor 2 is included to account, hopefully conservatively, for the N versions of the voter modules which, as was noted, will contain some amount of complexity. This cost is compared to the cost of a single module having the error (failure) rate, λ_s, the error (failure) rate desired of the stage, by forming the ratio

$$\frac{2\ N\ \$_\lambda}{\$_{\lambda_s}}$$

in which λ_s and λ are related by the equations 1), 2), 3). In the special case N=1, there are no voter modules and the ratio is unity. In figure 6 this ratio is plotted against λ_s for various values of N, and the conglomerate parameter

$$\frac{\$\ m\ r}{\$_o c(1-e^b)}$$

Note that each point along one curve, determined by a fixed set of the parameters above, corresponds to a different value of λ --satisfying the relations 1), 2), 3). Hence, given values for λ_s, m and the conglomerate parameter, corresponding to an application requirement and development environment, one can determine the values of λ and N which minimize the ratio.

Note also that in some cases the optimal policy is to use surprisingly unreliable program-voter pairs. When such is the case, several changes in policy suggest themselves. First, it becomes feasible to accumulate more error (failure) history data on modules than is customary, thereby providing greater confidence in the estimates of their (un)reliability. Secondly, it becomes economically feasible to subject the software modules to real-use testing since observable results will occur quickly, thus providing a better understanding of the relation of module errors to system failures and a rationale for relaxing the conservative assumption equating errors to failures, if appropriate, and addressing criticisms about the incompleteness of testing based on solely simulated input data streams.

While the problem of independence of errors remains, it is tempered by the almost certain knowledge that only a fraction of a module's errors will be correlated with those of other modules. How significant or insignificant the fraction is is an appropriate subject for study and experiment in light of the cost benefits available from software fault tolerance. It is further tempered by the additional, almost certain knowledge that only a fraction of software execution errors will cause system failure. Again, study and experimentation is warranted--especially in light of the propensity of humans to believe that they know more than they do when dealing with subjective probabilities (Lichtenstein, 1981).

CONCLUSION

The thesis of this paper, simply put, is that decisions about the use of redundancy in software fault tolerance should be made with the understanding that they provide a cost minimization as well as reliability enhancement potential, and the rudiments of a technique have been presented.

REFERENCES

Anderson, Thomas , and Lee, Peter A., 1981, Fault Tolerance, Principles and Practices, Prentice-Hall International, London.

Goldberg, Jack, 17-19 November 1981, "The SIFT Computer and its Development" in 4th AIAA/IEEE Digital Avionics Systems Conference: Collection of Technical Papers, AIAA, New York, pp.285-289.

Hecht, Herbert, February 1978, Fault-Tolerant Software Study, NASA Contractor Report #145298, The Aerospace Corporation, Los Angeles, California.

Hopkins, A. L., Smith, T. B., and Lala, J. H., October 1978, "FTMP - A Highly Reliable Fault Tolerant Multiprocessor for Aircraft" in Proceedings of the IEEE, Vol. 66, No. 10, pp. 1221-1239.

Lichtenstien, Sarah; Fischoff, Baruch; and Phillips, Lawrence D., June 1981, Calibration of Probabilities: The State of the Art to 1980, Contractor Report PTR-1092-81-6 prepared for the Office of Naval Research Contract #N00014-80-C-0150, Decision Research Division of Perceptronics, Inc., Eugene, Oregon.

Moore, E. F., and Shannon, C. E., 1956, "Reliable Circuits Using Less Reliable Relays" in Journal of the Franklin Institute, Vol. 262, Part I, pp. 191-208, Part II, pp. 281-297.

Moreira de Souza, J. and Landrault, C., April 1981, "Benefit Analysis of Concurrent Redundancy Techniques" in IEEE Transaction on Reliability, Vol. R-30, No. 1, pp. 67-70.

Nagel, Phyllis M., and Scrivan, James A., February 1982, Repetitive Run Experimentation and Modeling, NASA Contractor Report #165836, Boeing Computer Services Company, Seattle, Washington.

Putnam, Lawrence H., July 1978, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem" in IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, pp. 345-361.

Thibodeau, Robert, April 1981, An Evaluation of Software Cost Estimating Models, Contractor Report#1-940 prepared for RADC Contract F306-79-C-0244, General Research Corporation, Huntsville, Alabama.

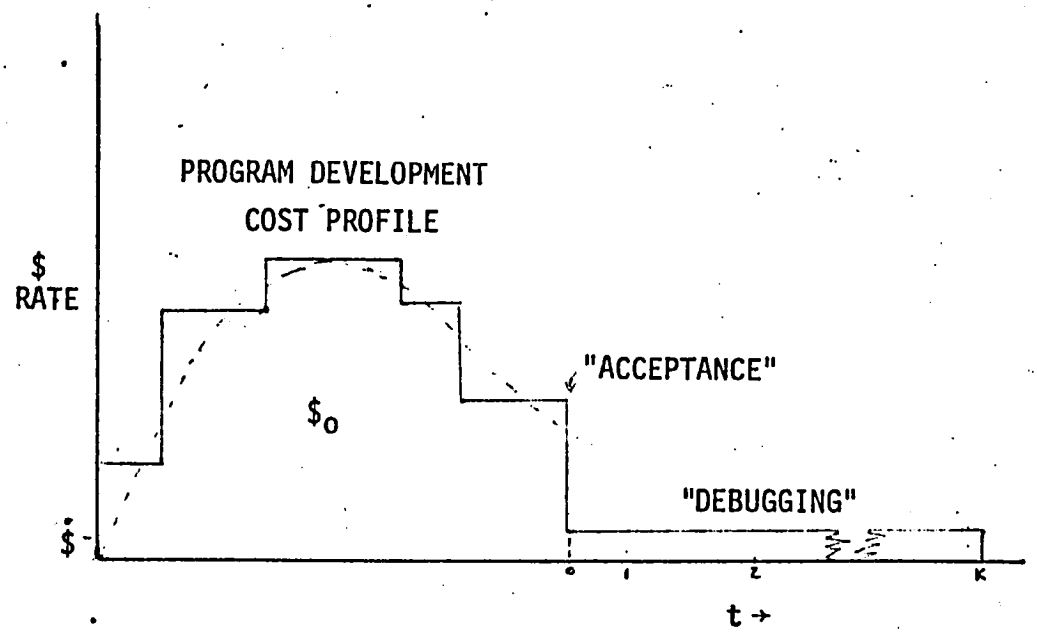

PROGRAM DEVELOPMENT
COST PROFILE
$
RATE
$o
"ACCEPTANCE"
"DEBUGGING"
$
0
1
2
K
t →

FIGURE 1

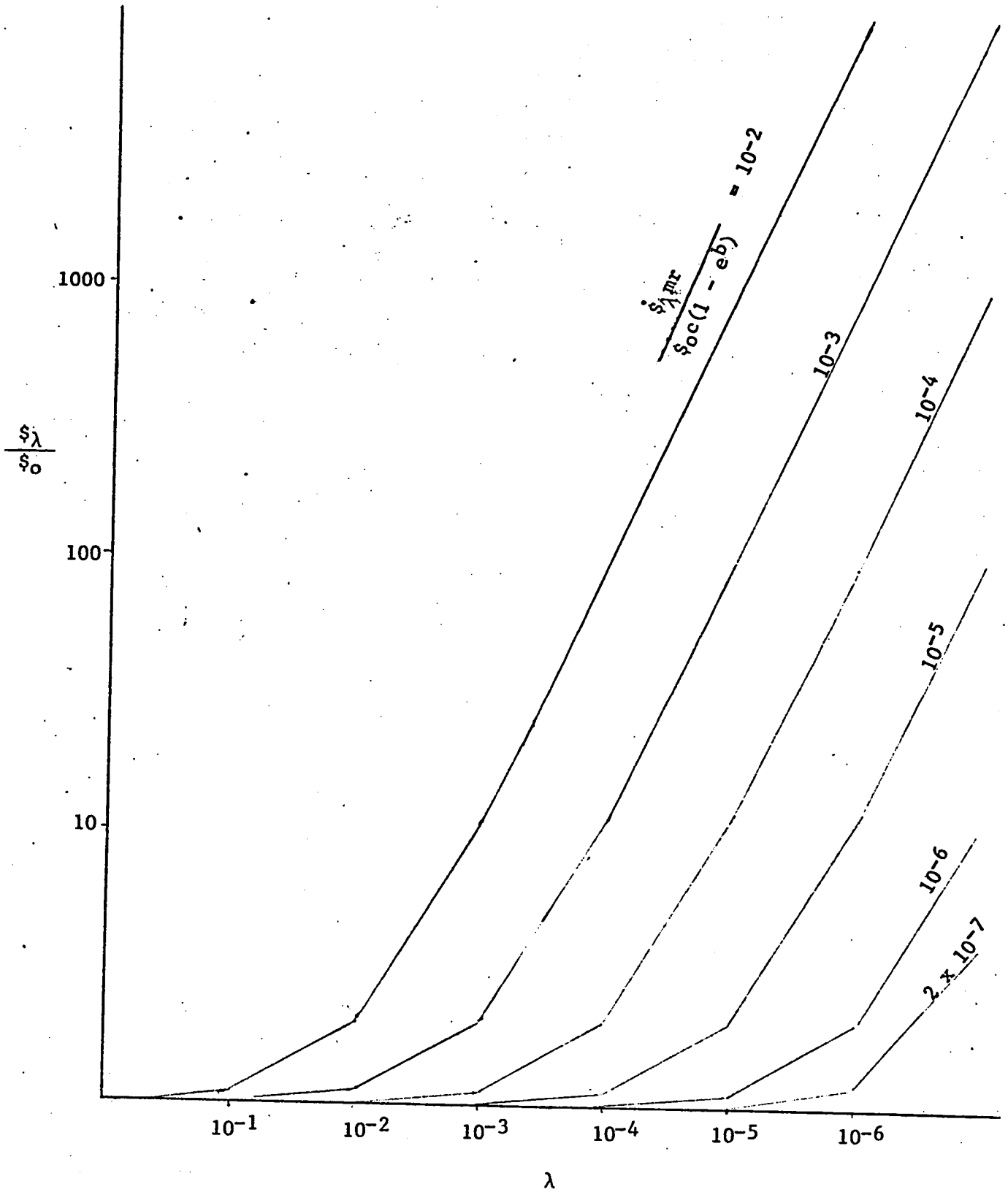

1000
100
10
$\frac{\$_\lambda}{\$_o}$
$\frac{\$_{\lambda mr}}{\$_{oc}(1 - eb)}$ = 10-2
10-3
10-4
10-5
10-6
2 x 10-7
10-1
10-2
10-3
10-4
10-5
10-6
λ

FIGURE 2

N-VERSION PROGRAMMING

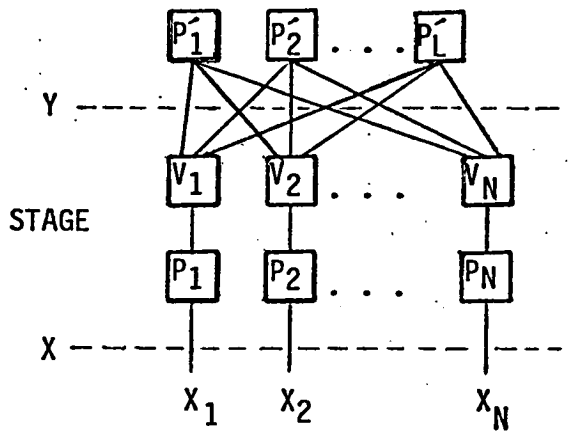


FIGURE 3

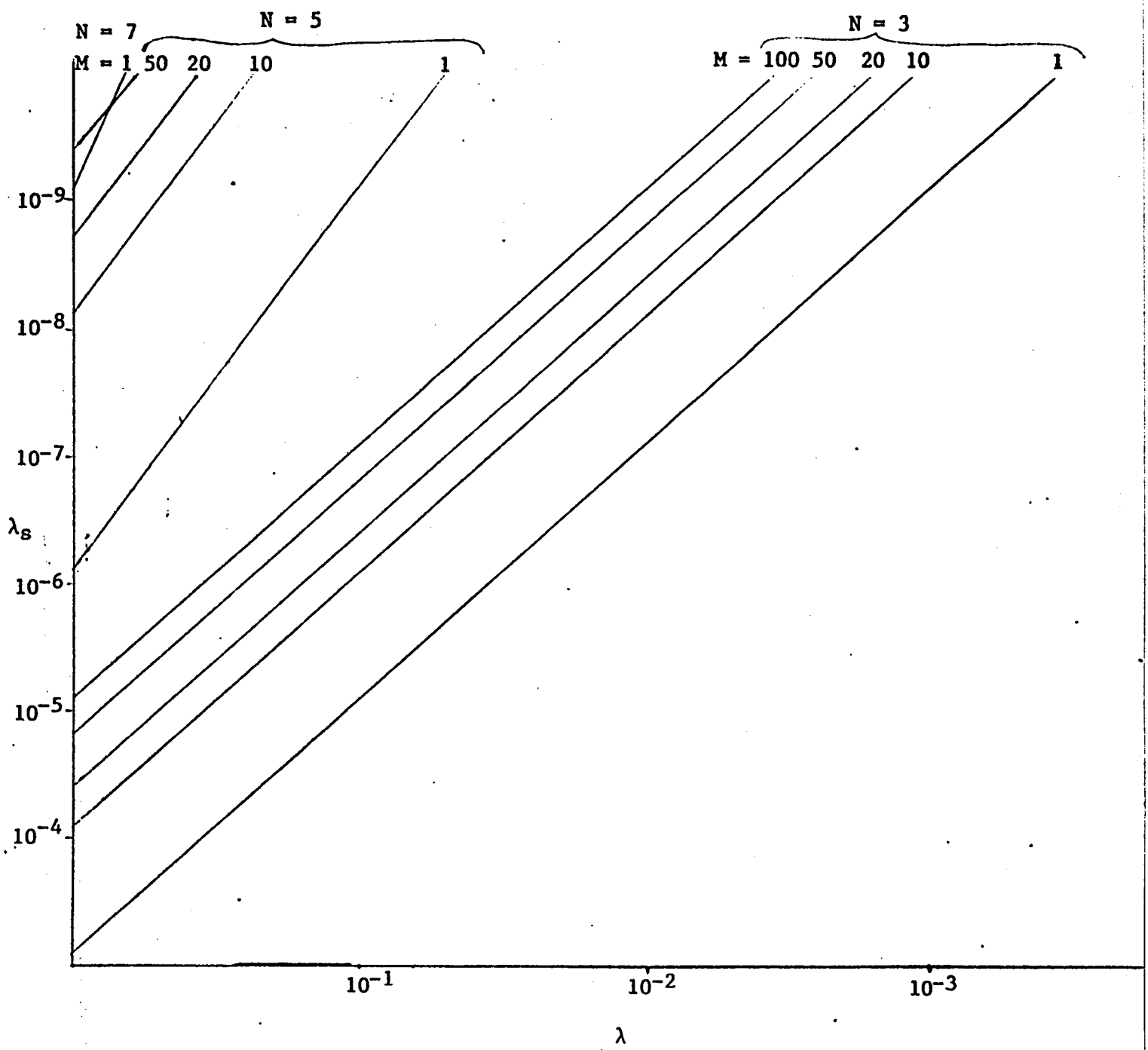
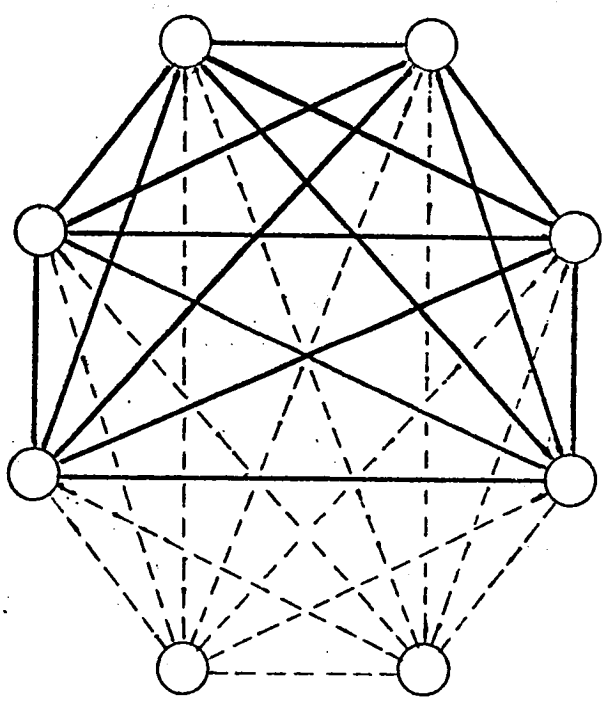

N = 7
N = 5
N = 3
M = 1 50 20 10 1
M = 100 50 20 10 1
10^{-9}
10^{-8}
10^{-7}
λ_s
10^{-6}
10^{-5}
10^{-4}
10^{-1}
10^{-2}
10^{-3}
λ

FIGURE 4



6 PROCESSOR SIFT CONFIGURATION
(EXPANDABLE TO 8)

FIGURE 5

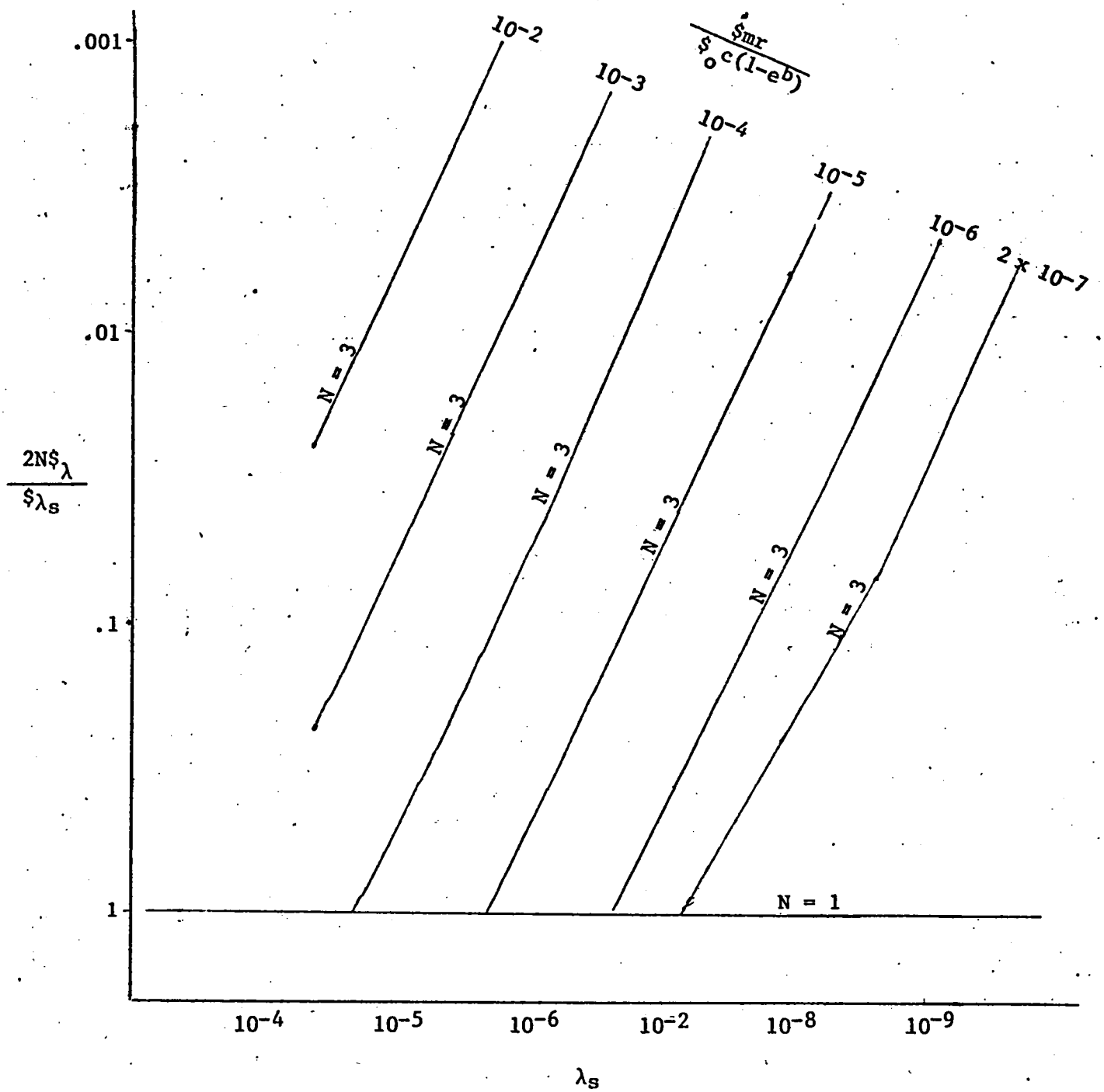

.001
.01
.1
1
$\frac{2N\$_\lambda}{\$\lambda_s}$
$\frac{\$_{mr}}{\$_o c(1-e^b)}$
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
2×10^{-7}
N = 3
N = 1
10^{-4}
10^{-5}
10^{-6}
10^{-2}
10^{-8}
10^{-9}
λ_s

FIGURE 6

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-84546		

4. Title and Subtitle	5. Report Date
The Cost of Software Fault Tolerance	September 1982
	6. Performing Organization Code: 505-34-43-08

7. Author(s)	8. Performing Organization Report No.
Gerard E. Migneault	
	10. Work Unit No.

9. Performing Organization Name and Address	11. Contract or Grant No.
NASA Langley Research Center Hampton, VA 23665	
	13. Type of Report and Period Covered: Technical Memorandum

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Washington, DC 20546	

15. Supplementary Notes

Paper presented at the AGARD Avionics Panel Fall '82 Meeting on "Software for Avionics" held in The Hague, Netherlands, September 6-9, 1982.

16. Abstract

This report examines the proposed use of software fault tolerance techniques as a means of reducing software costs in avionics as well as addressing the issue of system unreliability due to faults in software. A model is developed to provide a view of the relationships among cost, redundancy, and reliability which suggests strategies for software development and maintenance which are not conventional.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Fault tolerance Software cost Software reliability	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	12	A02

For sale by the National Technical Information Service, Springfield, Virginia 22161