NASA Technical Paper 1067

Fault-Tolerant Computing: A Preamble for Assuring Viability of Large Computer Systems

Raymond S. Lim

OCTOBER 1977

NASA

NASA Technical Paper 1067

Fault-Tolerant Computing: A Preamble for Assuring Viability of Large Computer Systems

Raymond S. Lim
Ames Research Center
Moffett Field, California

National Aeronautics
and Space Administration

Scientific and Technical Information Office

1977

FAULT-TOLERANT COMPUTING: A PREAMBLE FOR ASSURING VIABILITY OF LARGE COMPUTER SYSTEMS

Raymond S. Lim

Ames Research Center

SUMMARY

This paper describes why large computer systems need to be more fault-forgiving, or fault-tolerant, in order to be viable. The need for fault-tolerant computing is addressed from the viewpoints of (1) why it is needed, (2) how to apply it in the current state of technology, and (3) what it means in the context of the Phoenix computer system and other related systems. To this end, the value of concurrent error detection and correction is described from the viewpoints of (1) user protection, (2) program retry, and (3) repair. The technology of algebraic codes to protect memory systems and arithmetic codes to protect arithmetic operations is discussed.

PROLOGUE

Historically, at any point in time, the sizes of contemporary computers have been classified as small, medium, or large primarily on the basis of price and performance. At present, operational computers such as ILLIAC-IV (I4), Texas Instruments' Advanced Scientific Computer (ASC), Control Data Corporation's STAR-100, and Cray Research's CRAY-1 can be considered as large computers. For those organizations that are charged with the operational responsibility of these large computers, the increase in awareness of system reliability during the last few years has been phenomenal. At the Institute for Advanced Computation (IAC), Ames Research Center, one could claim that this has been prompted by the ILLIAC-IV experience. There are, however, several other considerations which prompted the current level of awareness about enhancing system reliability by incorporating fault-tolerant computing. These came from private industries, government agencies, professional societies, and academic research communities.

The concept of fault-tolerant computing is a technique for designing digital computer systems that can function properly despite the presence of faulty hardware components. In present technology, the principal technique used is to introduce redundancy into the system for the concurrent detection and correction of errors and the automatic hardware replacement of faulty functional modules. The principal interest in redundancy techniques arises from the lack of perfect components and perfect system fabrication methods. Even in Biblical times, when David carefully selected five smooth stones out of the brook in preparation for his battle with Goliath (ref. 1), the value of equipment redundancy was qualitatively recognized.

Nearly 25 years ago, in January 1952, on the California Institute of Technology campus, the first comprehensive statement of the challenge of fault-tolerant computing and the value of component redundancy was delivered by John von Neumann during five lectures (ref. 2). In the 1950's redundancy was used in the SAGE air defense computer system (refs. 3 and 4). This was a large vacuum-tube system with 30 duplex computers. Each computer had about 60 000 tubes.

In the 1960's, redundancy was used both in commercial computers and aerospace computers. To name a few, redundancy was used in the (OAO) data processor (ref. 5), the Bell Laboratories Electronic Signal Switching (ESS) computer (refs. 6 and 7), the SATURN V guidance computer (ref. 8), the IBM System/360 computers (refs. 9, 10, and 11), the JPL-STAR computer (ref. 12), and the MIT Multics System (ref. 13).

By the 1970's, redundancy found wide acceptance. It was used in the IBM System/370 computers (ref. 14), the Amdahl 470V/6 (ref. 15), the Raytheon aerospace computer called (SERF) (refs. 16 and 17), the Hughes (ARMMS) Computer (ref. 18), the generic design from IBM Research (ref. 19), and the Modular Spacecraft Computer MSC (ref. 20).

In large computers such as the ILLIAC-IV, the TI-ASC, and the CRAY-1, where fault-tolerant computing is sorely needed to enhance system reliability, it is surprising to find an almost complete absence of it. In the case of the TI-ASC, it is reported that a Hamming Code is used in the memory for single-bit error correction (ref. 21). In the CRAY computers, it is informally reported that, in subsequent models, a Hamming Code will be used for memory error correction.

Government agencies and professional societies have also provided impetus for the awareness of fault-tolerant computing. For example, the Office of Naval Research and Westinghouse Electric Corp. joined forces in sponsoring a Symposium of Redundancy Techniques for Computing Systems, held in Washington, D.C. on February 6 and 7, 1962 (ref. 22). In 1971, the IEEE Computer Society and Jet Propulsion Laboratory of the California Institute of Technology joined forces in sponsoring the first symposium on fault-tolerant computing, which was chaired by the very able A. Avizienis. Subsequently, this symposium was held annually.

It is in the areas of concurrent detection and correction of errors, redundancy techniques for replacement, and program checkpoints and retry that academic and other research communities have made advances in fault-tolerant computing. In this paper, these advances are articulated by way of examples and available new methods; however, no attempt is made to present any theory. The lack of a theoretical treatment is largely owing to the pioneering stage of fault-tolerant computing, and is somewhat application oriented.

INTRODUCTION

The problem of reliability, availability, and serviceability (RAS) is of continuing interest to both designers and users of computer systems since the building of the first computer in the 1940's. As presented in the Prologue, fault-tolerant computing has applied principally to three classes of computer systems: Aerospace Computers, Military Computers, and large ground-based computers. In this paper, the discussion is limited to large ground-based computers only. The issue of software reliability will not be discussed.

No matter how carefully constructed, computer systems fail. The larger the system, the higher the probability of failure. This is primarily caused by the absence of perfect components, the absence of perfect manufacturing methods, and human errors. Thus, computation without error remains an illusive goal. These points were well appreciated by system reliability theorists and were well proven in large computer systems, principally the ILLIAC-IV. No attempt is made here to report on the system reliability of the ILLIAC-IV. A comprehensive ILLIAC-IV history and system development can be found in an article by Falk (ref. 23).

According to system reliability theorist Professor D. Siljak of the University of Santa Clara, "We all know that, when a system becomes too complex, and it has too many interdependent parts, it eventually will reach the point of collapse" (ref. 24). The situation for large computer system reliability is not all that hopeless. Although complete protection against component failures may not be possible in practice, a limited degree of protection can be provided by designing the system for fault-tolerant computing. If this limited degree of protection is implemented properly, then, in most cases, it is possible to reduce the system unreliability to an acceptably low level.

The basic approach to fault-tolerant computing is to introduce protective redundancy into the design of the system early in the project phase of system architectural planning. The causes of system unreliability are expected to be present and to induce failures during a computing process, but their disrupting effects are automatically counteracted by the redundancy. The resources allocated to the increase in reliability are spent on protective redundancy. Minimizing the effects of faults involves: (1) the detection (or correction) of error, (2) the replacement of faulty modules (perhaps automatically), and (3) the restart of the program from the previous checkpoint.

Although, to date, the principle of fault-tolerant computing is not widely used in commercial systems, there are a number of developmental efforts in this direction. One principal development in this direction is the Amdahl 470V/6 Computer. The general lack of usage is principally owing to the intended application of most commercial systems, which can usually tolerate a system failure if the Mean Time Between Failures (MTBF) is within acceptable limits. In applications that cannot tolerate a system failure, like the banking and security institutions, a dual-system is generally used.

A failure of one system, at most, would degrade the system performance, but not cause a complete system failure. Thus, the prevailing attitude tends to be that fault-tolerant computing is a preamble for assuring the viability of large computer systems.

It is felt that the basic knowledge pertaining to fault-tolerant computing is now understood, at least from the standpoint of being able to make sound engineering judgements concerning the utilization of various techniques. How to provide it is a question being addressed by many designers and scholars (refs. 25 through 28). The suggested approaches differ considerably, to some extent, because there are as yet no agreed-upon figures of merit for the required system, and no single date by which they must be in the field.

The purpose of this report is to address three areas of fault-tolerant computing relevant to large computer systems development at IAC. Basically, the areas addressed are (1) why it is needed, (2) how to apply it, and (3) its application to the Phoenix System (ref. 29). The Phoenix System currently is an active IAC research project to develop a computer system to succeed ILLIAC-IV. No unified figure of merit or solidification is attempted in this report. Rather, it is a representation of the consensus of opinion of one group of designers, with experience in large computer system operation, concerning the most feasible approach to fault-tolerant computing for large computer systems in the 1976 to 1982 time frame.

The author gratefully acknowledges the influence and continuous encouragement received from his colleagues G. F. Feierbach and D. K. Stevenson. He also wishes to thank his colleagues and J. Korpi for reading and commenting on the work reported herein.

FAULT-TOLERANT COMPUTING FOR LARGE SYSTEM VIABILITY

In this section, the application of fault-tolerant computing is described from the viewpoints of (1) increased system reliability and availability, and (2) system integrity. System reliability and availability are described from the viewpoints of component reliability and the improvement of reliability if error corrections were provided. System integrity is described from the viewpoints of (1) error detection for user protection, (2) error detection for program retry, and (3) error detection for repair.

Reliability and Availability

The primary motivation for fault-tolerant computing in large computer systems is to increase the reliability to a degree that will allow the system to be used for solving large-scale problems. Large-scale problems that require vast amounts of computing power are the simulation of three-dimensional aerodynamic flow equations, simulation of climate models and

weather predictions, seismic modeling, and various applications involving linear programming. Even on large-scale computers, the computation time for these large-scale problems can be long — in some cases, 12 h or longer. This implies that the computer system must be operating correctly during the entire computational period. Here, "operating correctly" means the correct execution of a program, rather than the continued correct functioning of all the components of the system. In order to achieve such a level of reliability, the large computer system must be designed to be more fault forgiving. That is, the system must be able to tolerate a few errors and still function correctly. This leads to the concept of fault-tolerant computing. As described earlier, the basic approach to fault-tolerant computing is to introduce protective redundancy into the design of the system for concurrent error detection and correction. If the error is uncorrectable, the system should automatically reconfigure itself to replace the bad components and continue with the computation.

For small computer systems, the deployment of full-scale fault-tolerant computing may not be economically justified. For these systems, error detection in the form of word parity is generally regarded as sufficient. As an example, consider a memory system using a 1024-bit Random Access Memory Integrated Circuit (RAM IC). Off-the-shelf IC's generally have a failure rate of 0.1% per 1000 h. If the IC's are properly screened and burned-in, a failure rate of 0.01% per 1000 h or better probably can be obtained with currently available technology. With a device failure rate of 0.1% per 1000 h, a small memory system consisting of 4096 16-bit words using 64 IC's would have a mean time between failure (MTBF) of

$$\text{MTBF} = \frac{1}{d\lambda} = \frac{1}{64 \times \frac{0.001}{1000\ \text{h}}} = 15{,}600\ \text{h}$$

where

d = total number of IC's used

λ = IC failure rate

However, a large memory system consisting of 1.024×10^6 64-bit words (8 megabytes) using 65,536 of these IC's would have an MTBF of

$$\text{MTBF} = \frac{1}{65{,}536\ \frac{0.001}{1000\ \text{h}}} \cong 15\ \text{h}$$

A memory size of 8 megabytes is very common in the current technology. For example, the commercial computer Amdahl 470V/6 has this size memory. In the case of the ILLIAC IV (as installed), the memory size is 256K 64-bit words. Assuming that the IC's have the same λ, the MTBF would be 64 h.

Thus, for large computer systems, error correction is not only very attractive, but necessary.

The theory and methods of implementing error correction codes (ECC's) are well known (refs. 30 through 34). Also, the mathematics of reliability are now understood (refs. 35 and 36). If single-bit error correction is applied to the 1.024×10^6 64-bit words memory system, the reliability can be improved by 27 times (ref. 37). This increases the MTBF from 15 to 405 h. For this MTBF, and if the mean time to repair (MTTR) is less than 405 h to prevent the occurrence of a double-error, then the practical attainment of a fault-tolerant memory system is fulfilled, at least from the viewpoint of IC failures.

Although they are understood, the mathematics of reliability are complex subjects, and therefore no attempt is made here to dwell on such details. As an overview, the relationship between MTBF, MTTR, and the number of errors N, and, for small N, can be shown to be

$$\frac{1}{MTBF_e} \cong \frac{MTTR^{N-1}}{MTBF^N} + \frac{1}{MTBF_{ecc}}$$

where

MTBF = intrinsic MTBF of system calculated from component reliability.

$MTBF_e$ = effective MTBF of system with MTTR taken into consideration.

$MTBF_{ecc}$ = MTBF of error correction circuit.

For N = 1, the above equation shows $MTBF_e \cong MTBF$. This is the correct result because MTTR has not been taken into consideration. For N = 2, and ignoring the $MTBF_{ecc}$ term, the above equation shows

$$MTBF_e \cong MTBF\ (MTBF/MTTR)$$

If MTTR < MTBF, then the effective MTBF is improved by the factor (MTBF/MTTR). Other examples of computer system reliability with and without periodic maintenance can be found in reference 38.

The relationship between availability, MTBF, and MTTR is defined by

$$\text{Availability} = \frac{\text{uptime}}{\text{uptime + down time}} = \frac{1}{1 + (MTTR/MTBF)}$$

Again, if MTTR << MTBF, availability approaches unity.

In summary, system reliability and availability are directly affected by component failure rate, system MTBF, system MTTR, and system error correcting capability. At any given time, the component failure rate is fixed by the technology at that particular time. Any improvement on system reliability

and availability must come from improving system MTTR and providing some system error correcting capability. System MTTR is an illusive matter. It depends on how easily the system can be maintained, the resources of the maintenance staff, and the like. On the other hand, system error correction is real and very effective. As described later (in the section on current technology), a single-error correcting (SEC) Hamming Code requires only 7 parity-check bits to protect a 64-bit word memory system. If a SEC code is provided, then the system can fail only if two errors occur. If the single-error failure can be repaired before a double-error failure occurs, then the system MTBF is equal to $MTBF_e$ for N = 2. This effective MTBF is improved by the factor (MTBF/MTTR).

Although only memory system MTBF improvements are discussed in this section, the same type of discussion can also be applied to arithmetic units. Techniques for detecting errors in arithmetic operations are described in the section on current technology.

Protecting System Integrity

The secondary motivation for fault-tolerant computing is to provide the large computer system with the following attributes for protecting system integrity:

(1) Error detection for user protection
(2) Error detection for program retry
(3) Error detection for repair

These attributes are briefly described herein.

Error detection for user protection— Computer users, whether they are scientists, engineers, architects, or businessmen, use computers to perform complex calculations. Based upon the answers received from a computer, some important decisions are often made. If a computer is fault-intolerant, if it is incapable of signaling whether errors occurred during computation, then it is certainly not desirable to rely on results from such a computer. This is particularly true when large sums of money or other important factors depend on the decision. These other factors may occur when human safety is involved, such as in the design of a high-rise building, a bridge, or an aircraft. On the other hand, if a computer is fault-tolerant, automatic detection of errors provides protection to the user by: (1) signaling when an error has been detected, informing the user that the results or the program being run are probably wrong, and (2) by telling the user when the system is no longer operating correctly.

The automatic detection and correction of errors are not obtained without cost in performance and equipment, but, in large systems, the cost is justifiable and is small when compared to the overall system. The added equipment can also cause errors since it also has a certain MTBF. The issue here is not what causes the error, but, rather, the detection of errors. If

errors occur, regardless of the source, these errors should not be allowed to propagate through the system without being detected.

Within the context of error detection for user protection, it is good practice for the user to write the program to protect against programming errors and operator errors that may not be machine detectable. However, if there is a high degree of hardware error detection, the operational program need detect only a minimal class of errors, if any at all. This simplifies the programming, and perhaps decreases program running time.

Error detection for program retry— In a large computer system, a substantial percent of the detected errors are caused by intermittent errors. Intermittent errors are those errors that do not occur every time an operation is attempted. From the viewpoint of the user, an intermittent error could result in a stoppage of his program. The user must correct any resulting errors in his data and restart the computation. Such interruptions are not only inconvenient, but costly as well.

It is possible to design the system with program checkpoints and automatic program rollback to restart the computation interrupted by an intermittent error. Such an automatic recovery procedure is called a program retry. The program checkpoints can be implemented by hardware, software, or both. In a program checkpoint, it is necessary to save information about program status and any actions taken by the program for input and output of data.

Error detection for repair— In a large computer system, the cost of providing maintenance is very high. The task of designing a computer is a one-time effort, but the task of maintenance is a continuing effort throughout the lifetime of the system. Most of the cost is in preparation for eventual machine malfunctions rather than in the actual repair cost. These preparations include writing diagnostic software, writing repair documentation, training, and deployment of trained personnel for unscheduled maintenance.

Another factor in the high cost of maintenance is the difficulty in diagnosing trouble symptoms. In a large system without any effective error detection, the maintenance cost not only is very high, but the situation sometimes seems very hopeless. It should be pointed out that the cause of intermittent errors is very hard to diagnose without error detection. This is because the evidence of intermittent error is quite vague for a machine with no error detection. If the status of the machine is known immediately after the occurrence of the intermittent error, it is often possible to pinpoint the failing hardware to within a few replaceable components.

If error detection hardware is designed into the system to aid maintenance, the cost of repairing system malfunctions should decrease. For a memory system, the detected error can be automatically corrected and logged. This kind of error is transparent to the user and, hence, causes no program stoppage. The logged errors can be repaired during scheduled maintenance time. On the other hand, if the memory system has no error correction, any error will, at best, cause user program stoppage and also a

call for unscheduled maintenance. A computer with such a fault-intolerant memory may not be justified for certain applications.

For an arithmetic unit, the unit can also be designed to be fault-tolerant by detecting and correcting errors in arithmetic operations. The discussion on this topic is presented in the section on current technology.

Summary

In this section, reliability and availability were described in terms of IC failure rates, system MTBF, and system MTTR. System operation integrity and trustworthiness were described in terms of concurrent detection and correction of errors and program retry. The improvement of system MTBF and system integrity by fault-tolerant computing was described. The key issue here is that the system must be made trustworthy; that is, there must be enough confidence in the system that good answers can be reliably recognized as good answers.

CURRENT TECHNOLOGY

At the present time, the technology of fault-tolerant computing is somewhat in its infancy. The basic requirement of fault-tolerant computing, and perhaps the most difficult one, is the ability to detect and correct errors. The theory of algebraic codes, generated from the algebra of polynomials over finite fields, can be used to protect against errors in data transmission and in memory systems. The theory of arithmetic codes, based on the algebra of residue numbers, can be used to protect against errors in arithmetic operations. The theory of algebraic codes is now relatively well understood, at least from the viewpoint of being able to make sound engineering judgments concerning the application of various techniques (refs. 30 through 34 and 39 through 45). What is needed is more research and development directed toward practical applications. It should be pointed out, however, that error correction should be applied to compensate for unexpected imperfections, but not to compensate for an immature hardware technology. For such technology, it is probable that no error codes, no matter how powerful, can make the system viable.

Errors in Memory Systems

In random-access memory (RAM) systems, the primary sources of errors are defective transistors in a storage cell, open circuits, short circuits, and timing errors. These errors are called hard errors, and they can occur randomly and independently. Backplane wiring noises, cross-coupling signals, and other accidental transients can also occur randomly and independently. These errors are called soft errors. Because of the random and independent nature of the errors, systems can be designed so that memory failures are usually single-bit errors.

In recording storage media, such as magnetic tapes and magnetic disks or drums, surface defects include loss of oxide, scratches, dirt particles, and wrinkles. The effect of such disturbances can accumulate until the data are no longer readable. These defects typically assume sizes up to 0.00254 cm, and, since data usually are recorded serially on these media, the results are short burst of errors.

Correcting Errors in Memory and Data Transmission

The cyclic code, generated by an ideal in the algebra of polynomials, can be used for error correction in computer memory systems and also in data transmission. The cyclic code is chosen because of its ease in implementation. It can be implemented to correct errors in RAM systems where data transmission is in parallel, and where errors occur randomly and independently. The Hamming code is an example; today it is used in practically all large RAM systems that use any code at all.

The cyclic code can also be implemented to correct errors in rotating memory systems where data transmission is serial, and errors occur in bursts. The Fire Code is an example; it is used principally in IBM, or IBM compatible, equipment (ref. 44). Because of the mathematical complexity in the high-speed decoding of this code, the industry, in general, does not have the resources to understand it fully. For this reason, non-IBM-compatible equipment usually does not offer error correction.

The basic Fire Code was used as an inner code for the UNICON 690 laser memory. It is an 80-bit code, of which 64 bits are data. It is generated by

$$g(x) = (x^{10} + 1)(x^6 + x + 1)$$

The error correcting capability is a single-burst of length $b \leq 6$. Also, an outer Fire Code with high-speed decoding was proposed for use in the UNICON 690 (ref. 45).

Another code, the Reed-Solomon Code, is also a cyclic code suitable for correcting errors in RAM systems, as well as in disk memory systems. The Interlaced Code is yet another interesting code. As described later, it decomposes a burst error into random error.

Hamming Codes from GF(2)— The Hamming Codes with code symbols from a Galois field of two elements GF(2) are a subclass of the BCH codes. BCH codes are cyclic codes discovered by Bose and Chaudhuri in 1960 and, independently, by Hocquenghem in 1959. The Hamming Codes are single-error correcting codes generated by a primitive polynomial g(x) of degree m. This code has the following parameters:

Code length (bits): $n = 2^m - 1$

Number of parity-check bits: $n - k = m$

Number of information bits: $k = 2^m - m - 1$

Error-correcting capability: $t = 1$

Minimum distance: $d = 2t + 1 = 3$

To protect this code for decoder failure during the occurrence of a double-error, an overall parity check in the form of $(x + 1)$ can be appended to $g(x)$. This increases the minimum distance of code from 3 to 4. For example, in order to protect a 64-bit word length RAM system, the $(n,k) = (127,120)$ code can be used. This (127,120) code has $n - k = 7$, and it can be shortened to a (71,64) code. If double-error protection is required, the total redundancy is 8 bits, or 12.5%. The implementation of this code is easy. It can be implemented by table look-up using read-only memories (ROM), or it can be implemented by using XOR gates. Either way, the complexity is at most 35 IC's. Using currently available high-speed logic, the decoding time for the (71,64) code is 10 ns or less. Thus, for a large memory system, the decoder is not an issue. What is an issue is the 12.5% redundancy. If the memory system is destined to be used in a large computer system, the 12.5% redundancy is well justified for that peace of mind.

Fire Codes from GF(2)— The Fire Codes, discovered by Philip Fire in 1959, are a class of cyclic codes that can be constructed systematically for correcting a single burst of error in a codeword of n bits. The codes are rather efficient, and they can be implemented with feedback shift registers. In a manner similar to the Hamming Code, the implementation of the Fire Code is easy, and therefore it is not an issue.

An error burst of length b is defined as a vector whose incorrect components are confined to b consecutive bit positions, the first and last of which are incorrect. It is clear that for a given k and b, the objective is to construct an (n,k) code with as small a redundancy $n - k$ as possible. From coding theory, it is known from the Reiger bound that the number of parity check bits of a b-burst-error correcting code is

$$n - k \geq 2b$$

This is an upper bound, and codes that meet the Reiger bound are said to be optimal. The ratio

$$Z = (2b)/(n - k)$$

is a measure of the burst correcting efficiency of the code. An optimal code has $Z = 1$. The Fire Code at best has

$$Z = \frac{2b}{3b - 1} \cong \frac{2}{3}$$

which is not very optimal with respect to the Reiger bound. Despite this suboptimal condition, the Fire Code is an efficient code when it is used to protect a long record of data. For example, in the IBM 3830 disk controller, 56 bits are used to protect a record of data with lengths up to about 100 000 bits.

A Fire Code with code symbols from GF(2) that can correct any burst of length b or less and simultaneously detect any burst of length d > b is best described by its generator polynomial

$$g(x) = p(x)\ (x^c + 1)$$

where

$p(x)$ = an irreducible polynomial of degree m whose roots have order e; that is, the period of $p(x)$ is e

$c \geq b + d - 1$

$d \geq b$

$m \geq b$

$(c,e) = 1$, that is, c and e are relatively prime

For pure error correction purpose, the best choice is to set m = b and d = b. This results in a Fire Code with the parameters

$$n = \text{LCM}\ (e,c) = ec$$

$$n - k = c + m = 3b - 1$$

$$k = ec - (3b - 1)$$

Another version of the Fire Code, which is the half-length version, has the code parameters

$$n = \left(\frac{c}{2}\right) e$$

where c is an even integer

$$n - k = 3b - 2$$

$$k = \left(\frac{c}{2}\right) e - (3b - 2)$$

In the conventional decoder implemented by shift registers, decoding the Fire code requires 2n shifts; n shifts for parity checking and another n shifts for error correction. Since the first n shifts attributed to parity checking are concurrent with the reading operation, no time delay occurs in this operation. The second n shifts needed to locate the error burst and correct it is a time delay due to error correction. If n is very large, like 16 384

bits in a TENEX page, then this long time delay may be intolerable in real life operation. However, if there exists, within easy reach, some power for general computation that is resident either within the host computer or within the controller of the storage system, then the second n shifts can be reduced to the minimum of (e + c - 2) and the computation time. This is essentially the technique used in the IBM 3830 disk controller.

In the UNICON 690, a (630,614) Fire Code shortened to a (80,64) code was used. Since n = 80, on-line decoding was implemented.

Reed-Solomon Codes from GF(2^m)— The Reed-Solomon (RS) codes are non-binary codes with code symbols from a Galois field of 2^m elements GF(2^m). From coding theory, if p is a prime number and q is any power of p, there are codes with code symbols from a q-symbol alphabet. The codes are called q-ary codes. The RS codes are a special subclass of q-ary BCH codes. A t-error-correcting RS code has the following code parameters in GF(2):

Code length (bits): $n = m(2^m - 1)$

Number of parity-check bits: $n - k = 2mt$

Minimum distance: $d = 2t + 1$

Note that each code symbol is an m-tuple over GF(2), the RS code with error correcting capability t can be used to correct any of the following errors:

(1) All single bursts of length b_1, no matter where it starts, if

$$b_1 \leq m(t - 1) + 1$$

(2) Two bursts of length b_2 each, no matter where each burst starts, if

$$b_2 \leq m\,([t/2] - 1) + 1$$

(3) or any p-burst of length b_p, no matter where it starts, if

$$b_p \leq m\,([t/p] - 1) + 1$$

From these equations, it is concluded that the RS code can be used to correct random errors, single-burst errors, or multiple-burst errors.

The decoder of the RS code is very complex when compared to the Hamming Code and the Fire Code from GF(2). For this reason, this code should be used only if there are no other alternatives.

Interlaced Codes— The Interlaced Code is a burst-error-correcting cyclic code. This code has two interesting properties: (1) it decomposes burst errors into random errors, and (2) it reduces the problem of searching for a long efficient burst-error-correcting code to searching for a good short code.

Given an (n,k) cyclic code, it is possible to construct a $(\lambda n, \lambda k)$ cyclic code by interlacing. This is done by arranging λ code vectors in the original code into λ rows of a rectangular array and then transmitting them column by column. The resulting code is called an Interlaced Code. The code length is λ times as long with λ times as many information digits. The parameter λ is called the interlacing degree.

No matter where it starts, a burst of length λ will affect no more than one digit in each row of the array. Based on this concept, the error correcting capability of the Interlaced Code can be summarized as follows:

If the original code can correct	The Interlaced Code will correct
single errors	single bursts of length λ or less
t errors	t bursts of length λ or less
single burst of length b or less	single burst of length λb or less

Arithmetic Codes

Arithmetic codes are based on the concept of linear congruence from number theory. If $a_1 \equiv b_1 \bmod m$, $a_2 \equiv b_2 \bmod m$,..., $a_K \equiv b_K \bmod m$, then

$$a_1 + a_2 \cdot \cdot \cdot + a_K \equiv (b_1 + b_2 + \cdot \cdot \cdot + b_K) \bmod m$$

$$a_1 \cdot a_2 \cdot \cdot \cdot a_K \equiv (b_1 \cdot b_2 \cdot \cdot \cdot b_K) \bmod m$$

$$a_1 - a_2 \equiv (b_1 - b_2) \bmod m$$

The above equations suggest an easy way to check the operations of addition, multiplication, and subtraction. A similar equation for division does not exist. Therefore, division must be checked indirectly. For implementation purpose, the concept in the above equations can be written

$$R(N_1) + R(N_2) \equiv R(N_1 + N_2) \bmod m$$

$$R(N_1) \cdot R(N_2) \equiv R(N_1 \cdot N_2) \bmod m$$

where $R(N_1)$ is the residue of N_1 mod m. Subtraction is a subset of addition since, in most modern computers, subtraction is performed as 2's-complement (or 1's complement) addition.

There are two basic classes of arithmetic codes: separate and nonseparate. In practice, separate residue codes are used because of their ease of implementation. Let N be the number to be protected. Let R(N) be the check-symbol of N computed as residue of N modulo m. Then [N, R(N)] forms a codeword in the separate code, where N and R(N) can be separately

processed. As an example, consider the case of adding $N_1 = 43$ and $N_2 = 55$, with the check-base m = 15. If the addition has no error, then

$$R(43) + R(55) \equiv R(43 + 55) \mod 15$$

$$13 + 10 \equiv R(98) \mod 15$$

$$8 \equiv 8 \mod 15$$

Note that the codeword for $N_1 = 43$ is [43,13]. The error detecting capability is in the check-base m. For m = 15, the code can detect a burst length of 3. Since m = 15, the code requires 4 check bits to represent R(N).

For checking additions only, there is another method of detecting single-failure errors in an adder. This method is called parity-prediction, and IBM has implemented this method in the adder of System 360 Model 50 (ref. 10).

APPLICATION TO PHOENIX SYSTEM

The goal of the Phoenix project is to develop a computer system to succeed ILLIAC IV. Currently, ILLIAC IV has a total of 64 Processing Elements (PE's) that operate at a speed of about 40 megaflops (millions of floating-point operations per second). The design goal of Phoenix is to have 1024 PE's that can operate at about 10 000 megaflops (ref. 29). With this number of PE's and the expected operating speed, system reliability is a high priority consideration in the project. At this preliminary writing, the Phoenix system will have the following major subsystems:

Array Processing Elements
Array Buffer Memory
Problem Memory
Staging Memory
Archival Memory
Instruction Buffer
Instruction Memory
PE Data Permutation Network
Control Units

This section briefly describes how to make each subsystem fault-tolerant.

Array Processing Elements

The Phoenix System initially will consist of some multiple of 64 Processing Elements (PE's), and be capable of expanding to 1024 PE's. The PE will be organized around a high-speed pipelined floating-point arithmetic unit. The performance of this arithmetic unit will be approximately that of

a modern large-scale computer. The data word length is 64 bits. The number system to be used will be the conventional binary system. Although residue number systems (RNS) have inherent error detecting and correcting capability, RNS will not be chosen for the Phoenix System because these number systems have not yet advanced far enough for practical applications (refs. 46 through 49). In the notation of Sterbenz (ref. 50), the floating-point arithmetic chosen is

$$FP(r,p,R) = FP(2,48,op)$$

where r is the base of the number system, p is the precision that designates the number of r digits contained in the mantissa, and R specifies how the arithmetic is to be rounded. Either optimum rounding in the definition of Yohe (ref. 51), or ROM rounding in the definition of Kuck (ref. 52) will be used. The exponent field is 15 bits and its bias is 16 384.

Arithmetic operations in the PE will be protected by a Separate Residue Code for error detection. The check-base m for the code is selected to be

$$m = 2^c - 1 = 15 \qquad \text{for } c = 4$$

This choice of m will simplify the residue checking logic, since c divides p, and thus $(2^c - 1)$ divides $(2^p - 1)$, hence

$$|2^p - 1|_m = |2^{48} - 1|_{15} = 0$$

where the notation $|A|_m$ means the residue of A modulo m. Also, this choice of m = 15 will detect all single arithmetic errors as well as all single burst errors of length $b \leq 3$. The implementation of this mod-15 residue checker requires approximately 20 IC's with a total delay of 12 ns.

During user program computation, any error detected in a PE will automatically start a retry from the previous program checkpoint. If the retry fails, a standby spare PE will be logically activated to replace the faulty PE. At this writing, the ratio between active PE's and standby PE's is 64 to 1.

Other operations in the PE such as shift, rotate, complement, and some logical operations can also be protected either by Residue Codes or by parity checks. However, the detail of such discussions will not be presented here.

Array Buffer Memory

The Array Buffer Memory, or Buffer Memory (BM), is the working memory. It holds the working data set during computation, and it is also used as a data buffer by the Permutation Network during data routing between PE's. The BM communicates with a backup memory, called the Problem Memory (PM), for swapping working data sets. The BM will be divided into smaller modules,

one module per PE. The size of each module is 4096 64-bit words. For 1024 PE's, the total BM capacity is 2^{22} words, or 4 million words.

The design of the BM will use static RAM IC's. In the 1980 time frame, one can expect the availability of a 4096-bit static RAM with an access time of 15 ns. To build a 4-million-word BM, 65 536 IC's are required. As previously noted, the expected IC MTBF for λ = 0.02% per 1000 h is 75 h. The BM system MTBF is about 50 h when all other supporting hardware such as data path registers, cable transceivers, and the like are included.

In order to increase the BM system MTBF, the (127,119) Hamming Code shortened to a (72,64) code, described in an earlier section, will be used. The expected system MTBF is increased to 167 000 h, assuming a 24 h MTTR.

Problem Memory

The Problem Memory (PM) is a large backup memory used to store the entire program data base. It is desirable that PM be a random-access memory with a cycle time of about 200 ns. For this performance, the PM design can use MOS RAM's. At this writing, the size of PM is 0.5 million words per PE. For 1024 PE's, the total PM capacity is 34×10^9 bits.

In the 1980 time frame, one can expect the availability of a 65 536-bit MOS RAM with the specified performance. To build this PM, each module associated with the PE requires 512 IC's. For 1024 PE's, a total of 524 288 IC's are required with an expected system MTBF of about 5 h. With the total system divided into smaller modules of 0.5 million words each and the Hamming Code used in conjunction with a MTTR of 24 h, the system MTBF can be increased to about 3066 h.

Staging Memory

The Staging Memory (SM) is a large rotating disk memory used to stage Phoenix-bound programs. It is used principally as a staging device between the Archival Memory and the PM. At this writing, the size of SM should be about twice that of PM. The design of SM will use proven technology, as do today's 400 megabyte disk systems. The total system capacity is about 10^{11} bits. The system reliability will be further enhanced by on-line error-correcting Interlaced Code to be implemented in the SM Controller.

Archival Memory

The Archival Memory (AM) is a very large storage system used to store all user data and programs. The system capacity is at least 5×10^{13} bits. The reliability of the system must be at least as reliable as today's disk memory systems. At this writing, the exact technology used to implement the AM is not known. However, one observation on today's research in very large storage systems is that up to about 10^{13} bits, the most feasible technology

is magnetic. Beyond 10^{13} bits, the electronic ion technology is promising. The laser technology, on the other hand, is also emergent, but more research is needed on the problems of high-density tracking (3000 tracks per inch) and data obliterated by dirt.

Instruction Memory and Instruction Buffer

In the Phoenix System, the user program will be separated into two parts: the instructions and the data. The data part, as mentioned earlier, is stored in PM. The instruction part is stored in the Instruction Memory (IM). The size of IM is about two million 64-bit words. At this writing, a 64-bit instruction word length is contemplated. The exact word length will be known as soon as a formal description of the instruction set is available.

The Instruction Buffer (IB) is a buffer that holds the instructions to be executed. IB fetches its instructions from IM in a look-ahead manner. The size of IB is 128 instructions, which is the same as that in ILLIAC IV.

The design of IM will be similar to that of the Array Buffer Memory, for which a 4096-bit static RAM will be used. To build a 2-million word IM, 32 768 IC's are required, which is the same size as BM. Again, the Hamming Code will be used to increase the system reliability. This Hamming Code will protect both the IM and the IB, since the decoder will be located in IB. The 10 ns decoding time will be masked in IB by an instruction look-ahead feature.

PE Data Permutation Network

The PE Data Permutation Network (PN) is a data routing network capable of simultaneously routing data between PE's within the Phoenix System (ref. 53). It is a much more powerful routing network than that used in the ILLIAC-IV computer. The complexity of PN is about 40,000 IC's, with an expected system MTBF of about 127 h. It is desirable to increase this MTBF by about 10 times to at least 1000 h. The Hamming Code will be used to protect the data paths. At this writing, the PN path is 16 bits, and thus a (21,16) Hamming Code can be used for single-bit error correction. For the protection of the control logic, the method of error detection for combinational and sequential circuits, described in reference 41, can be used.

Control Units

The Control Units (CU's) of the Phoenix System are divided into two levels. The first level of control is the Central Control Unit (CCU). The second level of control is the Sextant Control Unit (SCU). For reliability enhancement, error-correcting codes, error-detecting logic, and duplications will be used, depending upon the circuit to be protected. In circuits like data buses and registers, error-correcting codes will be used. In control

circuits, where the use of codes is not possible, combinational and sequential error-detecting logic and duplications will be used.

CONCLUSION AND PROSPECTUS

This report, in a somewhat tutorial manner, addressed the concept of fault-tolerant computing and its application to large computer systems. Fault-tolerant computing was introduced from the viewpoints of (1) concurrent error detection and correction, (2) program rollback and retry, and (3) replacement of faulty components by standby spares. Large computer systems, in the class of ILLIAC-IV or larger, have problems in the areas of reliability, availability, and serviceability. Based upon operational experiences with ILLIAC-IV and observations of other large computer systems currently in operation, the consensus of opinion at IAC is that fault-tolerant computing is a preamble for assuring large computer system viability.

Research and development of large computer systems is currently in its infancy. Based upon our present assessment of relevant large-scale computational problems, one can foresee that the demand for large computer systems will probably increase rather than decrease in the future. To make these large systems viable, fault-tolerant computing will probably become a standard design requirement for these systems. Exactly how fault-tolerant computing is to be designed for future large systems will, of course, evolve with changing technologies.

Ames Research Center
National Aeronautics and Space Administration
Moffett Field, Calif. 94035, June 9, 1977.

REFERENCES

1. The Holy Bible: 1 Samuel, Chapter 17, verse 40.

2. Von Neumann, John: Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. Automata Studies, C.E. Shannon and J. McCarthy, eds., Princeton University Press, 1956, pp. 43-98.

3. Everett, R.R.; Zraket, C.A.; and Benington, H.D.: SAGE — Data-Processing System for Air Defense. Proc. East. Joint Comp. Conf., Washington, D.C., Dec. 1957, pp. 148-155.

4. Vance, P.R.; Dooley, L.G.; and Diss, C.E.: Operation of SAGE Duplex Computers. Proc. East. Joint Comp. Conf., Washington, D.C., Dec. 1957, pp. 160-163.

5. Lewis, Thomas B.: Primary Processor and Data Storage Equipment for the Orbiting Astronomical Observatory. IEEE Trans. on Electronic Comp., Dec. 1963, vol. EC-12, no. 5, pp. 677-687.

6. Downing, R.W.; Nowak, J.S.; and Tuomenoksa, L.S.: No. 1 ESS Maintenance Plan. Bell Sys. Tech. Jour., Sept. 1964, no. 5, pt. 1, pp. 1961-2019.

7. Kennedy, Peter J.; and Quinn, Thomas M.: Recovery Strategies in the No. 2 Electronic Switching System. Digest of 1972 Int. Symp. on Fault Tolerant Computing, IEEE Computer Society, June 1972, pp. 165-169.

8. Hardie, Fred H.; and Suhocki, Robert J.: Design and Use of Fault Simulation for Saturn Computer Design. IEEE Trans. on Electronic Comp., Aug. 1967, vol. EC-16, no. 4, pp. 412-429.

9. Blaauw, G.A.: The Structure of SYSTEM/360: Part V - Multisystem Organization. IBM Sys. Jour., vol. 3, no. 2, 1964, pp. 181-195.

10. Langdon, G.G. Jr.; and Tang, C.K.: Concurrent Error Detection for Group Look-ahead Binary Adders. IBM J. Res. Rev., Sept. 1970, vol. 14, no. 5, pp. 563-573.

11. Tang, D.T.; and Chien, R.T.: Coding for Error Control. IBM Sys. Jour., vol. 8, no. 1, 1969, pp. 48-86.

12. Avizienis, Algirdas; Gilley, George C.; Mathur, Francis P.; Rennels, David A.; Rohr, John A.; and Rubin, David K.: The STAR (Self-Testing-And-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design. IEEE Trans. on Computers, Nov. 1971, vol. C-20, no. 11, pp. 1312-1321.

13. Corbato, F.J.; Saltzer, J.H.; and Clingen, C.T.: Multics — The First Seven Years. AFIPS Conf. Proc., SJCC, May 1972, pp. 571-583.

14. A Guide to the IBM System/370 Model 145. Third ed. Tech. Publ. Dept., IBM Corp., 1133 Westchester Ave., White Plains, N.Y., Aug. 1972.

15. Amdahl 470 V/6 Features. Amdahl Corp., 1975.

16. Stiffler, J.J.: The SERF Fault-Tolerant Computer, Part I: Conceptual Design. Digest of 1973 Int. Symp. on Fault-Tolerant Computing, Palo Alto, Calif., IEEE Computer Society, 1973, pp. 23-26.

17. Parke, N.G. IV; and Barr, P.C.: The SERF Fault-Tolerant Computer, Part II: Implementation and Reliability Analysis. Digest of 1973 Int. Symp. on Fault-Tolerant Computing, Palo Alto, Calif., IEEE Computer Society, 1973, pp. 27-31.

18. Design of a Modular Digital Computer System DLR 4 and 5, Final and Phase III Report. (Prepared under Contract NAS8-27926.) Hughes Aircraft Co., Fullerton, Calif., Dec. 1973.

19. Carter, W.C.; Jessep, D.C.; Wadia, Aspi B.; Schneider, Peter R.; and Bouricius, Willard G.: Logic Design for Dynamic and Interactive Recovery. IEEE Trans. on Comp., Nov. 1971, vol. C-20, no. 11, pp. 1300-1305.

20. Conn, Ralph B.; Alexandridis, Nikitas A.; and Avizienis, Algirdas: Design of a Fault-Tolerant, Modular Computer with Dynamic Redundancy. AFIPS Conf. Proc., FJCC, 1972, pp. 1057-1067.

21. Watson, W.J.: The TI ASC — A Highly Modular and Flexible Super Computer Architecture. AFIPS Conf. Proc., AFIPS Press, N.J., FJCC, 1972, pp. 221-228.

22. Wilcox, Richard H.; and Mann, William C., eds.: Symposium on Redundancy Techniques for Computing Systems, Washington, D.C., 1962. Spartan Books, 1962.

23. Falk, Howard, Managing Editor: Reaching for a gigaflop. IEEE Spectrum, Oct. 1976, vol. 13, no. 10, pp. 65-69.

24. Cronk, M. (Staff Writer): Math Professor's Formula 'Proves' Free Enterprise. San Jose News, Local Section, Section B, Page 1B, San Jose, Calif., April 24, 1976.

25. Elmendorf, W.R.: Fault-Tolerant Programming. Digest of 1972 Int. Symp. on Fault-Tolerant Computing, Newton, Mass., IEEE Computer Society, 1972, pp. 79-83.

26. Avizienis, Algirdas: Design of Fault-Tolerant Computers. AFIPS Proc., Thompson Books, Washington, D.C., FJCC, 1967, pp. 733-743.

27. Hopkins, A.L.: A Fault-Tolerant Information Processing Concept for Space Vehicles. IEEE Trans. on Comp., Nov. 1971, pp. 1394-1403.

28. Bouricius, W.C.; Carter, W.C.; Roth, J.P.; and Schneider, P.R.: Investigations in the Design of an Automatically Repaired Computer. Digest of First Annual IEEE Comp. Conf., IEEE N.Y., 1967, pp. 64-67.

29. Institute for Advanced Computation: A Prospectus for High-Speed Computing. IAC Phoenix Project Memo. No. 006, Feb. 25, 1977. Institute for Advanced Computation, Sunnyvale, Calif. 94086.

30. Hamming, R.W.: Error Detecting and Error Correcting Codes. Bell Sys. Tech. Jour., April 1950, vol. 29, no. 2, pp. 147-160.

31. Peterson, William W.: Error-Correcting Codes. MIT Press, Cambridge, Mass., 1961.

32. Berlekamp, Elwyn R.: Algebraic Coding Theory. McGraw-Hill, N.Y., 1968.

33. Lin, Shu: An Introduction to Error-Correcting Codes. Prentice Hall, Englewood Cliffs, N.J., 1970.

34. Peterson, W. W.; and Weldon, E.J.: Error-Correcting Codes. MIT Press, Cambridge, Mass., 1972.

35. Barlow, R.W.; and Proschan, F.: Mathematical Theory of Reliability. John Wiley & Sons, 1965.

36. Amstadter, Bertram L.: Reliability Mathematics; Fundamentals; Practices; Procedures. McGraw-Hill, N.Y., 1971.

37. Levine, Len; and Meyers, Ware: Semiconductor Memory Reliability with Error Detecting and Correcting Codes. Computer, IEEE Computer Society, Oct. 1976, pp. 43-50.

38. Ingle, Ashok D.; and Siewiorek, Daniel P.: Reliability Models for Multiprocessor Systems With and Without Periodic Maintenance. Department of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Sept. 1976. (AD-A034 854/0ST)

39. Berlekamp, E. R., ed.: Key Papers in the Development of Coding Theory. IEEE Press, N.Y., 1974.

40. Slepian, David, ed.: Key Papers in the Development of Information Theory. IEEE Press, N.Y., 1973.

41. Sellers, Frederick F.; Hsiao, M.Y.; and Bearnson, L.W.: Error Detecting Logic for Digital Computers. McGraw-Hill, N.Y., 1968.

42. Rao, T.R.: Error Coding for Arithmetic Processors. Academic Press, N.Y., 1974.

43. Piece, William: Failure-Tolerant Computer Design. Academic Press, N.Y., 1965.

44. A high-speed decoding of the Fire-Code is implemented in the IBM 3830 disk controller for the IBM 3330 disk storage system. IBM 3330/3830 Maintenance Manual, Aug. 1971.

45. Lim, R.S.: Augmented Burst-Error Correction for UNICON Laser Memory. NASA TM X-62,442, June 1974.

46. Szabo, Nicholas S.; and Tanaka, Richard I.: Residue Arithmetic and Its Applications to Computer Technology. McGraw-Hill, New York, 1967.

47. Yau, Stephen S.; and Liu, Yu-Cheng: Error Correction in Redundant Residue Number Systems. IEEE Trans. on Comp., Jan. 1973, vol. C-22, no. 1, pp. 5-11.

48. Barsi, Ferruccio; and Maestrini, Piero: Error Correcting Properties of Redundant Residue Number Systems. IEEE Trans. on Comp., March 1973, vol. C-22, no. 3, pp. 307-315.

49. O'Keefe, Kenneth H.: A Note on Fast Base Extension for Residue Number Systems with Three Moduli. IEEE Trans. on Comp., Nov. 1975, vol. C-24, no. 11, pp. 1132-1133.

50. Sterbenz, Pat H.: Floating-Point Computation. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.

51. Yohe, J. Michael: Roundings in Floating-Point Arithmetic. IEEE Trans. on Comp., June 1973, vol. C-22, no. 6, pp. 577-586.

52. Kuck, D.J.; Parker, D.S.; and Sameh, A.H.: ROM-Rounding: A New Rounding Scheme. IEEE 3rd Symposium on Computer Arithmetics, Southern Methodist Univ., Dallas, Texas, Nov. 1975, pp. 67-72.

53. Feierbach, Gary F.; and Stevenson, David K.: A Feasibility Study of Programmable Switching Networks for Data Routing. IAC Phoenix Project Memo 003, May 1977. Institute for Advanced Computation, 1095 E. Duane Ave., Sunnyvale, Calif. 94086.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TP 1067		

4. Title and Subtitle

FAULT-TOLERANT COMPUTING: A PREAMBLE FOR ASSURING VIABILITY OF LARGE COMPUTER SYSTEMS

5. Report Date: October 1977

6. Performing Organization Code:

7. Author(s)

Raymond S. Lim

8. Performing Organization Report No.: A-7072

9. Performing Organization Name and Address

Ames Research Center
Moffett Field, Calif. 94035

10. Work Unit No.: 366-00-00

11. Contract or Grant No.:

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Washington, D.C. 20546

13. Type of Report and Period Covered: Technical Paper

14. Sponsoring Agency Code:

15. Supplementary Notes

16. Abstract

This paper describes why large computer systems need to be more fault-forgiving, or fault-tolerant, in order to be viable. The need for fault-tolerant computing is addressed from the viewpoints of (1) why it is needed, (2) how to apply it in the current state of technology, and (3) what it means in the context of the Phoenix computer system and other related systems. To this end, the value of concurrent error detection and correction is described from the viewpoints of (1) user protection, (2) program retry, and (3) repair. The technology of algebraic codes to protect memory systems and arithmetic codes to protect arithmetic operations is discussed.

17. Key Words (Suggested by Author(s))

Fault-tolerant computing
Error detection-correction
Reliable computers
Redundancy

18. Distribution Statement

Unlimited

STAR Category - 60

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	24	$3.25

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

NASA-Langley, 1977

5 1 1U,G, 091977 S00903DS
DEPT OF THE AIR FORCE
AF WEAPONS LABORATORY
ATTN: TECHNICAL LIBRARY (SUL)
KIRTLAND AFB NM 87117