# Modeling and Measurement of Error Propagation in a Multimodule Computing System

KANG G. SHIN, SENIOR MEMBER, IEEE, AND TEIN-HSIANG LIN, STUDENT MEMBER, IEEE

*Abstract*—**Because of excessive resource requirements, error detection mechanisms in real computing systems cannot usually be made complete, i.e., their coverage is less than 100 percent. This fact in turn implies that an error may propagate through system components before it is detected. The main purpose of this paper is to develop an error propagation model and methods to compute and measure the model parameters, i.e., distributions of error propagation times, rather than applying the model to various design and analysis problems.**

**A digraph model is used to represent a multimodule computing system and error propagation in the system is modeled by general distributions of error propagation times between all pairs of modules. Two algorithms are developed to systematically and efficiently compute the distributions of error propagation times. Experiments are also conducted to measure the distributions of error propagation times within the fault-tolerant multiprocessor (FTMP). Statistical analysis of experimental data shows that the error propagation times in FTMP do not follow a well-known distribution, the Weibull distribution, thus justifying the use of general distributions in our model.**

*Index Terms*—**Digraph, distribution and density functions of error propagation times, error propagation, experiment on FTMP, statistical analysis.**

## I. INTRODUCTION

IN ANY computing system, it is practically impossible to install a perfect detection mechanism with which all types of errors can always be detected before they propagate to other parts of the system. Thus, upon detection of an error, it is difficult to tell whether the error is induced by a fault that occurred in the same part of the system where the error is detected or it is the propagation of an error induced by a fault in some other part of the system. In other words, an error may propagate through the system components before it is detected.

To clarify the terminology used in this paper, an *error* is defined as an incorrect state of the system which could be an incorrect data, an incorrect control signal, or an abnormal system behavior, and a *fault* is the source of an error, e.g., a broken wire, an electrical short, or a bug in a program. The effects of error propagation on fault location, reconfiguration, and error recovery are significant, because of the uncertainty as to which components are really faulty and/or erroneous [1]. Most approaches reported in the literature circumvent the problem of error propagation by assuming a perfect coverage in detecting errors. However, such an assumption is unrealistic and, often, unacceptable for real systems, since even a near-perfect detection mechanism is very difficult to obtain without entailing an excessive amount of resources or performance degradation. It is therefore necessary to consider the error propagation problem in the design and analysis of fault-tolerant systems. As a first step to meet such a need, we propose, in this paper, a general error propagation model and examine its power and limitations.

Error propagation was first recognized and utilized in testing combinational circuits. The famous $D$-algorithm is an example of using the error propagation property of logic gates to generate test patterns for combinational circuits [2]. Recently, error propagation properties at the logic gate level [3] and the transistor level [4] have been derived for the design of totally self-checking integrated circuits. In [5], error propagation at the register level has been considered for the design of a strongly fault secure processor. Zielinski [6] has proposed a model of error propagation among communicating processes in a distributed computer system and used it to express an error recovery method with the recovery block scheme.

All the previous error propagation models are *deterministic* in nature, because they are based on a specific fault/error model or the system is assumed to have some restricted, predictable behavior. However, there is in practice very little *a priori* information on the behavior of faults and errors, and intercomponent communications may take place in an arbitrary fashion. So, error propagation cannot in general be modeled deterministically. In this paper, we propose a stochastic model where the primary parameters—error propagation times between all pairs of system components—are random variables. We shall also show how the distributions of error propagation times can be determined systematically and efficiently.

In addition to the development of an error propagation model, we shall show by example a method to get the error propagation times for real systems, i.e., direct measurement on a target system. This method is in sharp contrast with that in [7], where the authors measured the distribution of error propagation times via a gate level simulation for the CPU in an avionic miniprocessor. We focus here only on the modeling and measurement of error propagation. Application of the error propagation model for fault location, damage assessment, and error recovery will be addressed in forthcoming papers.

Manuscript received April 21, 1987; revised August 25, 1987. This work was supported in part by NASA Grant NAG-1-296 and ONR Contract N00014-85-K-0531. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

The authors are with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8718421.

In Section II, our error propagation model is described along with its justification. The system consists of multiple components and is represented by a digraph. The error propagation time between a pair of components in the system is the minimum of propagation times over all paths between the two components. Thus, it is necessary to systematically enumerate all paths between any pair of components. Because some of these paths have nodes and edges in common, the distribution functions of their propagation times are dependent on one another and, thus, very complex to derive. Two algorithms are presented in Section III to systematically enumerate propagation paths and efficiently compute the distribution functions. Section IV describes an experiment to measure the error propagation times on the fault-tolerant multiprocessor (FTMP) [8]. The experimental results are analyzed to obtain the distribution functions of error propagation times. The paper concludes with some remarks in Section V.

## II. Error Propagation Model

Consider a computing system composed of multiple components, each of which is called a *module*. A module represents any well-defined component of the system; it could be a hardware or software unit or even a combination of hardware or software. Each module is a self-contained entity with input/output from/to others.

*Definition 1:* A module is said to be *faulty* if a fault has occurred in the module and is said to have been *contaminated* if it contains one or more errors.

*Definition 2:* For each module in the system, the *faulty moment* is the time instant a fault occurs within the module, and the *contaminating moment* is the time instant the first error occurs due to either the manifestation of a fault within the module or the propagation of error(s) from other module(s).

*Definition 3:* For a module, the interval between the faulty moment and the contaminating moment is called the *fault latency* of the module.

### The Error Propagation Time

Error propagation among the modules can best be described by a digraph, denoted by $D = (V, E)$, where $N$ is the number of modules in the system, $V = \{v_1, \cdots, v_N\}$ is the set of nodes, and $E = \{e_{ij}: 1 \le i, j \le N\}$ is the set of directed edges. Each node in $D$ represents a module in the system, and the directed edge $e_{ij}$ represents the communication link from $v_i$ to $v_j$. Typical methods of communication between software modules are message passing and shared memory. Hardware modules can communicate via control and data signals. If there is no communication link from $v_i$ to $v_j$, then $E$ will not contain $e_{ij}$, or $e_{ij}$ is a null edge. A *propagation path* from $v_i$ to $v_k$, written as $(v_i, \cdots, v_k)$, is a directed path in $D$ where all nodes in the path are *distinct* so that an error may propagate from $v_i$ to $v_k$ through the path. It is meaningless to consider the case of error propagation into a module which has already been contaminated; this is the very reason why the nodes in a propagation path are distinct. Errors may propagate from $v_i$ to $v_k$ if there exists at least one propagation path from $v_i$ to $v_k$.

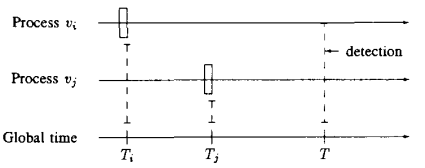


Fig. 1. Timing diagram of processes $v_i$ and $v_j$.

*Definition 4:* The *error propagation time* from $v_i$ to $v_j$, denoted by $X_{ij}$, is defined as the time interval between the contaminating moment of $v_i$ and that of $v_j$. The density function and cumulative distribution function of $X_{ij}$ are denoted by $g_{ij}(\cdot)$ and $G_{ij}(\cdot)$, respectively.

Clearly, $X_{ij}$ holds a physical meaning only when $X_{ij} \ge 0$. Thus, the definition of $X_{ij}$ is usually made under the assumption that $v_i$ is the only faulty module or the first module to be contaminated in the system. If the system is assumed to have at most one fault at a given time, the faulty module will be the first module to be contaminated.

Many useful pieces of information can be derived from $g_{ij}$'s and $G_{ij}$'s. For example, $g_{ij}$'s and $G_{ij}$'s can be used for evaluating the rollback recovery block scheme for concurrent cooperating processes [9]. Let each node (edge) in our graph model represent a process (interprocess communication). Each process is assumed to establish its own recovery point asynchronously with respect to the other processes. Before establishing a recovery point, an acceptance test is performed which is assumed to detect all types of errors, i.e., perfect coverage. Assume that an error is detected at time $T$ by process $v_k$ during the acceptance test and that process $v_i$ is diagnosed to be the source of the error. Then, for the subsequent recovery, it is useful to calculate the probability of some other process $v_j$ being contaminated, i.e., *damage assessment*. Let the most recent recovery points of $v_i$ and $v_j$ be established at time $T_i$ and $T_j$, respectively, as shown in Fig. 1. Since all the acceptance tests are assumed to have perfect coverage, the $v_i$'s contaminating moment, denoted by $CI$, must lie within the interval $[T_i, T]$ and the $v_j$'s contaminating moment, denoted by $CJ$, must be within the interval $[T_j, T]$. The probability of $v_j$ being contaminated can then be computed as

$$\begin{aligned}
&\int_{T_i}^{T}\int_{T_j}^{T} f_{CI}(x)\cdot f_{CJ}(y)\,dy\,dx \\
&\quad = \int_{T_i}^{T_j}\int_{T_j}^{T} \frac{1}{T-T_i}\cdot g_{ij}(y-x)\,dy\,dx \\
&\qquad + \int_{T_j}^{T}\int_{x}^{T} \frac{1}{T-T_i}\cdot g_{ij}(y-x)\,dy\,dx \\
&\quad = \frac{1}{T-T_i}\int_{T_i}^{T_j}(G_{ij}(T-x)-G_{ij}(T_j-x))\,dx \\
&\qquad + \frac{1}{T-T_i}\int_{T_j}^{T} G_{ij}(T-x)\,dx.
\end{aligned}$$

The knowledge of this probability can help us decide whether or not to roll $v_j$ back in case of $v_i$'s detection of an error.

The contaminated region within the system can be estimated from the collection of all modules' contaminating probabilities, which are determined by $g_{ij}$ and $G_{ij}$ as shown above.

*Direct Error Propagation Time*

Although error propagation times contain complete information on the behavior of error propagation and can be directly measured experimentally, there are several drawbacks as follows.

- It is very costly to measure error propagation times for all pairs of modules. For a system with $N$ modules, $N(N-1)$ error propagation times must be measured experimentally regardless of the number of communication links in the system.
- The distribution of any error propagation time is fixed under a specific fault model. However, should a new fault model be needed, all error propagation times must be measured again under the new fault model, since the distributions change with the fault model.
- The error propagation times over different paths are dependent on one another whenever they have some path segments in common. Also, the dependencies among the error propagation times are very difficult to experimentally measure but necessary to compute their joint distribution. A useful joint distribution, for example, is

$$\text{Prob } [X_{i1} \le x_1, \cdots, X_{iN} \le x_N]$$

which characterizes the spread of error(s) in the system from a faulty module $v_i$.

To overcome the above drawbacks, a *direct propagation time* is defined as follows.

*Definition 5:* If $e_{ij}$ is a nonnull edge in $E$, then the *direct propagation time,* denoted by $B_{ij}$, is the time for an error to propagate from $v_i$ to $v_j$ via $e_{ij}$. The density function and the cumulative distribution function of $B_{ij}$, denoted by $p_{ij}$ and $P_{ij}$, respectively, are called the *direct propagation functions* of $e_{ij}$.

The differences between an error propagation time and a direct propagation time lie in that 1) the latter is associated with a directed edge while the former is defined for every ordered pair of modules, and 2) the latter accounts for error propagation through a particular edge while the former is the minimum propagation time over all propagation paths between the given pair of modules. We shall show later how to systematically and efficiently compute error propagation times from direct propagation times. Since direct propagation times are defined for the communication links in the system, without loss of generality, one can assume that they are independent of one another and their distributions will not change with the underlying fault models. Moreover, this assumption greatly reduces the experimental cost to measure propagation times. For example, a five-node-eight-edge system would require 20 measurements of error propagation times for *each* fault model, but would require only eight measurements of direct propagation times for *all* fault models.

From the direct propagation functions, another useful function called the *error containment function,* $\text{EC}_i(t)$, is defined for a module $v_i$ as the probability that errors have not propagated from $v_i$ to other modules up to time $t$ (measuring from the $v_i$'s contaminating moment). For example, if $v_i$ has outgoing communications with $v_j$, $v_k$, and $v_m$, then $\text{EC}_i(t)$ can be calculated as

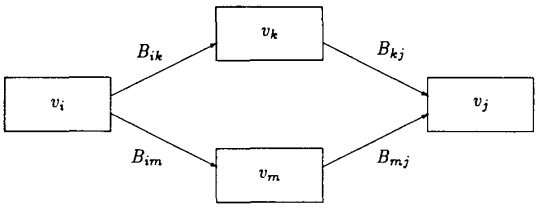

Fig. 2. An example graph.

$$\begin{aligned}\text{EC}_i(t) &= \text{Prob } [B_{ij} > t, B_{ik} > t, B_{im} > t] \\ &= (1 - P_{ij}(t))(1 - P_{ik}(t))(1 - P_{im}(t))\end{aligned}$$

because $B_{ij}, B_{ik}$, and $B_{im}$ are independent of one another.

*Join, Meet, and Conditioning Operations*

We have discussed the advantages of measuring direct propagation times instead of error propagation times. The immediate problem is then to compute error propagation times from direct propagation times. Consider an example system shown in Fig. 2. To get $X_{ij}$, one must first find all propagation paths from $v_i$ to $v_j$. The propagation time along a propagation path is the sum of all the direct propagation times of the edges in the path. Then, $X_{ij}$ is the minimum propagation time over all propagation paths, i.e.,

$$X_{ij} = \min (B_{ik} + B_{kj}, B_{im} + B_{mj}).$$

The distribution of $X_{ij}$ can be calculated by the following lemmas, since direct propagation times are independent of each other.

*Lemma 1:* If $Z = Y_1 + \cdots + Y_n$ and $Y_i, i = 1, \cdots, n$, are independent, the density function of $Z$ can be calculated by

$$f_Z(t) = f_{Y_1}(t) * \cdots * f_{Y_n}(t)$$

where $*$ denotes the convolution and $f_{Y_i}(t)$ is the density function of $Y_i$.

*Lemma 2:* If $Z = \min (Y_1, \cdots, Y_n)$ and $Y_i, i = 1, \cdots, n$, are independent, the cumulative distribution function of $Z$ can be calculated by

$$F_Z(t) = 1 - (1 - F_{Y_1}(t)) \cdots (1 - F_{Y_n}(t))$$

where $F_{Y_i}(t)$ is the cumulative distribution function of $Y_i$.

We shall call the computation in Lemma 1 a *join* operation and that in Lemma 2 a *meet* operation. Usually, the join operation is performed over the direct propagation times of all the edges in a propagation path and the meet operation is performed over a propagation path between a given pair of modules. However, because of the independence requirement in the meet operation, only disjoint propagation paths can become the operands of a meet operation. (Propagation paths with no common edges are said to be *disjoint.*) If propagation paths are not disjoint, i.e., some paths have common edges, the meet operation is not applicable since some operands are dependent. The dependence

problem could sometimes be eliminated by rearranging the order of join and meet operations so that the operands in the meet operation can be made independent. For example, $\min(X + Y, X + Z) = X + \min(Y, Z)$, where $X, Y$, and $Z$ are independent variables. The meet operation can be applied on $Y$ and $Z$, but not on $X + Y$ and $X + Z$.

Unfortunately, rearranging the order of operations cannot always solve the dependence problem. Consider, for example, the distribution of a random variable $V = \min(X + Y, X + Z, Y + W)$, where $W, X, Y$, and $Z$, are independent. It is impossible to obtain $f_V(t)$, the density function of $V$, by using the join and meet operations only. Nevertheless, $f_V(t)$ can be calculated by *conditioning* $V$ on $X$, i.e.,

$$f_V(t) = \int_{-\infty}^{\infty} f_{V|X=x}(t) \cdot f_X(x)\,dx$$

$$= \int_{-\infty}^{\infty} f_{\min(x+Y, x+Z, Y+W)}(t) \cdot f_X(x)\,dx$$

$$= \int_{-\infty}^{\infty} f_{\min(x+Z, Y+\min(x, W))}(t) \cdot f_X(x)\,dx$$

and the distribution of the conditional random variable $V|X$ is obtainable by the join and meet operations since $W, Y, Z$ are independent and $x$ is a constant. The above calculation will henceforth be called a *conditioning operation* on the variable $X$.

The distribution of any $X_{ij}$ can be calculated by applying the conditioning operations on all the $B_{ij}$'s involved in the calculation. But this is equivalent to an exhaustive approach which is time consuming and should be avoided. In the next section, we shall derive two algorithms for 1) generating all propagation paths systematically and 2) rearranging the order of operations so as to minimize the number of conditioning operations.

## III. Algorithms for Computing $g_{ij}$ and $G_{ij}$

Let $M$ be the number of directed edges and $N$ be the number of nodes in a digraph $D$ representing a multimodule computing system. Then, $D$ can be represented by an *adjacency matrix* $A$, where $A_i^j$ is the entry of the $i$th row and the $j$th column of $A$, $A_i^j \neq 0$ means the existence of a nonnull edge from $v_i$ to $v_j$ and $A_i^j = 0$ means the absence of an edge in $D$. Also, let $A_i$ and $A^j$ denote the $i$th row vector and the $j$th column vector of $A$, respectively. The number of nonzero entries in $A_i$ ($A^j$) is represented by $\|A_i\|(\|A^j\|)$, and if $S$ is a set, $\|S\|$ denotes the cardinality of the set. To simplify the notation for the edges in $A$, they are numbered from 1 to $M$ first by their row positions then by their column positions so that $e_{ij}$ is also called the $A_i^j$th edge or, more conveniently, $e_m$ if $m = A_i^j \neq 0$. For the example graph $D_1$ shown in Fig. 3, $N = 5, M = 8$, and the adjacency matrix is

$$A = \begin{pmatrix} 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \\ 0 & 7 & 8 & 0 & 0 \end{pmatrix}$$

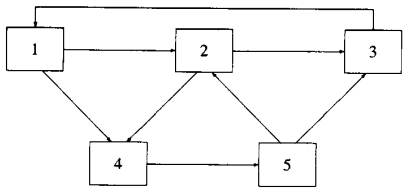

Fig. 3. The graph $D_1$.

where $e_{23}$ is called the third edge or $e_3$. $D_1$ will be used repeatedly as an example in the discussions that follow.

In order to calculate $g_{ij}$ and $G_{ij}$, two algorithms are developed: one, called PG, for systematically enumerating all propagation paths from $v_i$ to $v_j$ $\forall$ $i, j$ and the other, called OR, for computing distribution and density functions using the results from PG.

### *Path Generation*

The basic idea of PG is to simulate internode communications by passing *path tokens* around. A path token is represented by $[s_1, s_2, \cdots, s_m]$, where $s_1, \cdots, s_m$ are the ordered sequence of nodes traversed thus far. This path token was originally sent by node $s_1$ as $[s_1, s_2]$ to node $s_2$, and finally received by node $s_m$. Recall that, by definition of a propagation path, $s_1, \cdots, s_m$ should all be distinct. After receiving the path token, node $s_m$ records this path as a new propagation path from $v_{s_1}$ to $v_{s_m}$ and will attempt to append a new next node, if any, to the path token by checking both the adjacency matrix and the path token itself, and then pass the new path token to the next node. A path token stops generating new tokens when it cannot progress any further without visiting any node twice. Therefore, the algorithm will end when there are no path tokens being passed around. A node, denoted by node 0, is designated to keep track of the number of path tokens being passed around. The token count in node 0 is updated by all the other nodes by sending a count update message to node 0. Upon detection of no path tokens being passed around, node 0 issues a stop message to all the other nodes to terminate the algorithm.

A *parallel* implementation of PG can be accomplished by assigning a processing unit to each node provided that there are some communication mechanisms among the processing units for passing tokens around. The number of propagation paths between a pair of source and destination nodes, $(v_i, v_j)$, is stored in $NP_{ij}$. Initially, it is assumed that the adjacency matrix $A$ is available to every node and all $NP_{ij}$'s are set to zero.

*Algorithm PG:*

- For node 0
  1. **count** : = 0.
  2. Wait and receive any count update **cp** from other nodes.
  3. **count** : = **count** + **cp**.
  4. If **count** $\neq$ 0, go to step 2.
  5. Send a stop message to all the other nodes.
  6. Stop.

TABLE I
A RECORD OF RUNNING ALGORITHM PG

| Node 1 | | Node 2 | | Node 3 | | Node 4 | | Node 5 | |
|---|---|---|---|---|---|---|---|---|---|
| receive | send | receive | send | receive | send | receive | send | receive | send |
| | [1,2] | | [2,3] | | [3,1] | | [4,5] | | [5,2] |
| | [1,4] | | [2,4] | | | | | | [5,3] |
| [3,1] | [3,1,2] | [1,2] | [1,2,3] | [2,3] | [2,3,1] | [1,4] | [1,4,5] | [4,5] | [4,5,2] |
| | [3,1,4] | | [1,2,4] | [5,3] | [5,3,1] | [2,4] | [2,4,5] | | [4,5,3] |
| | | [5,2] | [5,2,3] | | | | | | |
| | | | [5,2,4] | | | | | | |
| [2,3,1] | [2,3,1,4] | [3,1,2] | [3,1,2,4] | [5,2,3] | [5,2,3,1] | [3,1,4] | [3,1,4,5] | [1,4,5] | [1,4,5,2] |
| [5,3,1] | [5,3,1,2] | [4,5,2] | [4,5,2,3] | [4,5,3] | [4,5,3,1] | [1,2,4] | [1,2,4,5] | | [1,4,5,3] |
| | [5,3,1,4] | | | [1,2,3] | | [5,2,4] | | [2,4,5] | [2,4,5,3] |
| [5,2,3,1] | [5,2,3,1,4] | [5,3,1,2] | [5,3,1,2,4] | [4,5,2,3] | [4,5,2,3,1] | [3,1,2,4] | [3,1,2,4,5] | [3,1,4,5] | [3,1,4,5,2] |
| [4,5,3,1] | [4,5,3,1,2] | [1,4,5,2] | [1,4,5,2,3] | [1,4,5,3] | | [2,3,1,4] | [2,3,1,4,5] | [1,2,4,5] | [1,2,4,5,3] |
| | | | | [2,4,5,3] | [2,4,5,3,1] | [5,3,1,4] | | | |
| [4,5,2,3,1] | | [3,1,4,5,2] | | [1,4,5,2,3] | | [5,2,3,1,4] | | [3,1,2,4,5] | |
| [2,4,5,3,1] | | [4,5,3,1,2] | | [1,2,4,5,3] | | [5,3,1,2,4] | | [2,3,1,4,5] | |

- For node $k$, $1 \le k \le N$,
  1. Send **cp** $= \|A_k\|$ as a count update to node 0.
  2. For $j \in \{1, \cdots, N\}$ and $A_k^j \neq 0$, send path token $[k, j]$ to node $j$.
  3. Wait and receive messages or path tokens from other nodes.
  4. If a stop message is received from node 0, then **stop**.
  5. If a path token $[s_1, s_2, \cdots, s_m = k]$ is received, record the path, increment $\mathrm{NP}_{s_1 k}$ by 1, and let **cp** $:= -1$.
  6. For $j \in \{1 \le i \le N : A_k^i \neq 0, i \notin \{s_1, \cdots, s_m\}\}$, send $[s_1, \cdots, s_m = k, j]$ to node $j$, and let **cp** $:=$ **cp** $+ 1$.
  7. Send **cp** to node 0 if **cp** $\neq 0$.
  8. Go to step 3.

It is easy to show that PG actually generates all possible paths in $D$ as follows. Let $P$ be any path in $D$ which starts at $s_1$, goes through $s_2, s_3, \cdots, s_{m-1}$, and ends at $s_m$. In PG, $s_1$ initializes the path token $[s_1, s_2]$ and sends it to $s_2$. Node $s_2$ will then send the path token $[s_1, s_2, s_3]$ to $s_3$, and so on. Eventually, there will be a path token $[s_1, \cdots, s_m]$ received by $s_m$ that represents exactly the path $P$.

As an example, a complete record of running PG for the example graph $D_1$ is given in Table I where the path tokens received and sent by each node are shown.

*Operation Rearrangement*

Conditioning is necessary whenever there are common edges among the paths for a given pair of modules $(v_i, v_j)$. So, the primary step in OR is to choose a minimum number of common edges to be conditioned upon so that the rest can be rearranged as operands of join and meet operations. Since the same algorithm applies to every source-destination pair and algorithms for different pairs do not need to communicate, they can be executed in parallel.

The output from OR is the order of operations to be performed. A prefix polish notation is used to represent the order of operations. The three operations defined earlier are denoted as

- $/Z$: conditioning on $Z$
- $+_n$: join with $n$ operands
- $\&_n$: meet with $n$ operands.

For convenience, let $B_m$ be the direct error propagation time for the $m$th edge.[1] If $B_m$ is the operand of a conditioning operation, then the sample value of $B_m$ after the conditioning is denoted by $b_m$. For example, the following two expressions have the same meaning:

$$X = /B_1 \,\&_2\, B_2 \,+_2\, b_1 B_3,$$

$$f_X(t) = \int_{-\infty}^{\infty} f_{\min(B_2, b_1 + B_3)}(t) \cdot f_{B_1}(b_1)\, db_1.$$

A constant operand (e.g., $b_1$) of the join or meet operation will be treated as a discrete random variable with only one point mass.

During the execution of PG, the propagation paths for each pair of modules are recorded in two matrices: 1) the *node traverse matrix* $\boldsymbol{VT}$ and 2) the *edge traverse matrix* $\boldsymbol{ET}$. The columns of $\boldsymbol{VT}$ correspond to nodes, whereas the columns of $\boldsymbol{ET}$ correspond to edges numbered as in $\boldsymbol{A}$. The rows of both matrices correspond to propagation paths so that there will be $m$ rows in both matrices if the total number of propagation paths is $m$ for a given pair of modules. The order of rows (paths) is irrelevant for OR as long as the same rows in both matrices correspond to the same path. In the discussion that follows, the path corresponding to the $k$th row will be called the $k$th path or path $k$. The entries of $\boldsymbol{VT}$ and $\boldsymbol{ET}$ are such that

[1] $B_m$ is the same as $B_{ij}$ defined earlier if $A_i^j = m \neq 0$, where $A$ is the adjacency matrix.

if $e_{ij}$ belongs to the $k$th propagation path and $A_i^j = m > 0$, then $VT_k^i = j$ and $ET_k^m = 1$. In other words, there is a nonnull edge in the $k$th path from $v_i$ to $v_j$ where $j = VT_k^i \neq 0$. For example, the node and edge traverse matrices for $(v_1, v_3)$ of $D_1$ are

$$VT = \begin{pmatrix} 2 & 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 5 & 3 \\ 2 & 4 & 0 & 5 & 3 \\ 4 & 3 & 0 & 5 & 2 \end{pmatrix},$$

$$ET = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

The fourth path for the pair $(v_1, v_3)$ is $[v_1, v_4, v_5, v_2, v_3]$ which includes edges $e_2, e_3, e_6$, and $e_7$.

A few new terms are necessary to explain OR. A *partial path* is a subset of all the edges in a propagation path, whereas a *subpath* is a segment of the propagation path. Clearly, for a given propagation path, a subpath is a partial path, but the converse is not always true. A *working group (subgroup)* for a given pair of nodes is defined as a collection of partial paths of all propagation paths between the two nodes and is evaluated as

$$\min\ (B_{i_1} + \cdots + B_{i_n}, \cdots, B_{j_1} + \cdots + B_{j_m})$$

where every entry in min is associated with a partial path of a distinct propagation path. Let $NS$, $ES$, and $PS$ represent the *node status vector, the edge status vector,* and the *path status vector,* respectively, where $NS$ and $ES$ are column vectors and $PS$ is a row vector. A working group can be derived from these status vectors and the node and edge traverse matrices. An edge is in a working group if it belongs to at least one partial path in the group. An edge starts from a node called *source* and ends at a node called *sink*. A node is said to belong to a working group if it is the source or sink of an edge in the group. The node status vector holds a physical meaning only if all partial paths in a working group are subpaths. These vectors for a working group are defined as follows:

$$NS_i = \begin{cases} 0 & \text{if } v_i \text{ is not in the working group} \\ 1 & \text{if } v_i \text{ is in the working group} \\ -1 & \text{if } v_i \text{ is the source node of all subpaths in the working group} \\ -2 & \text{if } v_i \text{ is the destination node of all subpaths in the working group} \end{cases}$$

$$ES_j = \begin{cases} 0 & \text{if the } j\text{th edge is not in the working group} \\ -1 & \text{if the } j\text{th edge is a constant edge in the working group} \\ 1 & \text{if the } j\text{th edge is a variable edge in the working group} \end{cases}$$

$$PS^k = \begin{cases} 0 & \text{if no partial path of the } k\text{th path belongs to the working group} \\ 1 & \text{otherwise.} \end{cases}$$

A variable edge $B_j$ becomes a constant edge $b_j$ following a conditioning operation.

Basically, OR is a recursive algorithm on working groups employing the divide-and-conquer principle. The initial working group is the set of all propagation paths between a given pair of nodes. For example, the status vectors from the initial working group for $(v_1, v_3)$ of $D_1$ are

$$NS^T = [-1 \quad 1 \quad -2 \quad 1 \quad 1],$$

$$ES^T = [\ 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1],$$

$$PS = [\ 1 \quad 1 \quad 1 \quad 1].$$

A working group is said to be *trivial* if the group contains only one edge or only one path or only constant edges. OR first checks whether or not the initial working group is trivial. If the initial working group is not trivial, OR will attempt to divide the working group into several working subgroups so that a join operation may be performed on the resulting subgroups. If such a division is not possible, then OR will condition on one or more edges to make the division possible. The same procedure will be applied recursively to all the subgroups until they all become trivial.

The division to allow for a join operation is possible if 1) all edges in the group are variable edges and there is at least one common intermediate node among the paths in the group, and 2) there is at least one common edge among the paths in the group. For the latter case, each common edge becomes a working subgroup and the original group minus all common edges becomes the final working subgroup. For the former case, the division is based on the following theorem.

*Theorem 1:* Suppose all edges in a group are variable edges. Then the common nodes, which include the common intermediate nodes as well as the source and destination nodes, will be visited by all the paths of the group in the same order. Besides, if any intermediate node $v_j$ that is not common is visited by a path between a pair of common nodes, say $v_i$ and $v_k$, $v_j$ can only be visited by all the other paths of the group in the same order as it was visited by the path.

*Proof:* Let $v_1$ and $v_2$ be the source and the destination, respectively, and $v_i$ and $v_j$ be any two common intermediate nodes. Suppose all paths visit these nodes in the order $v_1 \rightarrow v_i \rightarrow v_j \rightarrow v_2$ except for the path which visits these nodes in the order $v_1 \rightarrow v_j \rightarrow v_i \rightarrow v_2$. Then the group must contain the path $v_1 \rightarrow v_i \rightarrow v_2$ which does not visit $v_j$, and the path $v_1 \rightarrow v_j \rightarrow v_2$ which does not visit $v_i$, since all edges in the group are variable edges. It would be impossible to separate the above two paths from other paths of the group without conditioning on some edges in the group. This contradicts the assumption that $v_i$ and $v_j$ are common nodes. The other assertion of the theorem can be proved similarly. ■

According to this theorem, if all edges in a group are variable edges and there are $K$ common intermediate nodes, the group can be divided into $K + 1$ subgroups each containing all nodes and edges between every pair of adjacent common nodes. Each subgroup inherits all paths from the original group, but some paths in a subgroup may contain the same nodes and edges and, thus, are identical, since these paths traverse the same path segment in this subgroup but different segments in other subgroups. The identical or redundant paths are deleted before applying OR recursively on these subgroups.

If the current working group cannot be divided to allow for a join operation, OR will divide the group to allow for a meet operation even if conditioning on a few edges is required. The division for a meet operation is possible if there exists at least one *dominant edge*[2] which does not intersect any other variable edge in the working group.

*Definition 6:* For a working group, a variable edge $i$ is said to be *dominated* by another variable edge $j$ if for all the paths of the group the inclusion of edge $i$ implies the inclusion of edge $j$.

*Definition 7:* For a working group, a *dominant edge* is a variable edge which is not dominated by any other variable edges, and a *dominant subgroup* associated with a dominant edge consists of the dominant edge and all edges being dominated and all paths containing these edges.

*Definition 8:* For a working group, two variable edges *intersect* if both are included in at least one path of the group and neither of them dominates the other.

*Theorem 2:* If a dominant edge does not intersect other variable edges, then any edge in the associated dominant subgroup will not intersect edges outside the subgroup.

*Proof:* Suppose edge $j$ is dominated by a nonintersecting dominant edge $i$, but intersects edge $k$ which is not dominated by edge $i$. Then edge $k$ should intersect edge $i$ because if a path edge contains $j$ and $k$, then it has to contain edge $i$ too. This contradicts the assumption that edge $i$ is nonintersecting. ■

The division for a meet operation is done as follows: each nonintersecting dominant subgroup forms a working subgroup and the original group minus all nonintersecting dominant subgroups forms the final working subgroup.

If no nonintersecting dominant edges can be found, then some edges have to be conditioned on to create a nonintersecting dominant edge. This is done by inspecting edge *dominance* and *intersection* vectors.

*Definition 9:* The *edge dominance vector,* $\boldsymbol{DOM}$, is a column vector defined as

$$\boldsymbol{DOM}_i = \begin{cases} n & \text{if edge } i \text{ is a dominant edge which dominates } n-1 \text{ other variable edges} \\ 0 & \text{if edge } i \text{ is not a variable edge} \\ -n & \text{if edge } i \text{ is a variable edge which is dominated by } n \text{ other edges} \end{cases}$$

where $n$ is a positive integer.

*Definition 10:* The *edge intersection vector,* $\boldsymbol{INT}$, is a column vector defined as

$$\boldsymbol{INT}_i = \begin{cases} n & \text{if edge } i \text{ is a dominant edge which intersects } n \text{ variable edges} \\ 0 & \text{if edge } i \text{ is not a variable edge} \end{cases}$$

where $n$ is a positive integer.

For example, consider the initial working group for $(v_1, v_3)$ of $D_1$. The $\boldsymbol{DOM}$ and $\boldsymbol{INT}$ vectors are

$$\boldsymbol{DOM}^T = [2 \quad -1 \quad 2 \quad -3 \quad 0 \quad 5 \quad -3 \quad -1],$$
$$\boldsymbol{INT}^T = [3 \quad 2 \quad 3 \quad 0 \quad 0 \quad 2 \quad 0 \quad 2].$$

[2] To be defined below.

If $\boldsymbol{DOM}_k > 0$ and $\boldsymbol{INT}_k = 0$, then edge $k$ is a nonintersecting dominant edge.

The rules of choosing an edge to be conditioned on are the following.

1) If the set

$$C_1 = \{i : 1 \le i \le M, \boldsymbol{DOM}_i > 0, \ \boldsymbol{INT}_i = \max_{1 \le j \le M} \boldsymbol{INT}_j\}$$

has only one element $k$, choose edge $k$.

2) If $\|C_1\| > 1$, choose edge $k$ where $k$ is the first element of the set

$$C_2 = \{i : 1 \le i \le M, i \in C_1, \boldsymbol{DOM}_i = \max_{j \in C_1} \boldsymbol{DOM}_j\}.$$

After a new edge is conditioned on, $\boldsymbol{DOM}$ and $\boldsymbol{INT}$ should be recalculated. In the above example, edge 1 will be conditioned on first, and $\boldsymbol{DOM}$ and $\boldsymbol{INT}$ become

$$\boldsymbol{DOM}^T = [0 \quad -1 \quad 2 \quad -2 \quad 0 \quad 5 \quad -3 \quad -1],$$
$$\boldsymbol{INT}^T = [0 \quad 2 \quad 2 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1].$$

The second pick is then edge 3. This process is repeated until a nonintersecting dominant edge emerges or the working group becomes trivial.

This rule is based on the fact that the total number of intersections among variable edges are maximally reduced by conditioning on a dominant edge $k \in C_1$ to maximize the chance for a nonintersecting dominant edge to emerge. When $\|C_1\| > 1$, the edge dominating the most number of edges is chosen because the probability that one of those dominated edges becomes a nonintersecting dominant edge would be higher.

The output of this algorithm will be stored in an operation buffer *Obuf* which is empty initially. Adding something to *Obuf* is denoted by $\Rightarrow$ *Obuf* in OR.

*Algorithm OR:*

1. If $\|\boldsymbol{ES}\| \neq 1$, then go to step 2 else do the following.
   For $j \in \{1, \cdots, M\}$,
   if $\boldsymbol{ES}_j = 1$ then $B_j \Rightarrow$ *Obuf*,
   if $\boldsymbol{ES}_j = -1$ then $b_j \Rightarrow$ *Obuf*.
2. If $\|\boldsymbol{PS}\| \neq 1$, then go to step 3 else do the following.
   Let $n := \|\boldsymbol{ES}\|$ and $+_n \Rightarrow$ *Obuf*.
   For $i \in \{1, \cdots, NP\}$ and $j \in \{1, \cdots, M\}$,
   if $\boldsymbol{PS}^i \cdot \boldsymbol{ES}_j \cdot \boldsymbol{ET}_i^j = 1$ then $B_i \Rightarrow$ *Obuf*,
   if $\boldsymbol{PS}^i \cdot \boldsymbol{ES}_j \cdot \boldsymbol{ET}_i^j = -1$ then $b_i \Rightarrow$ *Obuf*.
3. Calculate *Var,* the number of distinct variable edges in the working group.
4. If *Var* $\neq 0$, then go to step 5 else do the following.
   Let $n := \|\boldsymbol{PS}\|$ and if $n > 1, \&_n \Rightarrow$ *Obuf*.
   For each path (the $i$th path) in the working group,
   let $C_i := \{j : 1 \le j \le M, \ \boldsymbol{ET}_i^j \cdot \boldsymbol{ES}^j = -1\}, m = \|C_i\|$,
   $+_m \Rightarrow$ *Obuf*, and for all $k \in C_i, b_k \Rightarrow$ *Obuf*.

5. If $Var \neq \|ES\|$, then go to step 6 else
   let $C := \{i{:}1 \leq i \leq M,\ NS_i = 1,\ PS \times VT^i = \|PS\|\}$.
   If $\|C\| = 0$, then go to step 6 else
   let $n := \|C\| + 1$ and $+_n \Rightarrow Obuf$.
   Divide the group into $n$ working subgroups according to Theorem 1.
   Run OR on every resulting subgroup.
6. Let $C := \{i{:}1 \leq i \leq M,\ ES_i \neq 0,\ PS \times ET^i = \|PS\|\}$.
   If $\|C\| = 0$, then go to step 7 else do the following.
   Let $n := \|C\| + 1$ and $+_n \Rightarrow Obuf$.
   For all $i \in C$,
   if $ES_i = 1$ then $B_i \Rightarrow Obuf$,
   if $ES_i = -1$ then $b_i \Rightarrow Obuf$,
   let $ES_i := 0$.
7. Calculate $DOM$ and $INT$.
   Let $C := \{i{:}1 \leq i \leq M,\ DOM_i > 0,\ INT_i = 0\}$.
   Let $D := \{j{:}1 \leq j \leq NP,\ PS^j = 1,\ ES_i \cdot ET_j^i \leq 0\ \forall i\}$.
   If $\|C\| + \|D\| = 0$, go to step 8.
   Let $n := \|C\| + \|D\| + 1$ and $\&_n \Rightarrow Obuf$.
   Divide the group into $n$ working subgroups according to Theorem 2.
   Run OR on every resulting subgroup.
8. Let $C_1 := \{i{:}1 \leq i \leq M,\ DOM_i > 0,\ INT_i = \max_{1 \leq j \leq M} INT_j\}$.
   Let $C_2 := \{i{:}1 \leq i \leq M, i \in C_1,\ DOM_i = \max_{j \in C_1} DOM_j\}$.
   If $k$ is the first element of $C_2, /B_k \Rightarrow Obuf$ and let $ES_k = -1$.
   Go to step 7.

As an example, the result of applying OR on the pair $(v_1, v_3)$ of $D_1$ is

$$\begin{aligned} X_{13} &= \min\{B_1 + B_3, B_2 + B_6 + B_8, B_1 + B_4 + B_6 \\ &\quad + B_8, B_2 + B_3 + B_6 + B_7\} \\ &= /B_1/B_3 \&_2 +_2 b_1 b_2 +_2 B_6/B_2 \&_2 +_3 b_2 b_3 B_7 \\ &\quad +_2 B_8 \&_2 b_2 +_2 b_1 B_4. \end{aligned}$$

## IV. Measurement of Propagation Times on FTMP

In this section, we will describe the measurement and analysis of our error propagation model for an experimental system, the fault-tolerant multiprocessor (FTMP) at the NASA Airlab [8], [10]–[14]. The purpose of performing such an experiment is twofold. First, we want to show the feasibility of experimentally determining the key parameters of the model, i.e., the distributions of direct propagation times. Second, it would be interesting and important to know the type of distribution that the direct propagation time follows in a real system. If the distribution is exponential or near exponential, such as Weibull distribution, a Markov model would be better suited to model the error propagation. Otherwise, general distributions should be used in the propagation model. The choice of FTMP is based on the availability as well as its facilities of injecting faults and collecting the subsequent data. Another advantage of FTMP is that an error detection circuitry is easier to build since it is a tightly synchronized triple modular system.

### *FTMP System Organization*

FTMP consists of ten identical line replaceable units (LRU's) each of which contains a processor module with local cache memory, a 16K system memory module, a 1553 I/O port, two bus guardian units (BGU's), a clock generator, and a power subsystem module. Nine of the ten processor modules are grouped into three triads with the remaining one being a spare. The system memory modules are also grouped into triads. The communications between processors and system memory modules are accomplished via three sets of system buses: a polling bus ($P$ bus) for resolving bus contention, a data transmit bus ($T$ bus), and a data receive bus ($R$ bus). Each set of buses includes five bus lines for redundancy, but only three of them are activated for the triad's communication.

All three members of a triad are running in tight synchrony, which is achieved by another set of system buses, the clock bus ($C$ bus). Four of the ten clocks are phase-locked together through the clock bus, while the other six clocks simply lock into one of these four clocks.

The access to system buses in each LRU is controlled by BGU's which operate in dyad. Associated with each system bus in an LRU is a voter. Whenever a module in the LRU reads data from system buses (i.e., $T$ bus, $R$ bus, or $P$ bus), the voter will take a majority vote on the data received from the three active bus lines. If a disagreement is found during the voting, it will be recorded on error latches for later fault identification. The complete hardware description of FTMP can be found in [11].

From the software point of view, FTMP can be regarded as a three-processor system with a shared memory of 48K. Since FTMP is designed mainly for real-time control applications, every task (or process) runs periodically in one of three rate groups: $R1$ (3.125 Hz), $R3$ (12.5 Hz), and $R4$ (25 Hz). Critical tasks, such as task dispatching, are running in the $R4$ group, while normal application tasks are running in the $R3$ group. The *system configuration controller* (SCC), which is an executive program handling fault detection, identification, and system reconfiguration, is dispatched in the $R1$ group to minimize the performance degradation. Currently, FTMP includes only two application tasks: an autopilot program and a display program. See [12] for more details on FTMP software.

To validate FTMP's fault-tolerance capability, a hardware fault injection system has been built which is controlled by a host computer VAX-11/750. This system uses injection implants inserted between any chips on LRU3 and their sockets to control the electrical connections between the pins and the circuit board. A special circuit board extender is also needed to make space for the injection implants. Three types of permanent faults at the pin level can be injected, i.e., *inverted signal, stuck-at-1,* and *stuck-at-0.* To operate this injection system, a customized version of the SCC program, called

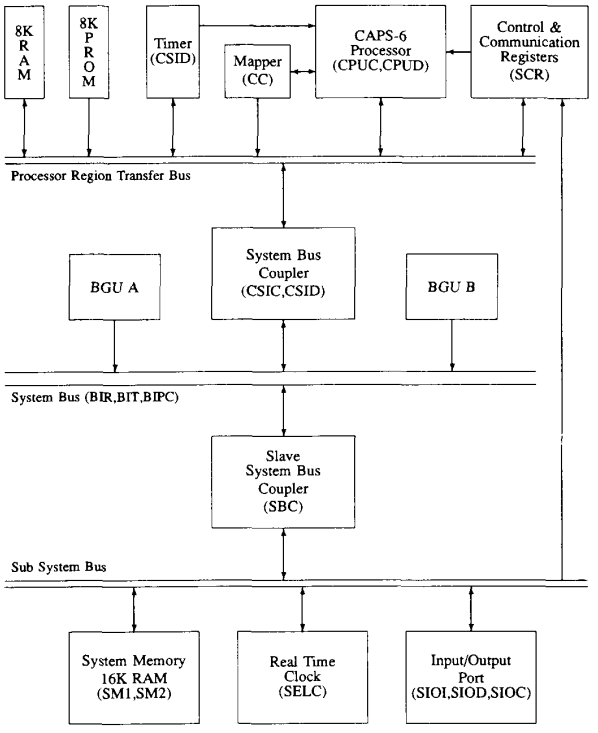


Fig. 4. Functional block diagram of an LRU.

FSCC, must be loaded which will receive queries from the host before every injection, issue proper reconfiguration commands to ensure that LRU3 is in an active triad, and then send a ready signal back to the host. With this protocol between the host and FTMP, one can inject faults repetitively under program control. The complete description of the fault injection system is in [13].

### *Experimental Model*

The organization of hardware components in an LRU is shown in Fig. 4. The abbreviation(s) shown in the parenthesis of each block indicates the circuit board(s) implementing that functional block. (For a more detailed description of each functional block, see [11].) By treating each functional block and each bus in Fig. 4 as a module, the error propagation model in Section II can be constructed to study error propagation within an LRU. Since FTMP performs a TMR voting on data going out of the system bus module, it is assumed that no errors can propagate through the system bus. Thus, we can isolate the processor region of an LRU which includes the following modules: 8K RAM, 8K PROM, timer, mapper, CAPS-6 processor, control and communication registers (CCR), processor region transfer bus (PRTB), system bus coupler (BCO), and system bus (SB). It is also assumed that no errors will come from the system bus or the subsystem bus. The resulting model is shown in Fig. 4.

In Fig. 5, there are nine modules, 17 directed edges, and hence 72 error propagation functions. To determine these functions, all 17 direct propagation times have to be measured and fed into the computation described in Section II.

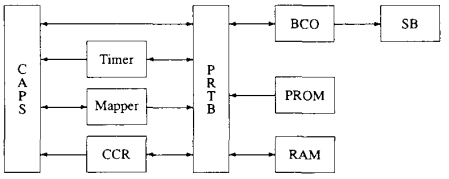


Fig. 5. FTMP's experimental error propagation model.

In this experiment, only one direct propagation time will be measured to show the feasibility of performing such an experiment. The one we chose is for the edge from module BCO to module SB. The other direct propagation times can be measured similarly. We assume that a module in Fig. 5 gets contaminated whenever it receives an erroneous input signal. Thus, the direct error propagation time from BCO to SB is equivalent to the time for errors to propagate from any input of BCO to its output to SB.

The experiment is set up as follows. A circuit to detect errors on the input and output of BCO is custom built. The design of this detection circuit is simplified by the triad organization of FTMP and the fact that each triad is tightly synchronized. First, the signal lines which constitute the input and output of BCO are identified. Then, the input and output lines in LRU0 and LRU3 are tapped out to the detection circuit for comparison. If LRU0 and LRU3 are in the same triad, an error is detected whenever their line values disagree. Using LRU0 as a reference, faults are injected into LRU3 and error propagation times from the input to the output of BCO are measured. In order to make a valid comparison, LRU0 and LRU3 must be in the same triad during fault injection. So, the FSCC program is modified to group LRU0 and LRU3 into the same triad before each injection. Experiments are performed under the normal FTMP workload.

### *Data Analysis*

Faults are injected into the circuit board CPUD which is a part of the data section of the CAPS-6 processor and is inside the module CAPS in Fig. 5. Selection of IC chips and pins for fault injection is arbitrarily made. The chips selected are *U2*, *U7*, and *U32*. *U2* (54LS253) is a dual 4-to-1 multiplexer and *U7* (54LS51) is a dual AND-OR-INVERT gate; both are among the chips for selection of carry-in signals for ALU. *U32* (54LS257) is a quad 2-to-1 multiplexer which can select a right-shifted output from ALU. Faults were injected at pin 2, 6, and 7 of *U2*; pin 3, 4, and 6 of *U7*; and pin 4 of *U32*. For each selected pin, three types of faults are injected: stuck-at-1, stuck-at-0, and inverted signal. Each type of fault is injected 500 times, so there would be 1500 faults injected for each selected pin. The sample density functions for each pin and each type of fault are plotted for the range of 0–1 ms in an interval of 40 $\mu$s. However, only the plots for inverted faults are shown in Figs. 6–12, since the plots for the other two types of faults are very similar.

Now we want to see if our data can fit the Weibull distribution which has been widely used for reliability modeling and life testing. The Weibull distribution has two parameters: $\alpha$ (the shape parameter) and $\lambda$ (the scale parameter). The prob-

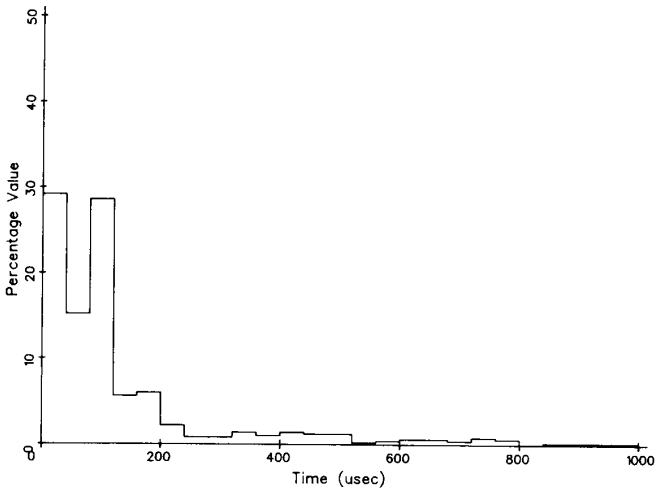


Fig. 6. Error propagation time distribution for CPUD $U2$ $P2$.

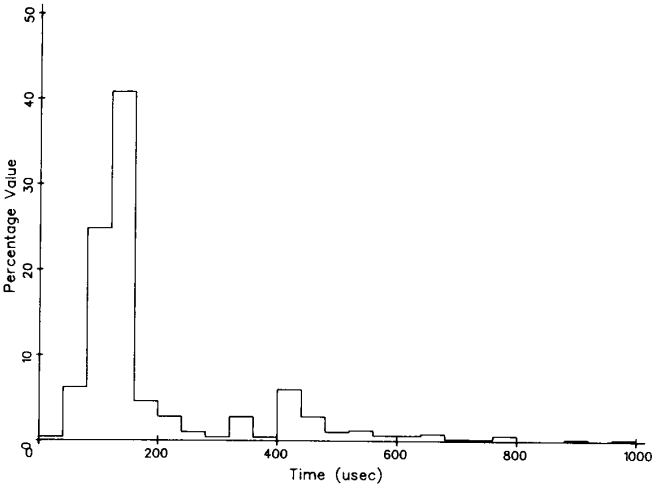


Fig. 7. Error propagation time distribution for CPUD $U2$ $P6$.

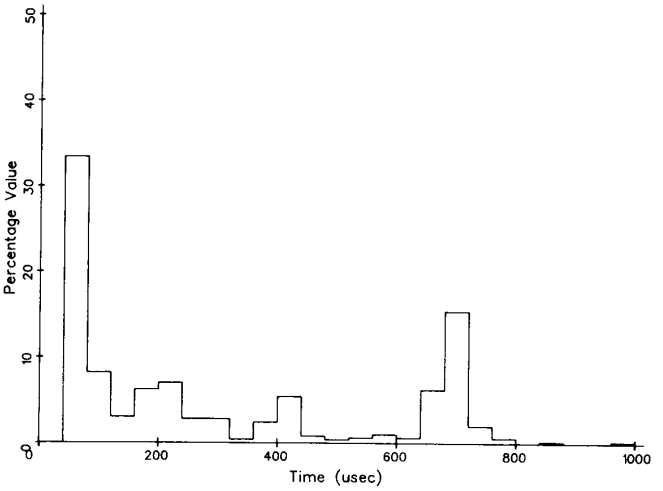


Fig. 8. Error propagation time distribution for CPUD $U2$ $P7$.

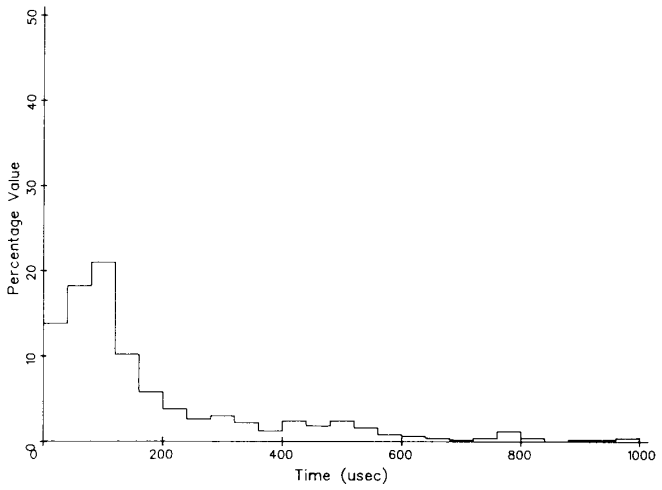


Fig. 9. Error propagation time distribution for CPUD $U32$ $P4$.

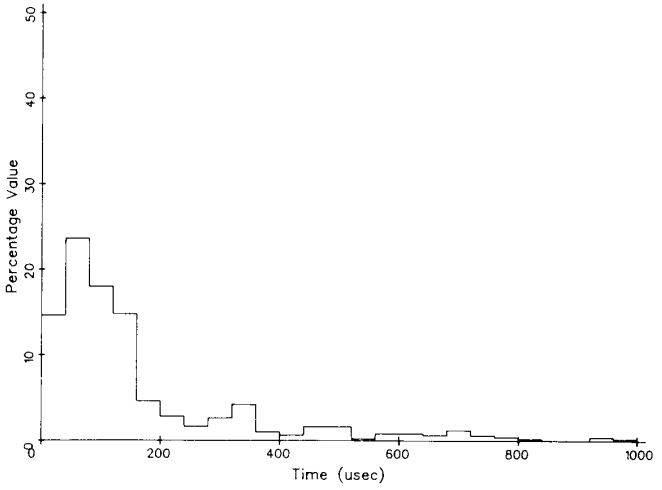


Fig. 10. Error propagation time distribution for CPUD $U7$ $P3$.

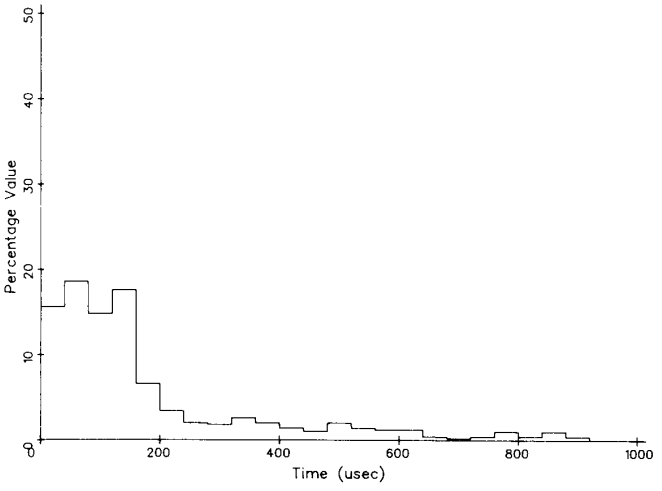


Fig. 11. Error propagation time distribution for CPUD $U7$ $P4$.

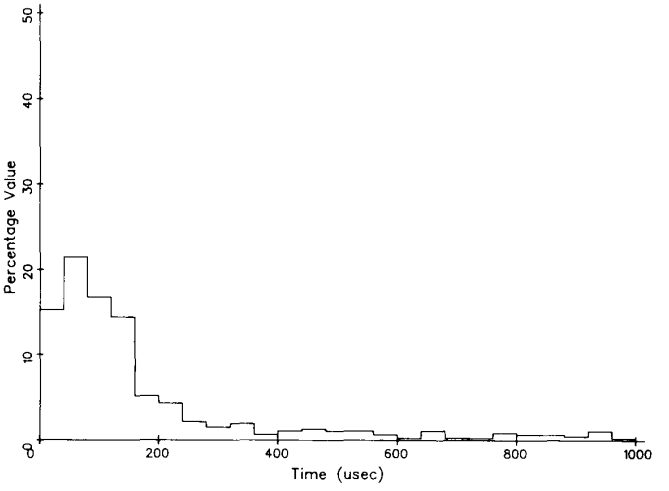


Fig. 12. Error propagation time distribution for CPUD *U7 P6*.

ability density function of the Weibull distribution is

$$\alpha\lambda(\lambda t)^{\alpha-1}e^{-(\lambda t)^{\alpha}}.$$

The exponential distribution is a special case of the Weibull distribution with $\alpha = 1$.

Various methods have been proposed to estimate the shape and scale parameters of the Weibull distribution [15]–[18]. We adopt the maximum likelihood estimator (MLE) method developed in [15]. The MLE shape estimator $\hat{\alpha}$ must satisfy the equation

$$\frac{n}{\hat{\alpha}} + \sum \ln x_i = n\frac{\sum x_i^{\hat{\alpha}} \ln x_i}{\sum x_i^{\hat{\alpha}}}$$

where $x_i, i = 1, \cdots, n$, represent the sample data. This equation can be solved by the Newton–Raphson method as shown in [15]. Once $\hat{\alpha}$ is known, the MLE scale estimator $\hat{\lambda}$ can be calculated as

$$\hat{\lambda} = \left(\frac{n}{\sum x_i^{\hat{\alpha}}}\right)^{1/\hat{\alpha}}.$$

The chi-square goodness-of-fit test is used after the parameter estimation to determine how well our data fit the Weibull distribution [19], [20]. In this test, the positive real line is divided into $m$ intervals, $I_1, \cdots, I_m$. Let $O_i$ be the number of data points appearing in $I_i$, and let $E_i$ be the expected number of data points in $I_i$ computed according to the hypothesized Weibull distribution. The statistic $C = \sum_{i=1}^{m} (O_i - E_i)^2/E_i$ has a chi-square distribution with the degree of freedom $m-3$, since two parameters have been estimated from the same set of data. For the accuracy of the test, the intervals have to be such that $E_i \geq 5$ for all $i$. Given a significance level $\beta$, the threshold value $\chi_\beta^2$ can be determined from a chi-square distribution table. If $C \geq \chi_\beta^2$, we conclude with a significant level $\beta$ that the data do not fit the Weibull distribution.

The results from the parameter estimation and goodness-of-fit test are tabulated in Table II with a significance level 0.05. The observation intervals are basically the same as the intervals used to plot density functions, i.e., 40 $\mu$s per interval. However, to meet the requirement that $E_i \geq 5$ for all $i$, several intervals may be merged into a single interval; this is the reason why the degree of freedom (dof) may differ as shown in Table II.

TABLE II
TEST RESULT OF FITTING THE WEIBULL DISTRIBUTION

| Data Set | Shape | Scale | dof | $C$ | $\chi^2_{05}$ |
|---|---|---|---|---|---|
| U2 P2 invert | 0.75 | 0.000149 | 8 | 191 | 2.7 |
| U2 P2 s-a-0 | 0.71 | 0.000177 | 8 | 138 | 2.7 |
| U2 P2 s-a-1 | 0.72 | 0.000146 | 9 | 101 | 3.3 |
| U2 P6 invert | 0.98 | 0.000285 | 8 | 883 | 2.7 |
| U2 P6 s-a-0 | 0.99 | 0.000279 | 7 | 1121 | 2.2 |
| U2 P6 s-a-1 | 0.92 | 0.000298 | 6 | 1187 | 1.6 |
| U2 P7 invert | 0.11 | 0.000322 | 10 | 856 | 3.9 |
| U2 P7 s-a-0 | 0.11 | 0.000318 | 9 | 250 | 3.3 |
| U2 P7 s-a-1 | 0.11 | 0.000358 | 10 | 1099 | 3.9 |
| U32 P4 invert | 0.71 | 0.000273 | 11 | 162 | 4.6 |
| U32 P4 s-a-0 | 0.71 | 0.000245 | 10 | 176 | 3.9 |
| U32 P4 s-a-1 | 0.73 | 0.000286 | 12 | 190 | 5.2 |
| U7 P3 invert | 0.81 | 0.000210 | 9 | 146 | 3.3 |
| U7 P3 s-a-0 | 0.73 | 0.000234 | 11 | 195 | 4.6 |
| U7 P3 s-a-1 | 0.81 | 0.000230 | 13 | 109 | 5.9 |
| U7 P4 invert | 0.89 | 0.000215 | 11 | 96 | 4.6 |
| U7 P4 s-a-0 | 0.70 | 0.000232 | 11 | 120 | 4.6 |
| U7 P4 s-a-1 | 0.74 | 0.000209 | 11 | 151 | 4.6 |
| U7 P6 invert | 0.74 | 0.000241 | 10 | 138 | 3.9 |
| U7 P6 s-a-0 | 0.76 | 0.000237 | 11 | 139 | 4.6 |
| U7 P6 s-a-1 | 0.82 | 0.000241 | 12 | 124 | 5.2 |

Clearly, none of the data sets fit the Weibull distribution. For some data sets, their density functions show more than one major peak, suggesting that the error propagation time will most likely be one of several discrete values. This also indicates that the shape of the distributions for error propagation

times are strongly related to a module's functionality. Hence, our conclusion from this experiment and analysis is that 1) it would be unrealistic to assume error propagation times follow any well-known distribution and 2) our error propagation model which assumes a general distribution is justified. (Use of Markov models for error propagation is unrealistic!)

*Remarks:*

- Although it is easier experimentally to measure the direct error propagation times on FTMP, the same principle can be applied to other systems. In our experiment, a good unit and the fault-injected unit are running synchronously so that error detection can be accomplished by directly comparing both units' signals in real time. In some other systems, there may not be hardware or software redundancy. In that case, experiments can be performed using time redundancy, i.e., running the same unit twice under the same workload and then comparing the results from both runs to detect errors. A drawback of this approach is that a large number of intermediate results along with timing information have to be stored for comparison. If a prototype system is not available, this experiment can still be done by simulation as in [7].
- Intuitively, errors propagate due to the communications between modules. So, errors would propagate faster if there are frequent communications between modules. If the workload contains more I/O bound tasks, the error propagation time would be generally shorter.

## V. Conclusion

Although the problem of error propagation in distributed fault-tolerant systems has been mentioned frequently by many researchers, very little on this has been reported in the literature. In this paper, we have developed an error propagation model for multimodule computing systems where the main parameters are the distribution functions of error propagation times. We have also derived two algorithms to systematically and efficiently calculate these functions. Finally, to show that the parameters of our model are obtainable from real systems, we have conducted experiments on FTMP. Statistical analyses of experimental data show that none of the measured error propagation times follow the Weibull distribution. In fact, different faults at different locations exhibit different distributions, justifying the necessity and reality of using general distributions in our model.

The reason that errors can propagate from module to module is the imperfect detection mechanisms implemented on each module. Thus, an error propagation model should be coupled with an error detection model. In this paper, however, we have focused specifically on the modeling of the error propagation in order to characterize the behavior of error propagation. Application of the error propagation model along with an error detection model for fault location, damage assessment, and error recovery is a matter of our future inquiry.

## References


[1] T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[2] J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed algorithms to compute tests to detect between failures in logic circuits," *IEEE Trans. Electron. Comput.*, vol. EC-16, pp. 567–580, Oct. 1967.

[3] J. E. Smith and G. Metze, "The design of totally self-checking combinatorial circuits," in *Dig. FTCS-7,* June 1977.

[4] M. Nicolaidis and B. Courtois, "Design of self-checking systems based on analytical fault hypotheses," Res. Rep. RR-353, IMAG, Mar. 1983.

[5] T. Nanya and T. Kawamura, "Error secure/propagation concept and its application to the design of strongly fault secure processors," in *Dig. FTCS-15,* June 1985, pp. 19–21.

[6] K. Zielinski, "Model of error propagation in systems of communicating processes," *Sci. Comput. Program.*, vol. 6, pp. 191–205, Mar. 1986.

[7] D. Lomelino and R. K. Iyer, "Error propagation in a digital avionic processor: A simulated-based study," Contractor Rep. 176501, NASA, 1986.

[8] A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," *Proc. IEEE,* vol. 66, pp. 1221–1240, Oct. 1978.

[9] K. G. Shin and Y.-H. Lee, "Evaluation of error recovery blocks used for cooperating processes," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 692–700, Nov. 1984.

[10] J. H. Lala, "Fault detection, isolation and reconfiguration in ftmp: Methods and experimental results," in *Proc. 5th IEEE/AIAA Digital Avion. Syst. Conf.*, Nov. 1983.

[11] T. B. Smith and J. H. Lala, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer Volume I: Principles of operation," Contractor Rep. 166071, NASA, May 1983.

[12] ——, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer Volume II: FTMP software," Contractor Rep. 166072, NASA, May 1983.

[13] ——, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer Volume III: FTMP test and evaluation," Contractor Rep. 166073, NASA, May 1983.

[14] ——, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer Volume IV: FTMP executive summary," Contractor Rep. 172286, NASA, February 1984.

[15] D. R. Thomas, L. J. Bain, and C. E. Antle, "Inferences on the parameters of the Weibull distribution," *Technometrics,* vol. 11, pp. 445–460, Aug. 1969.

[16] L. J. Bain and C. E. Antle, "Estimation of parameters in the Weibull distribution," *Technometrics,* vol. 9, pp. 621–627, Nov. 1967.

[17] A. C. Cohen, "Maximum likelihood estimation in the Weibull distribution based on complete and on censored samples," *Technometrics,* vol. 7, pp. 579–588, Nov. 1965.

[18] M. V. Menon, "Estimation of the shape and scale parameters of the Weibull distribution," *Technometrics,* vol. 5, pp. 175–182, May 1963.

[19] S. Dowdy and S. Wearden, *Statistics for Research.* New York: Wiley, 1983.

[20] R. J. Larsen and M. L. Marx, *An Introduction to Mathematical Statistics and its Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1981.



**Kang G. Shin** (S'75-M'78-SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

He is a Professor in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, which he joined in 1982. He has been very active and authored/coauthored over 120 technical papers in the areas of fault-tolerant real-time computing, computer architecture, and robotics and automation. In 1985, he founded the Real-Time Computing Laboratory, where he and his students are currently building a 19-node hexagonal mesh multiprocessor, called HARTS, to validate various architectures and analytic results in the area of distributed real-time computing. From 1970 to 1972, he served in the Korean Army as an ROTC officer and from 1972 to 1974, he was on the research staff of the Korea Institute of Science and Technology,

Seoul, Korea, working on the design of VHF/UHF communication systems. From 1978 to 1982, he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a Visiting Scientist at the U.S. Airforce Flight Dynamics Laboratory in Summer 1979 and at Bell Laboratories, Holmdel, NJ, in Summer 1980.

Dr. Shin was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, and the Guest Editor of the 1987 August special issue of IEEE TRANSACTIONS ON COMPUTERS on Real-Time Systems. He is a member of the Association for Computing Machinery, Sigma Xi, and Phi Kappa Phi. In 1987, he received the Outstanding Paper Award from the IEEE TRANSACTIONS ON AUTOMATIC CONTROL for a paper on robust trajectory planning.

**Tein-Hsiang Lin** (S'83) received the B.S. degree from the National Taiwan University, Taipei, Taiwan, in 1980, the M.S. degree from Iowa State University, Ames, in 1984 and the Ph.D. degree from the University of Michigan, Ann Arbor, 1988, all in electrical engineering.

He is currently an Assistant Professor in the Department of Electrical and Computer Engineering, State University of New York at Buffalo. His research interests include multiprocessor and distributed systems, performance evaluation, and fault-tolerant computing.