Software Dependability in the Tandem GUARDIAN System

Inhwan Lee, *Member, IEEE,* and Ravishankar K. Iyer, *Fellow, IEEE*

***Abstract*—Based on extensive field failure data for Tandem's GUARDIAN operating system, this paper discusses evaluation of the dependability of operational software. Software faults considered are major defects that result in processor failures and invoke backup processes to take over. The paper categorizes the underlying causes of software failures and evaluates the effectiveness of the process pair technique in tolerating software faults. A model to describe the impact of software faults on the reliability of an overall system is proposed. The model is used to evaluate the significance of key factors that determine software dependability and to identify areas for improvement.**

An analysis of the data shows that about 77% of processor failures that are initially considered due to software are confirmed as software problems. The analysis shows that the use of process pairs to provide checkpointing and restart (originally intended for tolerating hardware faults) allows the system to tolerate about 75% of reported software faults that result in processor failures. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events) being different from the original execution, is a major reason for the measured software fault tolerance. Over two-thirds (72%) of measured software failures are recurrences of previously reported faults. Modeling, based on the data, shows that, in addition to reducing the number of software faults, software dependability can be enhanced by reducing the recurrence rate.

***Index Terms*—Measurement, fault categorization, software fault tolerance, recurrence, software reliability, operational phase, Tandem GUARDIAN System.**

I. Introduction

THIS paper discusses evaluation of the dependability of operational software based on measurements taken from the Tandem GUARDIAN operating system. The Tandem GUARDIAN system is a commercial fault-tolerant system built for on-line transaction processing and decision support. The GUARDIAN operating system is a message-based operating system that runs on a Tandem machine. Many studies have sought to improve the software development environment by using the failure data collected during the development phase [1], [2], [3]. The dependability issues for operational software are typically very different from those for software under development, due to differences in the operational environment and software maturity. Also, the dependability of operational software needs to be investigated in the context of the overall system.

A study of the dependability of operational software based on real measurements requires, in addition to instrumentation and data collection, an understanding of the system architecture, hardware, and software. It also requires an understanding of the development, service, and operational environments. Typically, measurement-based studies attempt to answer several questions: What are the key failure modes and their significance, how well do specific fault-tolerance techniques work, and what is a realistic behavior model for the software and its associated parameters? This paper presents results based on field failure data collected from the Tandem GUARDIAN operating system. The data cover a period extending over four months. The issues addressed include software fault categorization, an evaluation of the **software** fault tolerance of process pairs (a key hardware fault-tolerance technique used in Tandem systems), and evaluation of the impact of software faults on the overall system.

The next section discusses related research. Section III introduces the Tandem GUARDIAN system and the measurements made. Section IV investigates the underlying causes (faults) that resulted in the observed software failures and categorizes the identified faults. The significance of failure recurrence is also discussed. Section V evaluates the software fault tolerance of process pairs. The reasons for achieving this software fault tolerance are investigated. This evaluation is important because, although process pairs are specific to Tandem systems, they are an implementation of the general approach of checkpointing and restart. Section VI builds a model that describes the impact of faults in the GUARDIAN operating system on the reliability of an overall Tandem system. A sensitivity analysis is conducted to evaluate the significance of the factors that determine software dependability and to identify areas for improvement. Section VII summarizes the major conclusions of this study.

II. Related Research

Software errors in the development phase have been extensively studied. Software error data collected from the DOS/VS operating system during the testing phase were analyzed in [4]. A wide-ranging analysis of software error data collected during the development phase was reported in [5]. An error

Manuscript received Aug. 1994; revised Feb. 1995.

This study was conducted while Inhwan Lee was with the Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. The authors may be reached by e-mail at: lee_inhwan@tandem.com, and iyer@crhc.uiuc.edu.

This work was supported by the National Aeronautics and Space Administration under grant NAG-1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), by Tandem Computers Incorporated, by the Office of Naval Research under Grant N00014-91-J-1116, and by the Advanced Research Projects Agency under grant DABT63-94-C-0045.

The findings, opinions, and recommendations expressed herein are those of the authors and do not necessarily reflect the position or policy of the United States Government and no official endorsement should be inferred.

IEEECS Log Number S95008.

analysis technique was used to evaluate software development methodologies in [6]. Relationships between the frequency and distribution of errors during software development, maintenance of the developed software, and a variety of environmental factors were analyzed in [7]. The orthogonal defect classification, the use of observed software defects to provide feedback on the development process, was proposed in [8]. These studies mainly attempt to fine-tune the software development environment based on error analysis.

Software reliability modeling has also been studied extensively, and many models have been proposed [1], [2], [3]. For the most part, these models attempt to estimate the reliability of software by analyzing the failure history of software during the development phase, verification efforts, and operational profile.

Measurement-based analysis of operational software dependability has also evolved over the past 15 years. An early study proposed a workload-dependent probabilistic model for predicting software errors based on measurements from a DEC system [9]. The effect of workload on operating system reliability was analyzed using the data collected from an IBM 3081 machine running VM/SP [10]. A Markov model to describe the software error and recovery process in a production environment using error logs from the MVS operating system was discussed in [11]. Software defects and their impact on system availability were investigated using data from the IBM MVS system in [12]. In [13], results from a census of Tandem systems were presented. The data showed that software was the major source (62%) of outages in the Tandem system. Dependability and fault tolerance of three operating systems—the Tandem GUARDIAN system, the IBM MVS system, and the VAX VMS system—were analyzed using error logs in [14].

Software failures have also been studied from the software fault-tolerance perspective. Two major approaches for software fault tolerance—recovery blocks and *N*-version programming—were proposed in [15], [16]. Dependability modeling and evaluation of these two approaches were discussed in [17]. The effectiveness of recovery routines in the MVS operating system was evaluated using measurements from an IBM 3081 machine in [18]. Software fault tolerance in the Tandem GUARDIAN operating system was discussed in [19], [20]. Architectural issues for incorporating hardware and software fault tolerance were discussed in [21], [22], [23].

III. Tandem System and Measurements

The Tandem GUARDIAN system is a message-based multiprocessor system built for on-line transaction processing and decision support [20]. A Tandem GUARDIAN system consists of two to 16 processors, dual interprocessor buses, dual-port device controllers, input/output (I/O) devices, multiple I/O buses, and redundant power supplies (Fig. 1). The key software components are processes and messages. With a separate copy of the GUARDIAN operating system running on each processor, these abstractions hide the physical boundaries between processors and systems and provide a uniform environment across a network of Tandem systems.

In the Tandem GUARDIAN system, a critical system function or user application is replicated on two processors as primary and backup processes, i.e., as a process pair. Normally, only the primary process provides service. The primary sends checkpoints to the backup, so that the backup can take over the function when the primary fails. The GUARDIAN system software halts the processor it runs on when it detects nonrecoverable errors. Nonrecoverable errors are a subset of exceptions in privileged system processes. They are detected by the operating system or explicit software checks made by privileged system processes. The designer determines whether a specific exception is nonrecoverable. The "I'm alive" message protocol allows the other processors to detect the halt and to take over the primaries that were running on the halted processor. With multiple processors running process pairs, dual interprocessor buses, dual-port device controllers, multiple I/O buses, disk mirroring, and redundant power supplies, the system can tolerate a single failure in a processor, bus, device controller, disk, or power supply.

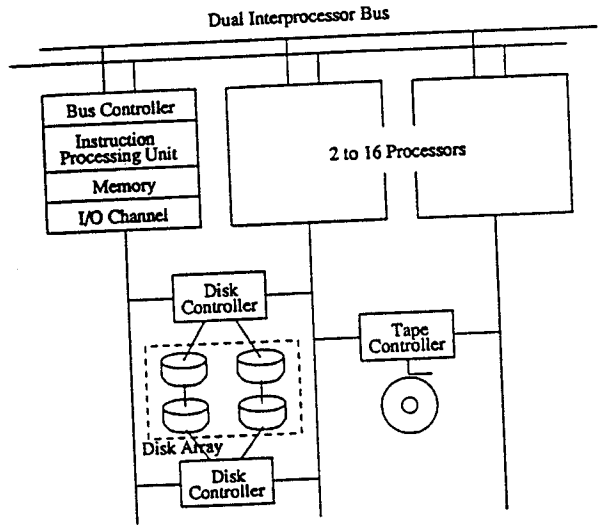


Fig. 1. Tandem GUARDIAN system architecture.

In this paper, a *software fault* is a defect in the measured software system, and a *software failure* is a processor failure due to software. The terms *processor halt* and *processor failure* are used interchangeably. Fig. 2 illustrates the software failure and recovery process in the Tandem GUARDIAN system. When a fault in the system software is exercised, an error (a first error) is generated. Depending on the processor state, this error may disappear or cause additional errors before being detected. The impact of a detected error ranges from a minor cosmetic problem at the user/system interface to a database corruption. A software failure occurs when the system software detects nonrecoverable errors and asserts a processor halt.

Once a software failure occurs, the system attempts to recover using backup processes on other processors. If this recovery is successful, the system can tolerate the software fault. The time it takes for the system to detect a processor halt and for the backup to attain the primary's pre failure state depends

on several factors, such as the priority of the process, processor configuration, and workload. The recovery usually takes about 10 seconds. If a job takeover is not successful or if a backup process faces the same problem after a takeover, a double processor halt occurs. Regardless of whether the recovery is successful, the software fault is identified and a fix is made. A single software fault can cause multiple software failures at a single site or at multiple sites ("Recurrences" in Fig. 2).

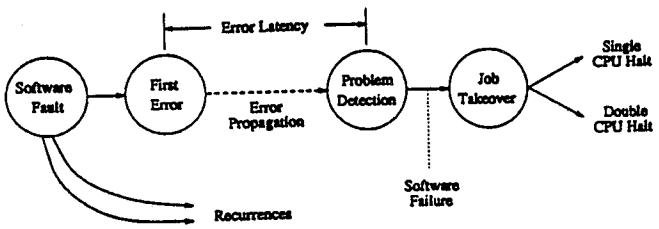


Fig. 2. Software failure and recovery in the Tandem GUARDIAN system.

The human-generated software failure reports used in this study were extracted from the Tandem Product Report (TPR) database, a component of the Tandem Product Reporting System (PRS). A TPR is used to report all problems, questions, and requests for enhancements by users or Tandem employees concerning any Tandem product. A TPR consists of a header and a body. The header provides fixed fields for information such as the date, problem type, urgency, user and system identifications, and a brief problem description. The body of a TPR is a textual description of all actions taken by Tandem analysts in diagnosing the problem. If a TPR reports a software failure, the body also includes the log of the memory dump analyses performed by Tandem analysts. The information in a TPR clearly indicates whether the incident was a software failure, whether the underlying fault was fixed, and whether the TPR shared the underlying fault with other TPRs. Two-hundred TPRs for the GUARDIAN operating system that cover a period extending over four months in 1991 were used for this study.

IV. Fault Categorization

Several studies have performed fault categorization based on faults identified during the development phase [4], [5], [7]. Software fault profiles in operational software can be quite different, due to differences in the operational environment and software maturity. We studied the underlying causes of 200 TPRs that reported processor failures seemingly due to faults in the Tandem system software [24].

Fig. 3 shows a breakdown of the TPRs into three categories (Software "Cause Identified," Software "Cause Unidentified," and "Non-Software Problem"). Determining whether a failure was caused by software faults is not straightforward, due partly to system complexity and partly to close interactions between the software and the hardware in the system. The only reliable approach is to declare an incident to be a software problem only after analysts have located a fault in the software, reproduced the incident, and designed and tested a software fix.

Software causes were identified for 153 TPRs ("Cause Identified"). If a TPR identified a fault in the software and resulted in a software fix, the incident was counted as a software problem, even if it was initially triggered by a non-software cause (e.g., a hardware fault). In 26 TPRs ("Cause Unidentified"), analysts believed that the underlying problems were software faults, but they had not yet located the faults. We use the term *unidentified failures* to refer to these cases. The rest of the TPRs ("Non-Software Problem") were due mainly to hardware faults (e.g., a failure in power supply) or operational faults (e.g., incorrectly specifying hardware specifications in a system table). Note that 76.5% of the TPRs that were initially classified as software problems were confirmed as software problems, 13% of them were probably software problems, and the rest (10.5%) were non-software problems. The 179 TPRs ("Cause Identified" and "Cause Unidentified") formed the basis of our analysis. Fig. 3 specifies which groups of the TPRs were used to build the subsequent tables.

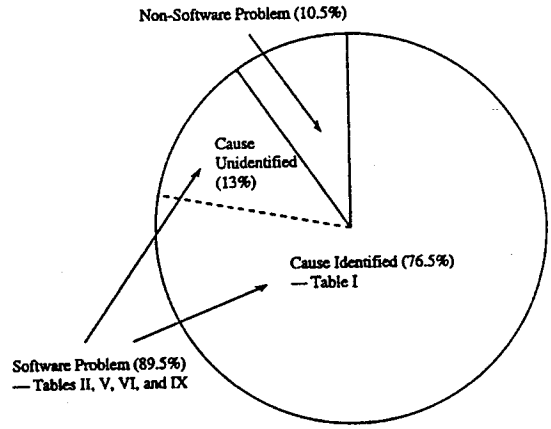


Fig. 3. Problem types.

Table I shows the fault categories we selected in conjunction with analysts. The table also shows the number of unique faults and the number of TPRs associated with each category. For example, the "Data fault" category contained 12 unique faults, and these faults caused 21 TPRs. Note that a single fault may recur and generate multiple TPRs, because many users run the same software. The 153 TPRs whose software causes were identified were due to 100 unique faults.

A software failure caused by a newly found fault is referred to as a *first occurrence*; a software failure caused by a previously reported fault is referred to as a *recurrence*. Recurrences exist for several reasons. First, designing and testing a fix of a problem can take a significant amount of time. In the meantime, recurrences can occur at the same site or at other sites. Second, the installation of a fix sometimes requires a planned outage, which may force users to postpone the installation and thus cause recurrences. Third, a purported fix can fail. Finally and probably most importantly, users who did not experience problems due to a certain fault often hesitate to install an available fix for fear that doing so will cause new problems.

TABLE I
SOFTWARE FAULT CATEGORIZATION

Fault Category	#Faults	#TPRs
Incorrect computation	3	3
Data fault	12	21
Data definition fault	3	7
Missing operation:	20	27
Uninitialized pointers	(6)	(7)
Uninitialized nonpointer variables	(4)	(6)
Not updating data structures on the occurrence of certain events	(6)	(9)
Not telling other processes about the occurrence of certain events	(4)	(5)
Side effect of code update	4	5
Unexpected situation:	29	46
Race/timing problem	(14)	(18)
Errors with no defined error-handling procedures	(4)	(8)
Incorrect parameters or invalid calls from user processes	(3)	(7)
Not providing routines to handle legitimate but rare operational scenarios	(8)	(13)
Microcode defect	4	8
Others (cause does not fit any of the above class)	10	12
Unable to classify due to insufficient information	15	24
All	100	153

Most of the categories in Table I are self-explanatory. "Incorrect computation" refers to an arithmetic overflow or the use of an incorrect arithmetic function (e.g., use of a signed arithmetic function instead of an unsigned one). "Data fault" refers to the use of an incorrect constant or variable. "Data definition fault" refers to a fault in declaring data or in defining a data structure. "Missing operation" refers to an omission of lines of source code. "Side effect of code update" occurs when not all dependencies between software modules were considered when updating the software. "Unexpected situation" refers to cases in which software designers did not anticipate a legitimate operational scenario, and the software did not handle the situation correctly. In the 24 TPRs we were "Unable to classify due to insufficient information," analysts did not provide detailed information about the nature of the underlying faults. "Missing operation" and "Unexpected situation" were the most common types of software faults in the measured software system. Additional code inspection and testing efforts can be used to identify such faults.

Out of the 100 software faults observed during the measured time window, 57 faults were diagnosed before the time window (i.e., were recurrences) and 43 were newly identified during the time window (i.e., were first occurrences). In other words, over two-thirds of the TPRs (72%; 110 out of 153) reported recurrences. When one considers that a single TPR may list a rapid succession of failures, which are likely to be caused by the same fault, the actual percentage of recurrences may be higher.

Recurrences are not unique to Tandem systems. Similar cases have been reported in IBM [25] and AT&T systems [26]. In environments where many users run (different versions of) the same software, the number of identified faults is not the only factor determining software dependability. Recurrences can seriously degrade software dependability in the field. In [25], a preventive software service policy that takes both the number of recurrences and the service cost into account was discussed. An approach for automatically diagnosing recurrences based on symptoms was proposed in [27]. The issue of recurrence is discussed further in Section VI.

V. SOFTWARE FAULT TOLERANCE DUE TO PROCESS PAIRS

In [13], [19], it was observed that process pairs allow the Tandem GUARDIAN system to tolerate certain software faults. That is, in many cases of processor halts due to software faults, the backup of a failed primary can continue the execution. This observation is rather counterintuitive, because the primary and backup run the same copy of the software. The phenomenon is explained by the existence of subtle software faults that are not exercised again on a restart of the failed software. Usually, field software faults not identified during the testing phase are subtle and require very specific conditions to be triggered. Since the process pair technique was not explicitly intended for tolerating software faults, study of field data is essential for understanding this phenomenon and for measuring its effectiveness.

This section investigates the user-perceived ability of the Tandem system to tolerate faults in its system software [24]. The software faults considered here are major defects that result in processor failures. Although process pairs are specific to Tandem systems, they are an implementation of the general approach of checkpointing and restart. This evaluation is important because it suggests that these may be low-cost techniques for achieving software fault tolerance in large, continually evolving software systems. Attempts were recently made in [28], [29] to take advantage of the subtle nature of some software faults to enhance software fault tolerance in user applications.

A. Measure of Software Fault Tolerance

Table II shows the severity of the measured software failures. In this table, a single processor halt implies that the built-in single-failure tolerance of the system masked the software fault that caused the halt. All multiple processor halts were grouped because, given the Tandem architecture, a double processor halt can potentially cause additional processor halts. For example, if the system loses a set of disks as a result of a double processor halt and the set of disks contains files required by other processors, additional processor halts can occur. There was one case in which a software failure occurred in the middle of a system reboot. Since each TPR reports just one problem, sometimes two TPRs were generated as a result of a multiple processor halt. There were five such cases. Thus, the 179 TPRs reported 174 software failures.

TABLE II
SEVERITY OF SOFTWARE FAILURES

Severity	# Failures	Further Characterized in
Single processor halt	138	Table III
Multiple processor halt	31	Table IV
Halt occurring during system reboot	1	–
Unable to classify	4	–
All	174	

In this evaluation, the term *software fault tolerance* (SFT) refers to the system's ability to tolerate software faults. Quantitatively, it is defined as

$$SFT = \frac{\text{number of software failures in which a single processor is halted}}{\text{total number of software failures}}. \quad (1)$$

SFT represents the user-perceived ability of the system to tolerate faults in its system software due to the use of process pairs. Table II shows that process pairs provide a significant level of software fault tolerance in the Tandem GUARDIAN environment. The measure of software fault tolerance is estimated to be 82% (138 out of 169, excluding the five special cases).[1]

B. Outages Due to Software

This evaluation first focused on the multiple processor halts. For each multiple processor halt, we investigated the first two processor halts to determine whether the second halt occurred on the processor executing the backup of the failed primary process. In these cases, we also investigated whether the two processors halted because of the same software fault.

TABLE III
REASONS FOR MULTIPLE PROCESSOR HALTS

Reasons for Multiple Processor Halts	# Failures
The second halt occurs on the processor executing the backup of the failed primary.	24
The second halt occurs due to the same fault that halted the primary.	(17)
The second halt occurs due to another fault during job takeover.	(4)
Unable to classify.	(3)
The second halt is not related to process pairs.	4
The system hangs.	(1)
Faulty parallel software executes.	(1)
There is a random coincidence of two independent faults.	(1)
A single processor halt occurs, but system coldload is necessary for recovery.	(1)
Unable to classify.	3
All	31

The level of software fault tolerance achieved with process pairs is high, but not perfect: a single fault in the system software can manifest itself as a multiple processor halt, which the system is not designed to tolerate. Table III shows that in 86% of the multiple processor halts (24 out of 28, excluding "Unable to classify" cases), the backup of the failed primary process was unable to continue the execution. In 81% of these halts (17 out of 21, excluding "Unable to classify" cases), the backup failed because of the same fault that caused the failure of the primary. In the remaining 19% of the halts, the processor executing the backup of the failed primary halted because of another fault during job takeover. About half of the multiple processor halts resulted in system coldloads. (A system coldload is a situation in which all processors in a system are reloaded.) The data showed that, in most situations, the system lost a set of disks that contained files required by other processors as a result of the first two processor halts, and other processors also halted. This sequence is the major failure mode of the system resulting from software faults.

C. Characterization of Software Fault Tolerance

The information in Table II raises the question of why the Tandem system lost only one processor in 82% of software failures and, as a result, tolerated the software faults that caused these failures. We identified the reasons for software fault tolerance (SFT) in all single processor halts (138 instances; refer to Table II) and classified them into several groups. Table IV shows that in 29% of single processor halts (40 out of 138), the fault that caused a failure of a primary process was not exercised again when the backup reexecuted the same task after a takeover. These situations occurred because some software faults are exposed in a specific memory state (e.g., running out of buffer), on the occurrence of a single event or a sequence of asynchronous events during a vulnerable time window (timing), by race conditions or concurrent operations among multiple processes, or on the occurrence of a hardware error.

TABLE IV
REASONS FOR SOFTWARE FAULT TOLERANCE

Reasons for Software Fault Tolerance	Fraction (%)
The backup reexecutes the failed task after takeover, but the fault that caused a failure of the primary is not exercised by the backup.	29
Memory state	(4)
Timing	(7)
Race or concurrency	(6)
Hardware error	(4)
Others	(7)
The backup, after takeover, does not automatically reexecute the failed task.	20
It is the effect of error latency.	5
A fault stops a processor running a backup.	16
The cause of a problem is unidentified.	19
Unable to classify.	12

Fig. 4 shows a real example of a fault that is exercised in a specific memory state. The primary of an I/O process pair, which is represented by SIOP(P) in the figure, requested a buffer to serve a user request. Because of the high activity in the processor executing the primary, the buffer was not avail-

[1] This measure is based on reported software failures. The issue of underreporting was discussed in [13]. The consensus among experienced Tandem engineers is that about 80% of software failures are not reported as TPRs and that most of them are single processor halts. If that assessment is true, then the software fault tolerance may be as high as 96%.

able. However, because of a software fault, the buffer management routine returned a "successful" flag, instead of an "unsuccessful" flag. The primary used the returned, uninitialized buffer pointer, and a halt occurred in the processor running the primary because of an illegal address reference by a privileged process. Clearly, such a situation was not tested during the development phase. Since a memory dump is usually taken only from a halted processor in a production system, a memory dump of the processor running the backup was not available. Our best guess is that the backup process served the request again after takeover but did not have a problem, because a buffer was available on the processor running the backup.

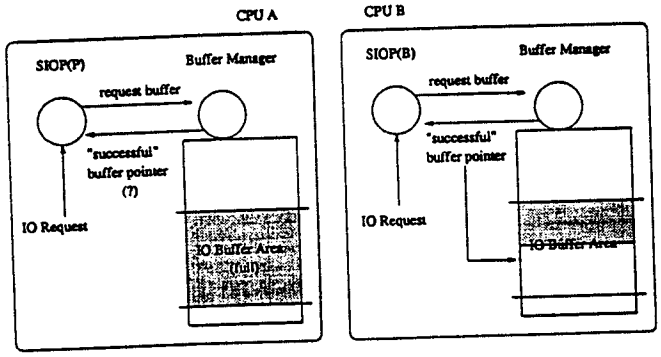


Fig. 4. Differences between the primary and backup executions.

Table IV also shows that, in 20% of single processor halts (28 out of 138), the backup of a failed primary process did not have to serve the failed request after a successful takeover. This happened because some faults are exposed while serving requests that are important but are not automatically resubmitted to the backup upon a failure of the primary. Fig. 5 illustrates an example of such a situation. In the figure, process PK is an execution of a utility to monitor processor activity for memory usage, message information, and paging activity. Process PK does not run as a process pair because, if the processor being monitored halts while executing PK, there is no need to monitor the halted processor any longer. Process MS collects resource usage data, and process TM is in charge of concurrency control and failure recovery. Both MS and TM run as process pairs.

When the operator ran PK with a certain option that is not frequently used, PK used an incorrect constant to initialize its data structure. As a result, it overwrote (cleared) the page addresses of the first segment in the segment page table. The first segment is always owned by MS, and MS was running on the processor. When MS stored resource usage data, it used incorrect addresses (addresses of zero) and corrupted the system global data. A processor halt occurred as a result of an address violation when TM accessed and used the address of a system data table. When the backups of the failed primaries took over, they did not have problems, because PK was running only on the halted processor.

Another example is the faults that cause processor failures during the execution of the operator requests for reconfiguring I/O units. An I/O unit is a device or program by which an end-user (a terminal operator, an application program, or an I/O mechanism) gains access to the system. Utilities to perform these reconfigurations run as process pairs, but the operator command to add, activate, or abort an I/O unit is not automatically resubmitted to the backup, because it is an interactive task that can easily be resubmitted by the operator if the primary fails. Suppose that an operator's request to add an I/O unit caused a failure of the primary. In this situation, the operator would typically recover the halted processor, rather than submit the same request to the backup. If the operator wants to repeat the same request, he or she would normally repeat it on the primary after the halted processor is reloaded. If the operator submits the request to the backup instantly upon a failure of the primary, one of two situations can be expected: the backup also fails, or the backup serves the request without any problem due to the factors in Table IV.

In the above examples, the task (i.e., process PK or a command to add an I/O unit) does not survive the failure. But process pairs allow the other applications on the halted processor to continue to run. This situation is not strictly SFT but a side benefit of using process pairs. If these failures are excluded, the estimated measure of SFT is adjusted to 78% (110 out of 141).

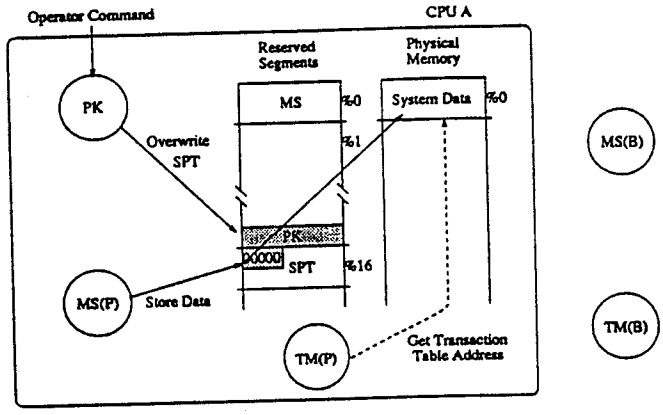


Fig. 5. Faults exposed by non-process pairs.

Another reason for the SFT is that some software faults cause errors that are detected after the task that caused the errors finishes successfully (effect of error latency). Fig. 6 shows an example. The figure shows a data transfer between two primary I/O processes: SIOP(P) and XIOP(P). The underlying software fault was an extra line in the SIOP software that caused SIOP(P) to transfer one more byte than was necessary. This fault did not always cause a problem, because the size of a buffer is usually bigger than the size of a message. When a message and a buffer had equal sizes, the first byte in the end tag of the buffer was overwritten. This corruption did not affect the data transfer, because tags are not a part of data area. (The tags are used to check the integrity of a data structure, bu for performance reasons, they are not checked after every data transfer.) The data transfer was successfully completed an checkpointed to the backup. The corrupted buffer tag was no a part of the checkpoint information. The corruption in the en tag was found later, when SIOP(P) returned the buffer to th

buffer manager. The buffer manager checked the integrity of the begin and end tags, found a corruption, and asserted a halt of the processor it runs on ("CPU A" in Fig. 6). The backups of the failed primaries would take over, but they would not have problems because the data transfer that caused the error was already completed successfully. The difference between this case and the first group of cases listed in Table IV is that the task that caused the failure of the primary did not have to be executed again in the backup.

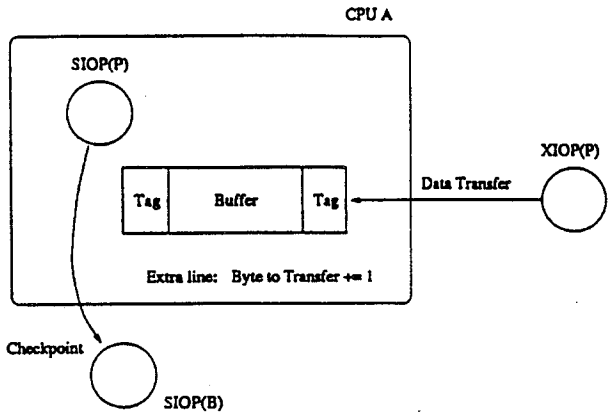


Fig. 6. Effect of error latency.

Table IV also shows that 16% of single processor halts (22 out of 138) were failures of backup processes. This result indicates that the SFT did not come without a cost; the added complexity due to the implementation of process pairs introduced software faults into the system software. The estimated measure of SFT (78%) can be adjusted again to 74% (88 out of 119) when these failures are excluded. All unidentified failures were single processor halts. This is understandable, because these failures were caused by subtle faults that are difficult to observe and diagnose. The reason that an unidentified failure caused a single processor halt is unknown. Based on their symptoms, we speculate that a significant number of unidentified failures were single processor halts because of the effect of error latency.

D. Discussion

The results in this section have several implications. First, the results show that hardware fault tolerance buys SFT. The use of process pairs in Tandem systems, which was originally intended for tolerating hardware faults, allows the system to tolerate about 75% of reported field faults in the system software that cause processor failures. Subtle faults exist in all software, but SFT is not achieved if the backup execution is a replication of the original execution. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution, is a major reason for the measured SFT. Each processor in a Tandem system has an independent processing environment; therefore, the system naturally provides such differences. (The advantages of using checkpointing, as compared with lock-step operation, in tolerating software faults were discussed in [19].) The level of SFT achieved by the use of process pairs will depend on the proportion of subtle faults in software. While process pairs may not provide perfect SFT, the implementation of process pairs is not as prohibitively expensive as is developing and maintaining multiple versions of large software programs.

Second, the results indicate that process pairs can sometimes allow the system to avoid multiple processor halts due to software faults, regardless of the nature of the faults, because software failures can occur while the system executes important tasks that are not automatically resubmitted to the backup on a failure of the primary. In such a case, the failed task does not survive, but the other applications on the failed processor do.

Third, short error latency with error confinement within a transaction is desirable [30]. In actual designs, such a strict error confinement might be rather difficult to achieve. Errors generated during the execution of a transaction may be detected during the execution of another transaction. Interestingly, long error latency and error propagation across transactions sometimes help the system to tolerate software faults. This result should not be interpreted to suggest that long error latency or error propagation across transactions is a desirable characteristic. It is a side effect of the system having subtle software faults. Long error latency and error propagation across transactions can make both on-line recovery and off-line diagnosis difficult.

Finally, an interesting question is: If process pairs are good, are process triples better? Our results show that process triples may not necessarily be better, because the faults that cause double processor halts with process pairs may cause triple processor halts with process triples.

E. First Occurrences vs. Recurrences

Table V compares the severity of the three types of software failures using the 174 software failures discussed in this section. There were two special cases ("Others") in the table: a multiple processor halt due to a parallel execution of faulty code (a system coldload was not required) and a software failure during a system reboot. With only a single observation in each case, the significance of these situations was unclear, and they were not considered in the subsequent analysis. "Severity Unclear" cases were also not considered further.

TABLE V
SEVERITY OF SOFTWARE FAILURES BY FAILURE TYPE

	#Failure Instances	#Double CPU Halts	#System Coldloads	#Severity Unclear	#Others
First occurrence	41	9	6	1	1
Recurrence	107	19	12	3	1
Unidentified	26	0	0	0	0

TableV indicates that a recurrence is slightly less likely than a first occurrence to cause a double processor halt. The binomial test was used to test this observation, because it does not require an assumption about the underlying distribution to construct a confidence interval [31]. Each failure was treated

as a random trial with the probability of a double processor halt being 0.23 (nine out of 39, following the statistics for the first occurrence). The hypothesis that the probability of a recurrence causing a double processor halt is equal to that of a first occurrence causing a double processor halt was tested by calculating the probability of having 19 or fewer double processor halts out of 103 trials. The p-value was 0.16; that is, the hypothesis was rejected at the 20% significance level.

Two of the six system coldloads due to first occurrences were single processor halt situations. These two failures capture the secondary failure mode of the system due to software, wherein a system is coldloaded to recover from a severe, single processor halt.

VI. Reliability Modeling of Operational Software

Software reliability models attempt to estimate the reliability of software. Many models have been proposed [1], [2], [3]. These models typically attempt to relate the history of fault identification during the development phase, verification efforts, and operational profile. The primary focus is on the software development phase, and the underlying assumptions are that software is an independent entity and that each software fault has the same impact.

The results from the previous sections indicated that other factors significantly impact the dependability of operational software. First, software faults can be highly visible or less visible. A single, highly visible software fault can cause many field failures, and recurrences can seriously degrade software dependability in the field. Second, for a class of software such as the GUARDIAN operating system, the fault tolerance of the overall system can significantly improve software dependability by making the effects of software faults invisible to users. Clearly, dependability issues for operational software in general can be quite different from those for the software in the development phase. Discussion of software reliability in the system context was provided in [32]. An approximate model to account for failures due to design faults was used to evaluate the dependability of operational software in [33]. The use of information on system usage (i.e., installation trail) to predict software reliability and to determine test strategy was discussed in [34].

This section asks the question: Which factors determine the dependability of the measured operating system? Using the software failure and recovery characteristics identified in the previous sections, this section builds a model to describe the impact of faults in the GUARDIAN operating system on the reliability of an overall Tandem system. Based on the model, the section conducts a sensitivity analysis to evaluate the significance of the factors considered and to identify areas for improvement.

A. Model Construction

We considered a hypothetical eight-processor Tandem system whose software reliability characteristics are described by the parameters in Table VI. In this analysis, the term *software reliability* means the reliability of an overall system when only the faults in the system software that cause processor failures are considered. A system failure was defined to occur when more than half the processors in the system failed. All parameters in the table except λ and μ were estimated based on the measured data (Sections IV and V). The values of λ and μ were determined to mimic the 30 years of software mean time between failure (MTBF) and the mean time to repair (MTTR) characteristics reported in [13]. Thus, the objectives of the analysis were to model and evaluate reliability sensitivity to various factors, not to estimate the absolute software reliability.

TABLE VI
Estimated Software Reliability Parameters

Failures:	First Occurrence	Recurrence	Unidentified
Failure rate	$\lambda_f = 0.24\,\lambda$	$\lambda_r = 0.61\,\lambda$	$\lambda_u = 0.15\,\lambda$
Prob(double CPU halt|software failure)	$C_{df} = 0.23$	$C_{dr} = 0.18$	$C_{du} = 0.0$
Prob(system failure| double CPU halt)	$C_{sdf} = 0.44$	$C_{sdr} = 0.63$	$C_{sdu} = 0.0$
Prob(system failure| single CPU halt)	$C_{ssf} = 0.05$	$C_{ssr} = 0.0$	$C_{ssu} = 0.0$
Failures: Software failure rate = $\lambda = 0.32$/year			
Recovery: Recovery rate = $\mu = 3$/hour			

In Table VI, "Prob(double CPU halt|software failure)" is the probability that a double processor halt (i.e., a failure of a process pair) occurs given that a software failure occurs. Similarly, "Prob(system failure|double CPU halt)" is the probability that a system failure occurs given that a double processor halt occurs. These two parameters were used to describe the major failure mode of the system because of software. The parameter "Prob(system failure|single CPU halt)" represents the secondary failure mode, which captures single processor halts severe enough to cause system coldloads. The table shows these probabilities for first occurrences, recurrences, and unidentified failures.

Based on the parameters in Table VI and on the following assumption, we built a continuous-time Markov model to describe the software failure and recovery in a hypothetical eight-processor Tandem system in the field.

ASSUMPTION 1. *The time between software failures in the system has an exponential distribution, and the three types of failures (first occurrence, recurrence, and unidentified) are randomly mixed.*

This assumption was necessary, because determining the above characteristics for a single system would require a minimum of a few hundred years of measurements. The assumption could not be validated using the measured data, because the measured data was collected from a large number of user systems running different versions of the operating system and having different operational environments and system configurations. Given this situation, the assumption seemed reasonable.

Fig. 7 shows the Markov model. In the model, S_i, $i = 0, .., 4$ represents that i processors are halted because of software faults. A system failure is represented by the S_{down} state. To evaluate software reliability, no recovery from a system failure was assumed. That is, the system failure state is an absorption state. The R_i state represents an intermediate state in which the system tries to recover from an additional software failure (ith processor halt) using process pairs.

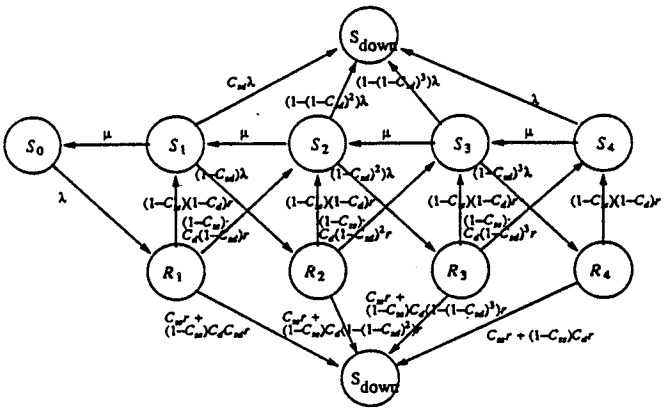


Fig. 7. Software reliability model.

If a software failure occurs during the normal system operation (i.e., when the system is in the S_0 state), the system enters the R_1 state. If the failure is severe enough to cause a system coldload, a system failure occurs; otherwise, the system attempts to recover from the failure by using backups. If recovery is successful, the system enters the S_1 state; otherwise, a double processor halt occurs. If the two halted processors control key system resources (such as a set of disks) that are essential for system operation, the rest of the processors in the system also halt and a system failure occurs; otherwise, the system enters the S_2 state and continues to operate. The value of r, the transition rate out of an R_i, is small and has virtually no impact on software reliability; a value of one transition per minute was used in the analysis. Since the system stays in an R_i state for a short time, additional failures occurring in an R_i state were ignored; in fact, these failures were implied in the failure rate (λ) in the corresponding S_i and S_{i+1} states. Given the model in Fig. 7, software reliability of the system can be estimated by calculating the distribution of time for the system to be absorbed to the S_{down} state, starting from the S_0 state.

In Fig. 7, the three coverage parameters C_d, C_{sd}, and C_{ss} were calculated from Table VI:

$$C_d = \text{Prob.(double CPU halt}|\text{software failure)} = \frac{\lambda_f C_{df} + \lambda_r C_{dr} + \lambda_u C_{du}}{\lambda_f + \lambda_r + \lambda_u}, \quad (2)$$

$$C_{sd} = \text{Prob.(system failure}|\text{double CPU halt)} = \frac{\lambda_f C_{df} C_{sdf} + \lambda_r C_{dr} C_{sdr} + \lambda_u C_{du} C_{sdu}}{\lambda_f C_{df} + \lambda_r C_{dr} + \lambda_u C_{du}}, \quad (3)$$

and

$$C_{ss} = \text{Prob.(system failure}|\text{single CPU halt)} = \frac{\lambda_f C_{ssf} + \lambda_r C_{ssr} + \lambda_u C_{ssu}}{\lambda_f + \lambda_r + \lambda_u}. \quad (4)$$

The parameter C_d includes the two cases explained in Section V: the failure of a process pair caused by a single software fault and the failure of a process pair caused by two software faults (the second halt occurs during job takeover). The parameter C_{sd} represents the probability that the system loses key system resources as a result of a double processor halt. The parameter C_{sd} is determined primarily by the system configuration and is discussed further in Section VI.D. The above three parameters can actually be obtained directly from Table V in Section V.E. Equations (2), (3), and (4) will be used to investigate the impact of recurrences (λ_r) on software reliability in Section VI.B.

The model (Fig. 7) includes the effect of multiple independent software failures. For example, if a software failure occurs when the system is in the S_i state ($i \neq 0$), the following three system failure scenarios must be considered (Fig. 8):

1) The system fails regardless of whether the new failure causes a single or double processor halt. This is because when the first processor halts because of the new failure, key system resources (such as a set of disks) become inaccessible.
2) The system fails because the new failure is severe and can only be recovered by a system coldload.
3) The new software failure causes a double processor halt, and the second processor halt causes a set of disks to become inaccessible.

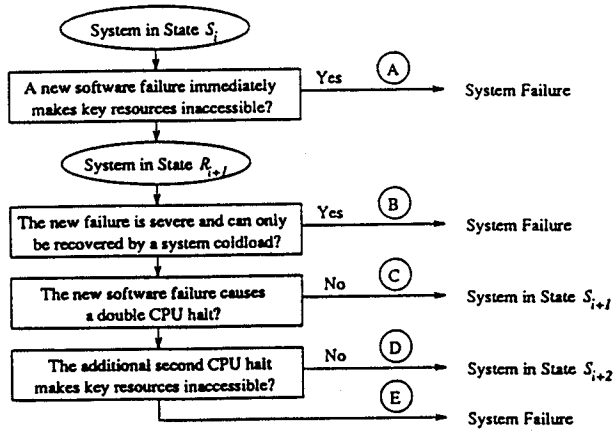


Fig. 8. Effect of multiple independent software failures.

It was not possible to directly measure the branching probabilities in Fig. 8 for each state from the data, because the major failure mode (i.e., a software failure occurred when the system is in the S_0 state, causing a double processor halt and subsequently causing a system failure) was dominant. These probabilities were estimated using the three measured parameters: C_d, C_{sd}, and C_{ss}. Table VII shows the branching probabilities in Fig. 8 estimated for each S_i ($i \neq 0$) state. For exam-

TABLE VII
PARAMETERS FOR MULTIPLE INDEPENDENT SOFTWARE FAILURES

	$i = 1$ (i.e., system in S_1)	$i = 2$	$i = 3$	$i = 4$
Path A	C_{sd}	$1 \quad (1 \quad C_{sd})^2$	$1 \quad (1 \quad C_{sd})^3$	1
Path B	$(1 \quad C_{sd})C_{ss}$	$(1 \quad C_{sd})^2 C_{ss}$	$(1 \quad C_{sd})^3 C_{ss}$	–
Path C	$(1 \quad C_{sd})(1 \quad C_{ss})(1 \quad C_d)$	$(1 \quad C_{sd})^2 (1 \quad C_{ss})(1 \quad C_d)$	$(1 \quad C_{sd})^3 (1 \quad C_{ss})(1 \quad C_d)$	–
Path D	$(1 \quad C_{sd})(1 \quad C_{ss})C_d (1 \quad C_{sd})^2$	$(1 \quad C_{sd})^2 (1 \quad C_{ss})C_d (1 \quad C_{sd})^3$	–	–
Path E	$(1 \quad C_{sd})(1 \quad C_{ss})C_d (1 \quad (1 \quad C_{sd})^2)$	$(1 \quad C_{sd})^2 (1 \quad C_{ss})C_d (1 \quad (1 \quad C_{sd})^3)$	$(1 \quad C_{sd})^3 (1 \quad C_{ss})C_d$	–

ple, given that an additional software failure causes a double processor halt when the system is in the S_1 state, the probability that the third processor halt does not cause a system failure (path D in Fig. 8) is $(1 \quad C_{sd})^2$. This is because the probability that the third processor halted and either of the two processors that were already halted control key system resources (i.e., cause a system failure) is C_{sd}. The branching probabilities in Table VII were used to determine the corresponding transition rates in the model (Fig. 7).

The same recovery rate was used regardless of the number of processors halted. This was because the recovery time is typically determined by the time required to perform a quick diagnosis and take a memory dump, which is done for one processor at a time. Previous studies assumed that the failure rate is proportional to the number of processors up and working [35]. The same software failure rate was assumed in all states, considering that, as more processors halt, the remaining processors will receive more stress. Again, the dominance of the major system failure mode did not allow us to estimate the parameters from the data.

The distribution of time for the system to be absorbed to the system failure state, starting from the normal state, was evaluated using the model in Fig. 7. SHARPE [36] was used for the evaluation. Fig. 9 shows the software reliability curve of the modeled system and confirms the assumed software MTBF of 30 years. The figure represents the reliability of an overall Tandem system in the field when only the faults in the system software that caused processor failures were considered.

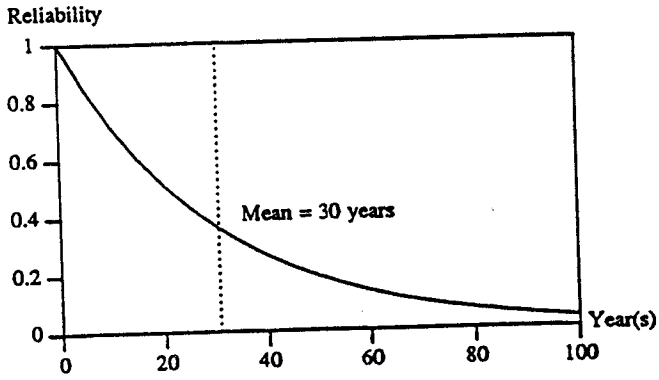


Fig. 9. System reliability due to software.

B. Reliability Sensitivity Analysis

Table VIII shows the six factors considered in the analysis. The second column of the table shows activities related to these factors, and the third column shows the model parameters affected by the factors. For example, a 10% reduction in the recurrence rate (λ_r), which can be achieved by improving the software service environment, will reduce λ by 6.1% (Table VI) and change C_d, C_{sd}, and C_{ss} accordingly. Refer to Equations (2), (3), and (4).

The coverage parameters C_d and C_{sd} are determined primarily by the robustness of process pairs and the system configuration, respectively. For example, C_d can be reduced by conducting extra testing of the routines related to job takeover. The parameter C_{sd} is determined by the location of failed process pairs and the disk subsystem configuration. This parameter is discussed further in Section VI.D. Analytical models for predicting coverage in a fault-tolerant system and the sensitivity of system reliability/availability to the coverage parameter were discussed in [37]. The recovery rate μ can be improved by automating the data collection and reintegration process.

TABLE VIII
FACTORS OF SOFTWARE RELIABILITY

Factor	Activity	Related Parameters: Detailed	Related Parameters: Overall
Software failure rate	Software development	λ_f, λ_r, λ_u	λ
Recurrence rate	Software service	λ_r	λ, C_d, C_{sd}, C_{ss}
Coverage parameter C_d	Robustness of process pairs	C_{df}, C_{dr}, C_{du}	C_d
Coverage parameter C_{sd}	System configuration	C_{sdf}, C_{sdr}, C_{sdu}	C_{sd}
Coverage parameter C_{ss}	-	C_{ssf}, C_{ssr}, C_{ssu}	C_{ss}
Recovery time	Diagnosability/ maintainability	μ	μ

Fig. 10 shows the software MTBF evaluated using the model in Fig. 7 while varying the six factors in Table VIII, one at a time. It is interesting to see that C_d and C_{sd} are almost as important as λ in determining the software MTBF. For example, a 20% reduction in C_d or C_{sd} has as much impact on software MTBF as an 18% reduction in λ. (The figure shows that the impact is approximately a 20% increase in software MTBF.) This result is understandable because the system fails primarily because of a double processor halt causing a set of disks to become inaccessible, not because of multiple independent software failures.

Fig. 10 also shows that the recurrence rate has a significant impact on software reliability. A complete elimination of recurrences ($\lambda_r = 0$ in Table VI) would increase the software MTBF by a factor of three. The impact of C_{ss} on software reliability is small, because severe, single processor halts causing system coldloads are rare. The impact of μ on software MTBF is virtually nil. In other words, recovery rate is not a factor as far as software reliability is concerned, again, because the system is unlikely to fail because of multiple independent software failures.

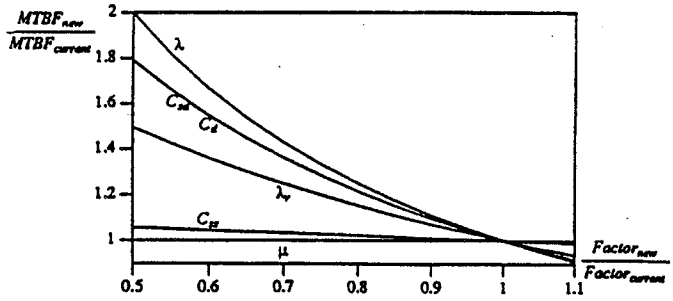


Fig. 10. Software MTBF sensitivity.

Typically, it is assumed that the number of faults in software is the only major factor determining software reliability. Fig. 10 clearly shows that in the Tandem system, there are four degrees of freedom in improving the software reliability: the number of faults in software, the recurrence rate, the robustness of process pairs, and the system configuration strategy. The first two are general factors, and the last two are platform-dependent factors. Efforts to improve software reliability can be optimized by estimating the cost of improving each of these four factors.

C. Reliability Sensitivity to Fault Category

This section investigates the impact of software faults in different fault categories (Table I in Section IV) on software reliability. In this section, a *failure group* is defined as the group of software failures caused by all faults that belong to a fault category. We estimated the software MTBF by assuming that each failure group is empty, i.e., the faults in a fault category did not cause software failures. The failure rate and the coverage parameters for the model in Fig. 7 were adjusted:

$$\lambda = \frac{\text{total no. of software failures} - \text{no. of software failures in a failure group}}{\text{total no. of software failures}} \lambda, \quad (5)$$

$$C_d = \frac{\text{total no. of double CPU halts} - \text{no. of double CPU halts in a failure group}}{\text{total no. of software failures} - \text{no. of software failures in a failure group}}, \quad (6)$$

$$C_{sd} = \frac{\text{total no. of system failures} - \text{no. of system failures in a failure group}}{\text{total no. of double CPU halts} - \text{no. of double CPU halts in a failure group}}, \quad (7)$$

and

$$C_{ss} = \frac{\text{total no. of severe, single CPU halts} - \text{no. of severe, single CPU halts in a failure group}}{\text{total no. of software failures} - \text{no. of software failures in a failure group}}. \quad (8)$$

In Equation (7), only those system failures caused by double processor halts (i.e., failures of process pairs) were counted.

Table IX shows the results. The last column of the table shows the improvement in software MTBF when failures caused by each fault category were eliminated. Only those categories that have more than 10 failures were considered. The table shows that "Missing operation" caused the greatest reliability loss. Further analysis showed that uninitialized pointers (Table I in Section IV) were responsible for more than half of the loss caused by this group of failures. The table also shows that "Unexpected situation" was another significant source of reliability loss. Most of this loss is attributed to faults such as incorrect parameters passed by user processes, illegal procedure calls made by user processes, and not considering all legitimate operational scenarios in designing software. (The reliability loss is not attributed to subtle faults, such as race conditions and timing problems.) Additional code inspection and testing efforts can be directed to these fault categories. Unidentified failures had virtually no impact on software reliability, because all of these failures caused single processor halts.

TABLE IX
RELIABILITY SENSITIVITY TO FAULT CATEGORY

Fault Category	#Failures	$\frac{MTBF_{improved}}{MTBF_{current}}$
Incorrect computation	3	-
Data fault	21	1.00
Data definition fault	7	-
Missing operation	27	1.47
Side effect of code update	5	-
Unexpected situation	46	1.35
Microcode defect	8	-
Others	12	1.06
Unidentified	26	1.00
Unable to classify	24	1.12

D. Impact of System Configuration on Software Dependability

System configuration is an issue that demonstrates the importance of considering the interactions between hardware, software, and operations. Table X shows a breakdown of the process pairs whose failures caused the 18 observed system failures, based on their configurability. In the table, a "Location-free" process pair is a pair that can be placed on any two processors in the system, independent of hardware configuration. The location of a nondisk or disk I/O process pair is determined by hardware configuration. The failure of a nondisk I/O or location-free process pair causes a system failure, because the process pair executes on the two processors that execute a disk process pair. Thus, a double processor halt resulting from a failure of such a nondisk I/O or location-free process pair would cause a set of disks to become inaccessible.

TABLE X
CONFIGURABILITY OF FAILED PROCESS PAIRS THAT CAUSED SYSTEM FAILURES

Failed Process Pair	#System Failures
Location-free process pair	7
Nondisk I/O process pair	5
Disk I/O process pair	2
Others	4

Table X shows that the number of system failures could potentially be reduced by 67% (12 out of 18) by avoiding the overlap in location between disk process pairs and the failed nondisk I/O or location-free process pairs. This result demonstrates the importance of considering software dependability in the context of an overall system.

VII. Conclusions

Based on field failure data collected from the Tandem GUARDIAN operating system, this paper discussed evaluation of the dependability of operational software. The software faults considered are major defects that result in processor failures and invoke backup processes to take over. The paper categorized the underlying causes of software failures, discussed the significance of failure recurrence, and evaluated the effectiveness of the process pair technique in tolerating software faults. The paper built a model to describe the impact of faults in the GUARDIAN operating system on the reliability of an overall Tandem system. The model was used to evaluate the significance of key factors that determine software dependability and to identify areas for improvement.

An analysis of the data showed that about 77% of processor failures that are initially considered due to software are confirmed as software problems, 13% of them are probably software problems, and the rest are confirmed as non-software problems. The analysis showed that hardware fault tolerance buys SFT. Using process pairs in Tandem systems, which was originally intended for tolerating hardware faults, allows the system to tolerate about 75% of reported software faults that result in processor failures. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events) being different from the original execution, is a major reason for the measured SFT. This shows that the checkpointing and restart technique can be used as a low-cost SFT strategy. The results indicated that the actual level of SFT achieved by the use of process pairs depends on the degree of difference in the processing environment between the original and backup executions and on the proportion of subtle faults in the software.

Over two-thirds (72%) of reported software failures in Tandem systems are recurrences of previously reported faults. The modeling, based on the data, showed that, in addition to reducing the number of software faults, software dependability in Tandem systems can be enhanced by reducing the recurrence rate and by improving the robustness of process pairs and the system configuration. Omission of lines of source code and not providing routines to handle rare but legitimate operational scenarios are the most common types of software faults in the GUARDIAN operating system. These types of faults are also the major causes of software reliability loss. The investigation of the impact of system configuration on software dependability demonstrated the importance of considering software dependability in the context of an overall system.

It is suggested that more measurements and analyses be conducted in the manner proposed here so that a wide range of information on the dependability of operational software is available.

Acknowledgments

We thank Tandem Computers Incorporated for their assistance in conducting this study. Many Tandem engineers provided technical help throughout the study. Special thanks are due to Gil Pitt and Bob Horst for their support and helpful suggestions, to Alan Wood and Randy McRee for their useful comments, and to Max Dietrich, Harish Joshi, Gary Smith, David Wong, and Leon Zar for their help in understanding the data. Thanks are also due to Fran Wagner for proofreading the manuscript.

References

[1] C. V. Ramamoothy and F. B. Bastani, "Software reliability—status and perspectives," *IEEE Trans. on Software Engineering*, vol. 8, no. 4, pp. 354–371, July 1982.

[2] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*. New York, N.Y.: McGraw-Hill Book Company, 1987.

[3] S. Yamada, M. Ohba, and S. Osaki, "S-shaped software reliability growth models and their applications," *IEEE Trans. on Reliability*, vol. 33, no. 4, pp. 289–292, Oct. 1984.

[4] A. Endres, "An analysis of errors and their causes in system programs," *IEEE Trans. on Software Engineering*, vol. 1, no. 2, pp. 140–149, June 1975.

[5] T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability*. New York, N.Y.: Elsevier North-Holland Publishing Company, Inc., 1978.

[6] D. M. Weiss, "Evaluating software development by error analysis: The data from the architecture research facility," *J. System and Software*, vol. 1, pp. 57–70, Mar. 1979.

[7] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Comm. of the ACM*, vol. 22, no. 1, pp. 42–52, Jan. 1984.

[8] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification—a concept for in-process measurements," *IEEE Trans. on Software Engineering*, vol. 18, no. 11, pp. 943–956, Nov. 1992.

[9] X. Castillo and D. P. Siewiorek, "A comparable hardware/software reliability model," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, Pa., July 1981.

[10] R. K. Iyer and D. J. Rossetti, "Effect of system workload on operating system reliability: A study on IBM 3081," *IEEE Trans. on Software Engineering*, vol. 11, no. 12, pp. 1,438–1,448, Dec. 1985.

[11] M. C. Hsueh and R. K. Iyer, "A measurement-based model of software reliability in a production environment," *Proc. 11th Ann. Int'l Computer Software & Applications Conf.*, Tokyo, Japan, Oct. 1987, pp. 354–360.

[12] M. S. Sullivan and R. Chillarege, "Software defects and their impact on system availability—a study of field failures in operating systems," *Proc. 21st Int'l Symp. on Fault-Tolerant Computing*, Montreal, Canada, June 1991, pp. 2–9.

[13] J. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. on Reliability*, vol. 39, no. 4, pp. 409–418, Oct. 1990.

[14] I. Lee, D. Tang, R. K. Iyer, and M. C. Hsueh, "Measurement-based evaluation of operating system fault tolerance," *IEEE Trans. on Reliability*, vol. 42, no. 2, pp. 238–249, June 1993.

[15] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *IEEE Computer*, pp. 67–80, Aug. 1984.

[16] B. Randell, "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, no. 1, pp. 220–232, June 1975.

[17] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. on Software Engineering*, vol. 16, no. 2, pp. 166–182, Feb. 1990.

[18] P. Velardi and R. K. Iyer, "A study of software failures and recovery in the MVS operating system," *IEEE Trans. on Computers*, vol. 33, no. 6, pp. 564–568, June 1984.

[19] J. Gray, "Why do computers stop and what can we do about it?" Tandem Computers Inc., Cupertino, Calif., Tandem Technical Report 85.7, June 1985.

[20] J. Bartlett, W. Bartlett, R. Carr, D. Garcia, J. Gray, R. Horst, R. Jardine, D. Lenoski, and D. McGuire, "Fault tolerance in Tandem computer systems," Tandem Computers Inc., Cupertino, Calif., Tandem Technical Report 90.5, May 1990.

[21] K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. on Computers*, vol. 38, no. 5, pp. 626–636, May 1989.

[22] J. H. Lala and L. S. Alger, "Hardware and software fault tolerance: A unified architectural approach," *Proc. 18th Int'l Symp. on Fault-Tolerant Computing*, Tokyo, Japan, June 1988, pp. 240–245.

[23] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *IEEE Computer*, pp. 39–51, July 1990.

[24] I. Lee and R. K. Iyer, "Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 operating system," *Proc. 23rd Int'l Symp. on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 20–29.

[25] E. N. Adams, "Optimizing preventive service of software products," *IBM J. of Research and Development*, vol. 28, no. 1, pp. 2–14, Jan. 1984.

[26] Y. Levendel, private communication.

[27] I. Lee and R. K. Iyer, "Identifying software problems using symptoms," *Proc. 24th Int'l Symp. on Fault-Tolerant Computing*, Austin, Tex., June 1994, pp. 320–329.

[28] Y. Huang and C. Kintala, "Software implemented fault tolerance: Technologies and experience," *Proc. 23rd Int'l Symp. on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 2–9.

[29] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," *Proc. 23rd Intl. Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 138–144.

[30] F. Cristian, "Exception handling and software fault tolerance," *IEEE Trans. on Computers*, vol. 31, no. 6, pp. 531–540, June 1982.

[31] R. V. Hogg and E. A. Tanis, *Probability and Statistical Inference*, third edition. New York, N.Y.: Macmillan Publishing Co., Inc., 1988.

[32] H. Hecht and M. Hecht, "Software reliability in the system context," *IEEE Trans. on Software Engineering*, vol. 12, no. 1, pp. 51–58, Jan. 1986.

[33] J.-C. Laprie, "Dependability evaluation of software systems in operation," *IEEE Trans. on Software Engineering*, vol. 10, no. 6, pp. 701–714, Nov. 1984.

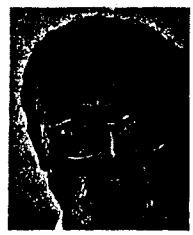
[34] T. Stalhane, "Assessing software reliability in a changing environment," *IFAC SAFECOM*, London, U.K., 1990, pp. 83–88.

[35] M. D. Beaudry, "Performance-related reliability measures for computing systems," *IEEE Trans. on Computers*, vol. 27, no. 6, pp. 540–547, June 1978.

[36] R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Trans. on Reliability*, vol. 36, no. 2, pp. 186–193, June 1987.

[37] J. B. Dugan and K. S. Trivedi, "Coverage modeling for dependability analysis of fault-tolerant systems," *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 775–787, June 1989.

Inhwan Lee received his BS and MS degrees in electrical engineering from Seoul National University, Korea, in 1979 and 1985, respectively, and his PhD in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1994. From 1979 to 1986, he was a research engineer at the Agency for Defense Development in Korea. He is now with Tandem Computers Incorporated. His research interests include fault-tolerant computing, performance and dependability measurement and modeling, software engineering, and computer architecture.



Ravishankar K. Iyer holds a joint appointment as Professor of Electrical and Computer Engineering, Computer Science, and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. He is also Co-Director of the Center for Reliable and High-Performance Computing and the Illinois Computing Laboratory for Aerospace Systems and Software, a NASA Center for Excellence in Aerospace Computing. Professor Iyer's research interests are in the area of reliable computing, measurement and evaluation, and automated design. He has served on several program committees for international conferences and is on the editorial boards of the *Journal of Electronic Testing*, the Springer-Verlag Series on Dependable Computing, and the *IEEE Trans. on Parallel and Distributed Systems*. He is currently Program Co-Chair for FTCS-25 (the Silver Jubilee of the International Symposium on Fault-Tolerant Computing).

Prof. Iyer is an IEEE Computer Society Distinguished Visitor, an Associate Fellow of the American Institute for Aeronautics and Astronautics, a Fellow of the IEEE, and a member of ACM, Sigma Xi, and the IFIP technical committee (WG 10.4) on fault-tolerant computing. In 1991, he received the Senior Humboldt Foundation Award for excellence in research and teaching. In 1993, he received the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems."