Proceedings of the 2002 International Symposium on Software Reliability Engineering

The Impact of Recovery Mechanisms on the Likelihood of Saving Corrupted State

Subhachandra Chandra
CoSine Communications
subhachandra@computer.org

Peter M. Chen
EECS Department
University of Michigan
pmchen@umich.edu

Abstract

Recovery systems must save state before a failure occurs to enable the system to recover from the failure. However, recovery will fail if the recovery system saves any state corrupted by the fault. The frequency and comprehensiveness of how a recovery system saves state has a major effect on how often the recovery system inadvertently saves corrupted state. This paper explores and measures that effect. We measure how often software faults in the application and operating system cause real applications to save corrupted state when using different types of recovery systems. We find that generic recovery techniques, such as checkpointing and logging, work well for faults in the operating system. However, we find that they do not work well for faults in the application because the very actions taken to enable recovery often corrupt the state upon which successful recovery depends.

1. Introduction

Recovering from software faults is a difficult and complicated task. On one hand, the recovery system must save enough state before the failure occurs to enable the system to recover. On the other hand, the recovery system must *not* save any state corrupted by the fault, since that would preserve the effects of the failure in all future recovery attempts. In addition, the recovery system must detect the failure before it corrupts critical state or sends incorrect information to other processors. Prior recovery research has focused primarily on saving enough state to enable recovery. Much less attention has been given to saving only uncorrupted state (other than minimizing the latency of error detection). Because of this emphasis on saving enough state, most recovery systems ignore the impact their own actions may have on the likelihood of saving corrupted state.

In reality, the method used to recover from failures has a marked effect on the likelihood that a program will save corrupted state, and thus on the likelihood that recovery will succeed. For example, logging each application event ensures that an application failure will effectively save corrupted state [10]. Recovery systems that ignore this effect reduce the scope of failures for which they recover.

The goal of this paper is to explore the interaction of recovery protocols and the likelihood of saving corrupted state in the presence of software faults (both in the application and the operating system). We begin by discussing some factors that influence whether the recovery systems will save corrupted state: the comprehensiveness and frequency of the recovery system, the location of the fault, and the quality of error detection present in the program. After describing the setup used in our work, we then present experimental results from several real applications in order to quantify the interaction between the recovery method and the likelihood of saving corrupted state.

2. Factors that influence the likelihood of saving corrupted state

Four main factors affect the likelihood that the recovery system will save corrupted state: the comprehensiveness of state saved by the recovery system, the frequency of state saved by the recovery system, the location of the fault, and the quality of error detection. In this section, we discuss each of these four factors.

The first two factors relate to the system used to recover after a failure. Recovery methods are built into application or system software to shield the user from the effects of a failure and to preserve the user's data. The recovery system must periodically save the state of the program so it can restart the program from an appropriate point after a failure. There are two main types of recovery systems: application-generic and application-specific. Application-generic recovery systems, such as those based on checkpointing and logging, operate independently from the program being recovered [6]. That is, they require no

information about the semantics of the state being saved, and they require only basic information about the events in the program, such as which events are visible outside the program ("output events") and which events are non-deterministic [10]. In contrast, an application-specific recovery system is written for a specific program and utilizes semantic information specific to that application. Most applications today include program-specific code to recover after a failure, while relatively few rely on generic methods of recovery.

The next two subsections explore how the comprehensiveness and frequency of the recovery protocol affect the likelihood of saving corrupted state. For both these factors, there are trade-offs between failure transparency for the user, the amount of work to be done by the application programmer, and the likelihood of saving corrupted state.

2.1. Comprehensiveness of state saved by recovery system

One key design decision when building a recovery system is how much state to save. Saving state before a failure preserves the user's work and enables recovery from that point.

Application-specific recovery systems can choose a variety of strategies for saving different pieces of state. The recovery system can choose to discard (i.e. not recover) some state if the user of a program does not mind the loss of that state; for example, few editors save the current cursor position. Or, an application-specific recovery system may choose to invoke a program-specific function to reconstruct a piece of state based on other saved state; for example, a web browser may re-fetch the most recent web page based on a URL history. Finally, an application-specific recovery system may choose to save a piece of state and restore it to the saved value during recovery; for example, the URL history may be saved in a log and recovered from that log.

In contrast, application-generic recovery systems lack information about the semantics of different pieces of state. Because they do not know which state can be lost or how to recover state in an application-specific manner, application-generic recovery systems must save all state and restore it during recovery. Saving state in this manner can be done with either checkpointing or logging.

Saving all state via generic methods like checkpointing and logging has two benefits and two drawbacks. The first benefit is that saving more state makes failures more transparent to the user, because the user is less affected by the failure. The second benefit is that saving all state frees the application programmer from writing code to recover state. However, saving more comprehensive state hurts performance, and it increases the chance that a failure will lead to the recovery system saving corrupted state. The more comprehensive the state that is saved, the more likely it is that state corrupted by the fault will be included in the saved state. State saved by application-specific recovery mechanism also may have been corrupted by the failure. However, these systems tend to save less state than generic recovery systems. Because application-generic recovery systems must save a more comprehensive set of state than application-specific recovery systems, we expect a higher fraction of failures to save corrupted state when using a generic recovery system.

2.2. Frequency of state saves by recovery system

A second key design decision when building a recovery system is how frequently to save state. As with comprehensiveness, the frequency at which state must be saved is determined in large part by whether the recovery system has application-specific knowledge. In essence, recovery systems that know nothing about the application must save state whenever the application executes events that are visible outside the program [10]. Recovery systems that understand application semantics can sometimes avoid saving state at these times. For example, moving the cursor is visible outside the program, but an application-specific recovery system may know that the program does not need to remember this event after recovery.

Increasing the frequency of saving state involves the same trade-offs as increasing the comprehensiveness of the state that is saved. Saving state more frequently makes failures more transparent to the user, because less work is lost during a failure. If state is saved so frequently that each visible event leads to a state save, then application programmers need not be concerned with specifying events that can be lost, so their job becomes easier. However, saving state more frequently increases the likelihood that a failure will lead to the recovery system saving corrupted state. The greater the frequency at which state is saved, the greater the likelihood that state is saved between the fault activation and the halting of the program. In other words, more frequent saves allow less time for error-detection mechanisms to detect the error and stop the program. A failure will save corrupted state if the recovery system saves state after the fault activation and that state includes corruptions induced by the fault activation. Because application-generic recovery systems must save state more frequently than application-specific recovery systems, we expect a higher fraction of failures to save corrupted state when using generic recovery systems.

2.3. Fault location

Another factor that affects how frequently failures save corrupted state is the layer of the fault relative to the recovery system. We view the recovery system as occupying a specific layer in the system. Its job is to recover the layers above it, while leaving the recovery of layers below it to other parts of the system. For example, a checkpointing library seeks to recover a single application, but it does not seek to recover the operating system below that library [12]. Recovery systems below the operating system seek to recover the operating system and all applications [2].

Based on this view, we can classify faults as occurring *above*, *below*, or *within* the recovery system. Faults that occur *above* the recovery system directly affect the state of the program that is being recovered. That is, these faults will always lead to corrupted state in the domain being recovered, thereby making it possible for the recovery system to save corrupted state. One reason that application faults are so difficult to recover from is that they occur above the recovery system and can lead easily to corrupted state being saved.

In contrast, faults that occur *below* the recovery system often do not affect state above the recovery system. Faults below the recovery system can affect state in the domain being recovered only if the corruption propagates up through the intervening layers. Consider a user-level checkpointing library as an example. Faults in the operating system can crash the system without ever affecting the state of the application linked with the checkpointing library. If no state is corrupted in the domain being recovered, then the recovery system cannot possibly save corrupted state. It is still possible for an operating system fault to corrupt application state (e.g. by corrupting a system-call return value), but it is relatively infrequent.

Faults may also occur *within* the recovery system itself. For example, a fault within the operating system's file-system code may cause a checkpointing library to save incorrect checkpoint data. Faults within the recovery system are similar to those below the recovery system in that they may or may not affect state above the recovery system.

2.4. Quality of error detection

The final factor that affects whether or not a failure leads to corrupted state being saved is the quality of the error detection present in the system and application. One goal of error detection is to stop a faulting program *before* it saves corrupted state to stable storage. The longer the latency between fault activation and error detection, the more likely it is that corrupted state will be saved.

3. Experimental apparatus

In this paper, we evaluate the effect of varying the comprehensiveness and frequency of the recovery system. We explore both application-generic and application-specific recovery systems, and we evaluate faults below and above the recovery system. The programs we evaluate detect errors with ad hoc redundancy (primarily assertions and memory address range checks). Our general strategy is to (1) run a variety of general-purpose applications (described in Section 3.1), (2) inject faults (described in Section 3.2) into an application or the operating system until it fails, (3) recover and continue the application and system using a variety of recovery schemes (described in Section 3.3), then (4) evaluate whether or not the recovery system saved corrupted state (described in Sections 4-6).

3.1. Applications

We use three general-purpose applications to conduct our experiments: nvi, Postgres and oleo. nvi is a commonly used Unix text editor and consists of about 87,000 lines of code. Postgres is a database management system developed at U.C. Berkeley and consists of about 327,000 lines of code. oleo is a terminal-based spreadsheet program and consists of about 53,000 lines of code. While our results are specific to the software being tested, we believe that examining several general-purpose applications that are in common use can help contribute to an understanding of how faults behave in general.

Nvi is an interactive application that reads user input and displays its work on the screen. It saves output to a file when given a command by the user. We modified the application to simulate user input by reading keyboard input from a text file. This allowed us to automate the thousands of experiments used in this paper. The keyboard input we use consists of about 8000 characters that were logged while we wrote the introduction to a paper.

Postgres can run in a variety of conditions ranging from accepting input directly from the user to running in a client-server configuration. We run Postgres in the mode where it reads SQL commands from a file. The SQL commands used to drive Postgres are based on the TPC-B benchmark; each transaction updates some data in a banking database.

Oleo is an interactive application that reads user input and displays its work on the screen. It saves the state of the spreadsheet to a file when given a command by the user. We modified oleo to read its input from a file. The keyboard input we use to drive oleo replays the keystrokes used to create the budget for a grant proposal and consists of about 500 characters.

The operating system used in our experiments is FreeBSD 2.2.7, which is a version of Unix based on the BSD 4.4 kernel. We used a customized version of FreeBSD (FreeBSD-Rio) that included code to implement reliable main memory [11]. FreeBSD-Rio ensures that all file cache data is written to disk during an operating system crash, which provides the same level of data reliability as a file system that synchronously writes all file-cache data to disk.

3.2. Fault models

We evaluate the failure and recovery behavior of applications in the presence of application and operating system faults. This section describes the faults we inject into the application or operating system. The fault-injection mechanisms are described in more detail in Chandra's dissertation [3]. Our models are derived from field studies of commercial databases and operating systems [14, 13, 9] and from prior models used in fault-injection studies [1, 8, 7]. The faults we inject range from low-level faults such as flipping bits in memory to high-level software faults such as memory management errors and uninitialized variables. We trigger a fault in the running application or operating system at a random time in the program's execution. Injected faults fall into two categories: bit flips and high-level software faults.

The first category of faults flips random bits in the application or operating system's heap and stack. These faults simulate the corruption of a process's address space by wild pointers and hardware faults.

The second category of faults introduces programming errors into the application or operating system. We inject these faults by modifying the source code, compiling and running the modified source code, then triggering a fault dynamically while the program is running. We modify the source code by using a custom parser that identifies all potential pieces of code where a particular fault can be injected, then adds code to inject the fault on demand. When the fault-triggering mechanism is activated, it sets a global variable that triggers one of the instances of faulty code. A brief description of each fault type follows.

Memory-management faults: These faults affect the program's dynamic memory allocation and de-allocation. We modified the malloc() and free() routines to free a block of memory while the particular block of memory is still in use. Our modified routines keep track of allocated memory that has not yet been freed. When the fault injection routine is triggered, it prematurely frees one of the allocated blocks of memory still in use.

Off-by-one faults: These faults simulate software bugs where the programmer uses the wrong conditional operator to determine if a loop will execute another iteration or terminate. We insert these bugs by replacing the operator ">" with ">=", and by replacing the operator "<" with "<=". When a fault is injected, it sets a global variable in the program that causes the erroneous operator to be used instead of the original operator.

Initialization faults: These faults simulate software bugs where the programmer fails to initialize a variable. We insert these bugs by changing the initialization code so that a variable is initialized with zero or is left uninitialized.

Incorrect branch faults: These faults simulate software bugs where the programmer declares a conditional branch statement instead of an iteration statement. We insert these bugs by replacing the keyword "while" with an "if" statement.

Delete instruction faults: These faults simulate software bugs where the programmer forgets a simple expression statement. We insert these bugs by skipping over a simple expression statement.

Change destination variable faults: These faults simulate software bugs where the programmer assigns a value to the wrong variable. We insert these bugs by replacing a destination variable in an assignment statement with another variable.

3.3. Recovery systems

As described in Section 2, the design of the recovery system affects the likelihood that failures cause corrupted state to be saved. This section describes three recovery systems we use to evaluate these effects quantitatively. We describe how and when each of our three recovery system saves and recovers state, and we describe how we detect saved corrupted state when using each recovery system. The first recovery system exemplifies a typical application-specific recovery system, while the second recovery system exemplifies a typical application-generic recovery system. The third recovery system represents an intermediate point along the continuum between application-specific and application-generic recovery.

Application-specific recovery. Most applications provide an application-specific recovery system. This type of recovery system is written by the application programmer, who can take advantage of his/her knowledge of program semantics and user requirements to minimize both the amount of state saved and the frequency of saving that state. The scope and frequency of state saved varies for our three applications. Nvi saves the state of the file being edited when directed by the user and also saves recovery state to a temporary file when triggered by the built-in auto-save mechanism. Postgres commits the state of each

transaction to the database file at the end of each transaction. Oleo saves the state of the spreadsheet only when the user issues a save command; unlike nvi, oleo has no auto-save mechanism.

We detect when an application or operating-system failure saves corrupted state by examining the state of file data (stable storage) after the application or operating system crashes. We compare the crash state of these files with a set of reference states for these files. The set of reference states is generated by running the application without any faults and archiving the complete history of the set of files (i.e. all versions that the set of files goes through during the program's execution). After a failure, we compare the crash state of the files against each version in the reference set. If the crash state of the files matches any of the versions in the reference set, then the failure has not corrupted the data on stable storage compared to a fault-free run. For these faults, the run will complete successfully after recovery if the user continues entering input from the point represented by the matched state and the fault does not reoccur. However, if the crash state of the files does not match any of the versions in the reference set, then the failure has caused the application to commit incorrect state, and this state will be visible to the application after recovery.

Postgres includes a mechanism to abort transactions that were in-progress at the time of the crash. This mechanism rolls back changes to the database file that were not yet committed at the time of the crash. We make full use of Postgres's application-specific recovery mechanism by rolling back uncommitted transactions before comparing against the set of reference states. Prior results indicate that the transaction mechanism significantly reduces the number of failures that save corrupted state [4].

Application-generic recovery. This recovery system uses an application-generic checkpointing library (Discount Checking) to recover the application [10]. Checkpointing systems like Discount Checking are unaware of the semantics of application data and output events, and hence they must be conservative in how frequently and comprehensively they save state. Application-generic recovery systems must save all application state through checkpointing or logging at each event visible to the user (these are commonly called "output commits" [6]). We expect application-generic recovery systems to save corrupted state more often due to their higher frequency and greater comprehensiveness of saving state.

We detect when an application or operating-system failure saves corrupted state by having Discount Checking recover the application from its last checkpoint and trying to complete the program's execution. We deactivate the fault injection mechanism during recovery so that no new faults are injected. As a result, the recovering run will complete successfully if and only if the last checkpoint before the crash contains no corrupted data. As described in Section 2, an application-level fault saves corrupted state if a checkpoint (which saves all application state) is taken after the fault was activated. An operating system fault saves corrupted state if it leads to application state being corrupted, and an application checkpoint is taken after the corruption.

Low-frequency application-generic recovery. This recovery system is similar to an application-generic recovery system, but takes checkpoints only when an application-specific recovery system would save state. As with application-generic recovery, we use Discount Checking to save the complete state of the process. However, we modify Discount Checking in this system to take checkpoints only when the application-specific recovery system would save state. We detect saved corrupted state for this strategy in the same manner as we do for application-generic, i.e. by recovering from the most recent checkpoint.

This recovery system represents an intermediate point between pure application-specific and application-generic recovery systems. We measure its behavior to try to separate out the two factors (comprehensiveness and frequency) that differ between application-specific and application-generic recovery systems. We can get an idea of the effect of the increased comprehensiveness of an application-generic recovery system by comparing the low-frequency application-generic recovery system with the application-specific recovery system (since they differ only in comprehensiveness). Likewise, we can get an idea of the effect of the increased frequency of an application-generic recovery system by comparing the low-frequency application-generic recovery system with the application-generic recovery system (since they differ only in frequency).

4. Measurement methodology

Our main goal is to compare the impact of different recovery systems on the likelihood of saving corrupted state. To enable a direct comparison, we would like to compare how different recovery systems behave on the same fault. A natural way to accomplish this is to run three fault-injection experiments, where each experiment injects the same fault at the same point in execution but uses different systems to recover after the failure. However, because the random timer mechanism we use to inject faults is non-deterministic, we cannot repeatably inject the same fault at the same point in execution across multiple runs. Instead, we enable a direct comparison on the same

Fault	Faulty Runs	Runs That Save Corrupted State			Undetected Errors
		App-Specific	Low-Freq App-Generic	App-Generic	
Stack	50	0	0	0	0
Alloc	50	24	40	50	0
Heap	50	6	12	35	8
Off by One	50	6	7	9	12
Init Errors	50	0	2	2	0
Delete Branch	50	25	27	34	8
Delete Inst	50	12	14	24	3
Change Dest Var	50	1	5	8	5
Total	**400**	**74 (19%)**	**107 (27%)**	**162 (41%)**	**36 (9%)**

Table 1. Results for nvi (application faults). Only faulty runs are represented (runs that either crashed or ran to completion with incorrect results). The Undetected Errors column gives the number of faulty runs in which the fault was never detected and the program ran to completion with incorrect results. The middle columns show the number of runs that saved corrupted state and thereby could not recover correctly. The remainder of the runs crashed then recovered successfully.

fault by applying all three recovery systems during a *single* fault-injection run and evaluating separately whether each recovery system saved corrupted state for that fault. This technique allows us to compare different recovery systems on the same fault, and it also speeds up our experiments significantly. Even with this optimization, the fault-injection experiments for this paper took many machine-months.

The following describes how we use a single fault-injection run to evaluate all three of our recovery systems. We run an experiment with the application linked with Discount Checking saving state at the higher frequency. We detect when application-generic recovery saves corrupted state simply by having Discount Checking recover from the most recent checkpoint. We also measure when low-frequency application-generic recovery saves corrupted state from that same experiment by having Discount Checking recover from the checkpoint that would have been taken most recently under a low-frequency recovery system. We measure when application-specific recovery saves corrupted state from that same experiment by examining the crash state of the files as described in Section 3.3.

In each fault-injection run, there are three possible outcomes. First, the program may finish execution with the correct final result. This outcome indicates that the fault injected in this experiment did not cause an error, either because the fault was not activated or because the fault had no lasting effect (e.g. it changed a variable that was overwritten before being read). We are interested in evaluating the behavior of failures, so we discard these runs. Second, the program may finish execution but have the wrong final result. This outcome indicates that the system's and program's error checking overlooked the failure. Because these runs are not detected as errors by the normal error detection mechanisms, they do not trigger recovery. Like the runs that save corrupted state, these runs prevent the program from finishing correctly (i.e. they finish incorrectly). Third, the program or operating system may fail and recover. We are primarily interested in this third outcome, as it is the only outcome whose behavior depends on the recovery system. We use the methods described in Section 3.3 to evaluate whether a run with this outcome saves corrupted state (and hence cannot recover).

5. Results of application-level faults

We first measure the behavior of faults originating in the application. As mentioned above, we only consider runs that finish with a wrong final result or that fail before finishing. About 3-4% of all the runs into which we inject faults fall into one of these two categories. For each application, we repeat the fault-injection experiments until we obtain 50 faulty runs (i.e. failures or undetected errors) for each fault type. This gives us a total of 400 data points for each of the three applications.

Table 1 shows the results for nvi. 9% of the runs we consider complete incorrectly without triggering the error-detection mechanism. These runs are shown in the right-most column of all tables. For runs in which an error was

Fault	Faulty Runs	Runs That Save Corrupted State			Undetected Errors
		App-Specific	Low-Freq App-Generic	App-Generic	
Stack	50	0	16	17	1
Alloc	50	0	22	24	0
Heap	50	0	0	44	2
Off by One	50	0	0	0	8
Init Errors	50	0	2	3	2
Delete Branch	50	0	0	38	6
Delete Inst	50	1	2	6	5
Change Dest Var	50	2	2	3	0
Total	**400**	**3 (1%)**	**44 (11%)**	**135 (34%)**	**24 (6%)**

Table 2. Results for Postgres (application faults).

Fault	Faulty Runs	Runs That Save Corrupted State			Undetected Errors
		App-Specific	Low-Freq App-Generic	App-Generic	
Stack	50	0	0	3	0
Alloc	50	0	2	34	9
Heap	50	0	0	12	19
Off by One	50	0	0	10	7
Init Errors	50	0	3	15	8
Delete Branch	50	0	0	19	7
Delete Inst	50	0	2	9	18
Change Dest Var	50	3	3	5	20
Total	**400**	**3 (1%)**	**10 (3%)**	**107 (27%)**	**88 (22%)**

Table 3. Results for oleo (application faults).

detected, many still saved corrupted state and hence could not recover successfully. Application-specific recovery had the lowest fraction of runs that saved corrupted state (19%). The fraction of runs that saved corrupted state increased to 41% when using an application-generic recovery scheme. This dramatic increase is due to the increased comprehensiveness and frequency of saving state when using an application-generic recovery system. We attempt to isolate the effects of these two factors (comprehensiveness and frequency) by using the low-frequency application-generic recovery system. Under this recovery system, 27% of faulty runs do not complete successfully, which is slightly closer to the application-specific results than to the application-generic results. Lowering the frequency at which state is saved significantly reduces the fraction of runs that do not complete correctly, even if the recovery system still saves all application state.

Table 2 shows the results for Postgres. The fraction of runs that saved corrupted state is much smaller than for nvi. We attribute this to two factors. First, Postgres has very thorough error detection, in part because its application domain (databases) places a high premium on stopping the system before data integrity is compromised. Second, Postgres updates data within atomic transactions,

Fault	Faulty Runs	Runs That Save Corrupted State			Undetected Errors
		App-Specific	Low-Freq App-Generic	App-Generic	
Stack	50	0	1	6	0
Alloc	50	1	5	19	0
Heap	50	2	3	4	0
Off by One	50	0	6	11	0
Init Errors	50	3	2	8	1
Delete Branch	50	1	2	12	0
Delete Inst	50	0	1	6	0
Change Dest Var	50	2	0	5	0
Total	**400**	**9 (2%)**	**20 (5%)**	**71 (18%)**	**1 (0%)**

Table 4. Results for nvi (operating system faults).

and this allows Postgres to often roll the computation back to a point before the fault was injected (when using application-specific recovery). Postgres' application-specific recovery only saves corrupted state 3 times in our experiments. However, note that the number of runs that save corrupted state increases from 3 to 135 when switching from application-specific to application-generic recovery! As with nvi, lowering the frequency of saving state gains much of the benefit of application-specific recovery. This is because the frequency of saves in Postgres is very small, thus there is a high probability that the fault was triggered after the last state commit. In these cases, no state is corrupt at the time of the last checkpoint, and increasing the comprehensiveness of state saved cannot cause corrupted state to be saved.

Table 3 shows the results for oleo using our three recovery systems. oleo's application-specific recovery saves corrupted state only 3 times in our experiments. As with Postgres, we attribute this to the low frequency of saving state in oleo—state is saved only when the user issues a save command. This view is supported by the results with low-frequency application-generic recovery, which shows that saving a comprehensive set of state increases only slightly the runs that saved corrupted state, as long as the state is saved at a low frequency. However, the higher frequency of saving state under application-generic recovery dramatically increases the fraction of runs that save corrupted state to 27%. For oleo, a very large fraction of faulty runs (22%) complete incorrectly without triggering any error-detection mechanism. Apparently this application has very sparse error detection. This large number of undetected errors limits the effectiveness of any recovery scheme.

6. Results of operating system faults

We next measure the behavior of faults injected into the operating system. As mentioned above, we only consider runs that finish with a wrong final result or runs in which the target program or operating system fails before the application finishes. About 10% of all the runs into which we injected faults fell into one of these two categories. We repeat the fault-injection experiments till we obtain 50 candidate runs for each fault type per application. This gives us a total of 400 runs for each of the three applications. Before recovering the program from a failure, we first reboot a version of the operating system without any of our injected faults; this ensures that any failures after recovery are due to corrupted state that was saved before the failure. We also reboot before each run.

Table 4 shows the results for nvi when faults are injected into the operating system. Note that the fraction of runs that save corrupted state (for all three recovery systems) is much lower when faults are injected into the operating system than when faults are injected into the application. The fraction of runs that save corrupted state with application-specific recovery drops from 19% to 2%; the fraction with application-generic recovery drops from 41% to 18%; and the fraction with low-frequency application-specific recovery drops from 27% to 5%. These big drops in the number of runs that save corrupted state are due to the fault being below the recovery system, as discussed in Section 2.3. For an operating system to save corrupted state in the application, it must first leak out of the operating system and corrupt the state of the application. The primary interaction path between an application and the operating system is the system-call interface. These

Fault	Faulty Runs	Runs That Save Corrupted State			Undetected Errors
		App-Specific	Low-Freq App-Generic	App-Generic	
Stack	50	0	5	5	0
Heap	50	1	3	3	0
Off by One	50	0	0	0	0
Init Errors	50	0	0	0	0
Delete Branch	50	1	2	2	0
Delete Inst	50	0	1	2	0
Change Dest Var	50	0	0	0	1
Total	**350**	**2 (1%)**	**11 (3%)**	**12 (3%)**	**1 (0%)**

Table 5. Results for Postgres (operating system faults). No results are reported for Alloc faults because running the Postgres workload with operating system faults did not trigger any crashes or undetected errors.

Fault	Faulty Runs	Runs That Save Corrupted State			Undetected Errors
		App-Specific	Low-Freq App-Generic	App-Generic	
Stack	50	4	0	3	0
Alloc	50	0	0	0	0
Heap	50	1	1	1	0
Off by One	50	3	0	0	0
Init Errors	50	0	1	1	0
Delete Branch	50	1	3	4	0
Delete Inst	50	5	0	1	0
Change Dest Var	50	3	4	4	0
Total	**400**	**17 (4%)**	**9 (2%)**	**14 (3%)**	**0 (0%)**

Table 6. Results for oleo (operating system faults).

interfaces are narrow and are used less frequently than normal application instructions, so it is common for a faulty operating system to crash before corrupting application state. However, note that comprehensive and frequent saving of state, as is done in application-generic recovery, dramatically increases the likelihood of saving corrupted state, even when faults are injected below the recovery layer.

Table 5 shows the results for Postgres when faults are injected into the operating system. As with nvi, operating-system faults cause fewer undetected application errors and runs that save corrupted state than application-level faults. As with application-level faults, operating-system faults save corrupted state less frequently for Postgres than they do in nvi, which is probably due to Postgres's more robust error detection. For Postgres, application-generic recovery and low-frequency application-generic recovery both cause a similar increase in the number of runs that save corrupted state when compared to application-specific recovery. One reason for the relatively small increase for application-generic recovery is that our Postgres workload makes fewer system calls than our input-intensive nvi workload (65,000 versus 147,000), and this limits the

opportunity for operating-system faults to corrupt application state.

Table 6 shows the results for oleo when faults are injected into the operating system. The results are similar to the other applications in the reduction of undetected errors and runs that save corrupted state relative to application faults. One anomaly in the results for oleo is that application-specific recovery actually saves corrupted state slightly more frequently than when using application-generic recovery. Because application-generic recovery saves state more frequently and comprehensively than application-specific recovery, our intuition is that all runs that save corrupted state under application-specific recovery should also save corrupted state under application-generic recovery. After investigating the runs that contradicted this intuition, we discovered that, for each of these runs, the corrupted state being saved in application-specific recovery is caused by an operating-system error when oleo is writing the files used by application-specific recovery. In contrast, application-generic recovery uses Discount Checking, which saves state via a memory-mapped file, and this interface uses less operating-system functionality than direct file system operations. Hence the lower number of corrupted state saves in application-generic recovery is due to a more robust method of saving state compared to application-specific recovery. We also see this effect in a few runs of nvi (Table 4, Init errors and Change Dest Var).

7. Related work

While many researchers have designed and implemented various recovery schemes, we know of none who has evaluated how the type of recovery system changes the likelihood of corrupted state being saved (and hence changes the likelihood of recovery completing successfully).

An earlier paper by Chandra and Chen [4] presented an initial study on how often software faults violate the fail-stop model (by saving corrupted state) for a single application (Postgres) under application-specific recovery. The methodology and scope of the earlier study differs from the current paper. The earlier study evaluates fail-stop behavior only for application-level faults and uses only application-specific recovery, whereas the present paper evaluates faults in the operating system and application, and uses three different recovery systems. The earlier study is also less accurate at determining when a run saves corrupted state, because it takes only periodic snapshots of the file state, whereas the current paper archives the complete history of the set of application files.

Costa, et al. evaluated the impact of hardware and software faults in the Oracle database [5]. They found that 1% of the runs resulted in data corruption, i.e. data that did not match the TPC-C consistency checks for their workload. This figure is a percentage of all runs, most of which did not lead to a failure; the percentage of faulty runs that resulted in data corruption would be several times higher. These experiments represent data corruptions while using application-specific recovery. Our research differs from this research by investigating the effect of other recovery systems, as well as by investigating several applications, and including operating system faults.

Lowell, et al. explore generic recovery and describe two invariants needed to ensure that recovery is possible and transparent [10]. The paper uses some of the experiments included in the present paper to evaluate how often these invariants can be upheld simultaneously. The present paper differs by measuring and comparing both application-specific and application-generic recovery, as well as an intermediate point between these styles of recovery systems.

8. Conclusions

Our goal was to study how a recovery system's mechanism for saving state affects the likelihood that the system will save corrupted state and thereby prevent successful recovery. Because recovery systems depend on the state they save, any corruptions to that state will prevent recovery from succeeding. Generic recovery systems that work for general applications are especially vulnerable, because their lack of knowledge about the application causes them to save a more comprehensive set of state at a higher frequency than what is needed to recover a specific application. This increased frequency and comprehensiveness of saving state increases the likelihood that they will save the state of the fault itself. Our experiments compared application-specific and application-generic recovery and also examined the separate effects of increased comprehensiveness and increased frequency.

For application-level faults, we found that using the error detection and recovery built into the application caused 1-19% of faulty runs to save corrupted state. Using a generic recovery system increased the fraction of runs that saved corrupted state to 27-41%, with more of the increase due to a higher frequency of saving state than to a more comprehensive set of state being saved. This high frequency of saving corrupted state calls into question the viability of using generic recovery to survive application-level software faults.

Results were more encouraging when we considered faults below the recovery system, such as faults in the operating system. For these faults to save corrupted state, they must first propagate into state saved by the recovery system. Our results showed that only a few percentage of

operating-system faults saved corrupted state for most applications and recovery systems.

These results shed light on the relationship between the state-saving methods used by recovery systems and the likelihood that recovery will complete successfully. Our hope is that this work will encourage the design of better fault detection and recovery mechanisms. Recovery mechanisms should take care to minimize the chance of saving corrupt state. Future recovery systems may require application knowledge to save less state less frequently. We believe a fruitful area for future research to be hybrid application-specific / application-generic recovery systems that can accomplish this without undue burden on the application programmer.

9. References

[1] James H. Barton, Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, April 1990.

[2] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-Based Fault-Tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[3] Subhachandra Chandra. *An Evaluation of the Recovery-Related Properties of Software Faults*. PhD thesis, University of Michigan, September 2000.

[4] Subhachandra Chandra and Peter M. Chen. How Fail-Stop are Faulty Programs? In *Proceedings of the 1998 Symposium on Fault-Tolerant Computing (FTCS)*, pages 240–249, June 1998.

[5] Diamantino Costa, Tiago Rilho, and Henrique Madeira. Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection. In *Proceedings of the 2000 International Symposium on Fault-Tolerant Computing*, June 2000.

[6] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[7] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.

[8] Wei-Lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, November 1993.

[9] Inhwan Lee and Ravishankar K. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 20–29, 1993.

[10] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–304, October 2000.

[11] Wee Teck Ng and Peter M. Chen. The Design and Verification of the Rio File Cache. *IEEE Transactions on Computers*, 50(4):322–337, April 2001.

[12] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Winter 1995 USENIX Conference*, pages 213–224, January 1995.

[13] Mark Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability–A Study of Field Failures in Operating Systems. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, June 1991.

[14] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, pages 475–484, July 1992.