IMPROVING SOFTWARE FAULT TOLERANCE IN HIGHLY AVAILABLE DATABASE SYSTEMS

by

Mark Sullivan and Michael Stonebraker

Memorandum No. UCB/ERL M90/11

8 February 1990

IMPROVING SOFTWARE FAULT TOLERANCE IN HIGHLY AVAILABLE DATABASE SYSTEMS

by

Mark Sullivan and Michael Stonebraker

Memorandum No. UCB/ERL M90/11

8 February 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

IMPROVING SOFTWARE FAULT TOLERANCE IN HIGHLY AVAILABLE DATABASE SYSTEMS

by

Mark Sullivan and Michael Stonebraker

Memorandum No. UCB/ERL M90/11

8 February 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

1. Introduction

Today, software errors rank with hardware and operator errors as a source of failures in fault tolerant systems [13]. However, current trends may soon make software errors the most common cause of failures. Redundancy can greatly limit hardware failures. Better user interface designs should reduce the number of accidents caused by operators. On the other hand, the demand for increased functionality is making systems programs like the data manager and operating system larger and more complex. The advent of shared memory multiprocessors may make concurrency-related errors more common. Even the increased speed of processors may work against software; more system capacity can allow more concurrent users and more ambitious demands on the system. This paper considers several ways to reduce the impact of software errors on reliability and availability.

In order to reduce the loss of reliability due to software errors, we must address the issue of **error latency**. Current transaction processing systems (TPS) assume that the errors causing system failure are **fail-stop**. When a fail-stop error occurs, it damages transient system state (pages in main memory) and quickly halts the system. Techniques for repairing the system after these errors are well-understood [15].

The fail-stop error model, however, represents only part of the failure threat facing a real system. Most hardware and all power failures are fail-stop, but most system software errors damage transient state without stopping the TPS. The broken system eventually detects its damage and tries to recover, but only after a delay (the error latency). A latent software error might propagate damage to permanent state, making recovery impossible. The delay between error occurrence and damage to permanent state is termed **propagation latency**.

Traditionally, commercial systems have increased software fault tolerance by improving the quality and frequency of software error checking [13]. Detecting errors quickly decreases error latency, hence limits the chance that an error causes permanent damage before it is detected. Unfortunately, adding software checkers to a system is expensive. Individual checkers must be designed for each data structure and algorithm, and running them frequently increases processor load.

Another approach to increasing software reliability is to limit the points at which errors can propagate to permanent storage. Increasing propagation latency can be as effective as decreasing error latency

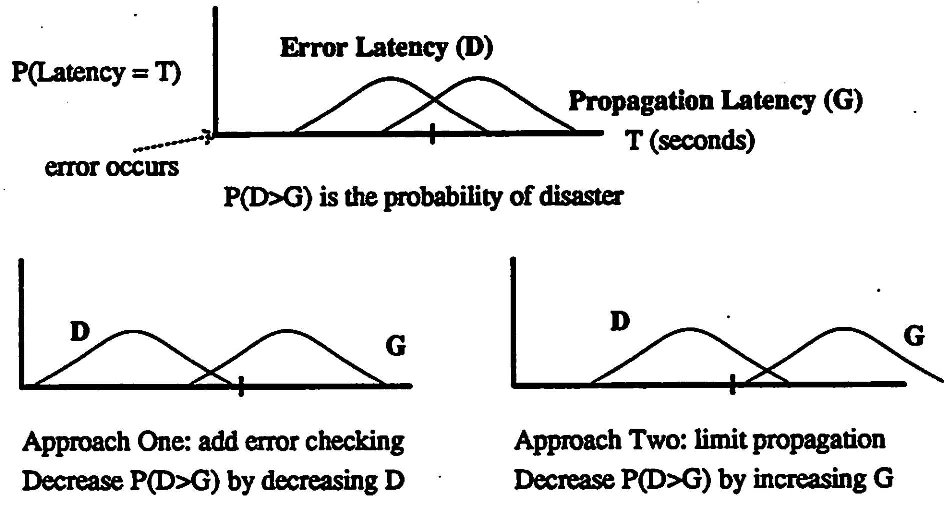

P(Latency = T)
Error Latency (D)
Propagation Latency (G)
T (seconds)
error occurs
P(D>G) is the probability of disaster
D
G
D
G
Approach One: add error checking
Decrease P(D>G) by decreasing D
Approach Two: limit propagation
Decrease P(D>G) by increasing G

Figure 1: Two Approaches to Increasing Software Fault Tolerance

for reducing the chance of disaster (see figure 1). Instead of checking more often, we prevent damaged data from becoming permanent until existing error checkers have had an opportunity to examine it. Unlike error checkers, the the techniques used to increase propagation latency are not tied to particular data structures, so they do not need to be redesigned as the system evolves.

In this paper, we are going to discuss several fault tolerance techniques which help make the existing error checkers more effective. These techniques include:

(a) **Page Guarding**: write-protecting data not currently in use (e.g. disk cache) can decrease the amount of data the checkers need to consider.

(b) **Deferred Write**: applying updates to copies of data records instead of the real data records can help keep transactions from propagating errors to one another. Broken transactions have a chance to detect their errors before writing shared data.

(c) **Delayed Commit**: preventing a transaction's effects from becoming permanent for a short time after the transaction has completed increases propagation delay. The delay gives latent OS errors a chance to materialize before the transactions they affect are committed.

TPS simulations have been used to estimate the performance impact of these techniques. The simulations indicate that none of these mechanisms reduce throughput by more than seven percent. Assessing reliability has proved more difficult. In order to analyze reliability, errors are grouped into classes. The model of errors emphasizes the concurrency management, recovery management, and pointer errors. Field measurements indicate these three classes cause the most trouble for existing systems. The techniques are then rated, in part, according to the error classes against which they are effective. Analytic models based on different assumptions about errors also help determine reliability effects.

The second part of the paper addresses availability. When recovery speed increases, the amount of system downtime after a failure goes down. In order to make recovery fast, we must reexamine the decision to use disk instead of non-volatile memory for permanent storage. By storing the tail of the transaction log in main memory, the system can avoid going to disk during transaction commit. Non-volatile memory improves recovery speed by making it unnecessary to reload the disk cache after a failure and reducing the number of disk accesses required for log processing. However, memory-based permanent storage faces an increased risk from software errors. In this section, we discuss the software reliability differences between memory and disk storage. Non-volatile main memory can be protected against software errors using many of the same techniques we use to decrease the impact of software errors.

The paper has six sections. We describe our model of the TPS and model of errors before introducing the fault tolerance techniques. An evaluation section discusses simulations, reliability models, and published observations about errors in existing systems. The design of a DBMS with a main memory log is outlined in the fourth section. The fifth section gives related work. We conclude with a summary of results to date and an outline of several possible research directions.

2. TPS Model: Definitions and Assumptions

The TPS runs on a shared memory multiprocessor such as Spur [16]. It includes three kinds of programs: applications (front-ends), the database management system (DBMS), and the operating system (OS). Figure 2 shows the process structure of the TPS. Each application program runs as a single process. Each application communicates with a single DBMS process over a two-way message channel (UNIX pipe). DBMS processes share a region of main memory that contains a disk cache (buffer pool) and a lock table. Assume that the shared memory region in the DBMS is fixed size and cannot be paged out by the OS.

A transaction is initiated by an application process and runs in one of the DBMS processes. Each transaction operates on a small set of data records (its **operands**), some of which will be written. Records are small (about 100 bytes) and reside in pages (4K or 8K bytes) which are read in from disk on demand by the buffer manager. Most data records are rarely rereferenced and are removed from main memory after a few seconds (on an LRU basis). Others are referenced often and rarely if ever leave main memory.

Before completing, the transaction produces a set of **log records** and copies them to recovery store. Recovery store, usually an area on magnetic disk, holds all information used to reconstruct the system after a failure and must be assumed safe from errors. After a system failure, the log records from committed transactions are used to reproduce the values of operands written by the transactions.

Improving Software Fault Tolerance in Highly Available Database Systems

Mark Sullivan
Michael Stonebraker

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

February 8, 1990

Abstract

Software errors often damage the transient state of a transaction processing system (TPS) without causing the system to fail immediately. We propose several techniques to increase the chance of detecting latent software errors before disaster occurs. The same techniques can improve recovery speed by making non-volatile memory a more practical medium for permanent storage. These techniques include:

(1) using hardware write protection to guard data in the database buffer pool from errors,

(2) using a shadow-paging scheme to reduce the chance that an erring transaction propagates errors to correct pages,

(3) inserting an artificial delay between the time a transaction completes its work and the time it is considered committed. Because of the delay, errors may remain undetected for a longer time without causing irrecoverable damage.

Simulations show these techniques reduce transaction throughput by as little as one to seven percent. An analytic model estimates reliability improvements given several possible models of errors. Our proposal also outlines the software fault tolerance concerns in designing a data manager that writes log records to non-volatile memory on commit instead of disk.

This work is supported by National Science Foundation grant 87-15235

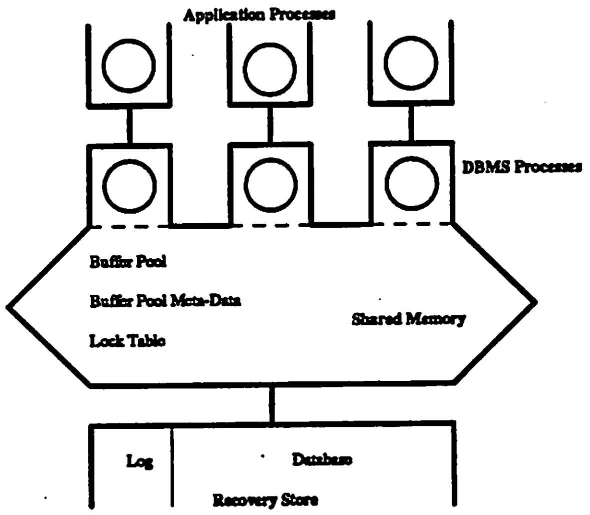


Figure 2: TPS Process Structure

At various points during the transaction, the TPS programs will check their data for consistency using standard fault tolerance techniques (array bounds checks, check back pointers in linked lists, etc.). These consistency checks are referred to collectively as the **transaction error checker**. Where possible, the error checker is assumed to run at the end of the transaction, immediately before commit. If the error checker detects an inconsistency, it will either abort the transaction or cause the entire system to recover.

An **error** is an event that causes the TPS to write inconsistent values to data records. Error events include:

(a) processor/device failures,

(b) propagated software errors which damage transient data (e.g. the process stack) and indirectly damage data records,

(c) software errors in system software (DBMS and OS),

(d) software errors in applications which misdirect the system software.

An **error cluster** is a collection of error events happening over a short time. The errors in a cluster are caused by some outside event (hardware error, system load passes certain threshold, user repeatedly attempts the same bad operation). Error events cause the running transaction to either write an inconsistent value into a record (**wrong value errors**) or write to the wrong record (**wrong record errors**).

Wrong record errors include operand, pointer, and meta-data errors. In **operand errors**, erring software causes the transaction to use the wrong record as an operand. An operand error might cause a transaction that is supposed to increase all employee's salaries by ten percent to mistake several items from the inventory for employees and give them a raise too. **Pointer errors** happen when a transaction writes to a data record that is not an operand for the current transaction. An uninitialized pointer or a array bounds overflow fit into this error class. **Decision errors**, a special case of pointer errors, will be described in the next section. **Meta-data errors** happen when the data structures used to find an operand are damaged. For example, the operating system could damage a DBMS process page table causing the DBMS to use data on the wrong page.

Wrong value errors happen when the transaction uses correct data but makes inconsistent changes to records anyway. For example, a numerical error that causes fifty dollars to disappear from a bank balance would be a wrong value error. Two special cases of wrong value errors are **abort errors** and

Term	Description
Operand (Wrong Record)	Application asks for wrong record
Pointer (Wrong Record)	Uninitialized pointers
Decision (Wrong Record)	Defined in section 3
Meta-data (Wrong Record)	Operand name misresolved
Wrong Value	Correct record, bad update
Abort (Wrong Value)	records not restored after abort
Synchronization (Wrong Value)	locking protocol violated

Table 1: Error Classes

Term	Description
Log error	Log records damaged
Data error	Cache data damaged, but not log
Recovery store error	Recovery meta-data damaged

Table 2: Disaster Causes

synchronization errors. An abort error happens when a transaction aborts without cleaning up any writes it has made. Synchronization errors happen when the locking protocols or other resource allocation protocols are violated (e.g two writers on a single record, memory leak).

An error causes disaster if a damaged record cannot be restored to a consistent value during recovery. There are three ways in which errors can cause disasters. A **log error** causes disaster by damaging the transaction log. If the error is not detected before transaction commit, the TPS will not have a correct record to use in recovery. A miscalculation that caused an employee's salary to become negative would probably be a log error since the erring transaction would record the illegal value in the log. **Data errors** damage data records but are not reflected in the transaction log. For example, an uninitialized pointer may cause the DBMS to overwrite a record in the cache. These errors cause disaster only when the damaged record is flushed out of the main memory cache. A **recovery store error** is an error that causes good data in recovery store to be overwritten. Such an error occurs when, for example, the transaction log for one transaction is written over top of the log for another.

Tables 1 and 2 summarize the six error classes and three kinds of disasters. Errors having to do with bad output to the terminal are considered log errors (wrong value), since they cause disaster in a way similar to log errors.

3. Proposal: Five Software Fault Tolerance Techniques

3.1. Page Guarding

Several factors work to increase the size of a program like the data manager. First, modules with very different functionality are often combined into a single address space to increase intermodule communication speed. Modules in the same address space can access services provided by one another with an inexpensive procedure call. If they were in separate address spaces, communication cost would include context switch overhead. Second, much of the DBMS or OS address space is devoted to disk cache. If main memories become larger and disk access speeds remain constant, the disk cache can be expected to grow.

Unfortunately, access to large amounts of data becomes a liability when the program malfunctions. An uninitialized pointer or array out-of-bounds error can damage data unrelated to the code causing the error. Worse, reliability concerns can override the need to add functionality or improve performance by discouraging system designers from adding new modules (e.g. new access methods or network protocols). Further, much of the address space is rarely referenced. Measurements from a VAX 11/780 show that the mean interreference time for a byte of the 4.2 BSD UNIX kernel is 45 minutes at peak load [7]. The mean interreference time is four hours during off-peak times.

The page **guarding** fault tolerance technique uses existing memory management hardware to restrict access to this unneeded data. In page guarding, the operating system makes some data pages unwritable by clearing the write-access bit for the page in the (DBMS) process's page table. Software errors that cause a transaction to write to these protected pages will be trapped by hardware. The DBMS can change the protection for the page as needed through *guard* (write-protect) and *unguard* (unprotect) system calls. In order to update a page, the DBMS must *unguard* the page, modify the data, and *guard* the page again.

3.1.1. Costs and Benefits

The exact cost of updating a guarded page will depend on the TPS hardware. The *guard* and *unguard* operations must be system calls since they change the access bits in the DBMS process's page table. *Guard* and *unguard* must update the system's memory management unit (MMU) to reflect the new page protection status. Guarding or unguarding several pages at once will have roughly the same cost as guarding or unguarding only one. The cost is dominated by the cost of the system call and the MMU update -- costs which are unrelated to the number of pages guarded or unguarded. Performance would be best if the DBMS postponed reguarding the pages written by a single transaction until transaction commit. The transaction would make an *unguard* call the first time each page is written and a single *guard* call immediately before commit. Shifting page protections is particularly expensive in a shared memory multiprocessor, since updating the MMU usually requires cooperation from all processors. In the Mach "shootdown" algorithm, the cost is 1 to 3.5 ms for a transaction processing workload on 9 processor Sequent [4].

Guarding presents a model for large system programs that is between the multiple disjoint address spaces model and the single address space model in cost, reliability, and ease of use. For one module to access data owned by another, it makes a procedure call and unguards the data. Shifting protection boundaries with *guard/unguard* calls is cheaper than making the full context switch required in the multiple address space model. Once the data has been unguarded, access cost is that of normal data in a single address space. The guarding model makes system design easier because data belonging to different system modules can be grouped together in a single protection region when needed. A system program partitioned into multiple address spaces is obviously much less flexible.

Because guarding is a passive method of error detection, it will be particularly cost effective for protecting large disk caches. The error latency for pointer errors that damage cache data is reduced to zero. Furthermore, any errors detected would probably not be detected at all in a system without guarding. The transaction error checker could not possibly examine all data in the cache for errors.

3.1.2. Limits to the Effectiveness of Guarding

Unfortunately, guarding only protects against pointer errors. Even then, at any given time, some data records have to be unguarded so they can be updated. These records can still be damaged by pointer errors. Not only is a record being updated at risk, but so are many of the records near it. The unit of protection must be at least as large as the page size imposed by the system hardware. Updating one record requires unguarding all of the records on the page.

System designers face a performance/reliability tradeoff regarding the length of time pages are unguarded and the cost of guarding. Reguarding a page immediately after it is written decreases the opportunity for erring software to write the page. On the other hand, leaving the page unguarded until commit time can decrease the number of *guard* system calls a transaction makes.

3.1.3. Unguard Keys: Restricting the Right to Unguard Data

A second reliability problem concerning guarding is that bad software can unguard pages just as easily as good software. In the simple guarding model, writing to data at a particular address involves two decisions: the decision to unguard the data, and the decision to write to it. If an error affects the software making both decisions, guarding is useless. We are going to call errors that make both decisions incorrectly **decision errors.**

The problem of decision errors can be illustrated with this rather extreme example: Suppose the system designer wanted to reduce the number of unguarded pages in the system to one per processor. The designer changes the compiler so that every store instruction it produces is surrounded by unguard() and guard() system calls so that only the page being written is ever unguarded. As intended, no software errors will ever be able to damage pages other than the current one. Sadly, though, all pointer errors are now decision errors. No software errors can affect the decisions to unguard and to write a page independently.

Part of the concern about decision errors is the any part of the DBMS can make an unguard call. Even if the program makes careful checks before unguarding, out of date or broken code can avoid the checks. Associating unguard keys with regions of guarded memory can limit the places at which decision errors might occur.

An unguard key is essentially a capability [11] for making unguard calls. In the DBMS, an unguard key works as follows: during initialization, the DBMS guards one or more contiguous region of virtual memory with a system call (GuardRegion(high,low)). GuardRegion() returns an unguard key, a capability that gives its holder the right to unguard pages in the region. Subsequent calls to unguard() fail if the caller does not pass the correct key as a parameter.

In the DBMS, the buffer pool would have several unguard keys -- one for the data region, one for the pinned catalogs, one for each type of pinned index. If a transaction wanted to unguard a data page, for example, it would have to go through the buffer manager, since only the buffer manager would have the key. If a caller tried to unguard buffer manager meta-data or one of the system catalogs, the buffer manager would refuse. If another part of the DBMS tried to unguard data, the call would fail since it cannot produce the buffer pool key. The increase in cost is negligible. The OS can verify that a key is correct in a few instructions.

3.2. Deferred Write: Shadow Copy Limits Propagation

Guarding helps protect unused or read-only data from current transactions. It is no help in protecting one transaction's unguarded pages from another transaction's errors. If the DBMS is broken enough to ignore the locking protocol, one transaction can overwrite pages unguarded by another.

The deferred write fault tolerance technique helps address these concerns. As in the last technique, all pages are write protected when not in use. Instead of unprotecting a record to be written, however, the transaction makes a scratch copy of the record in writable virtual memory. At commit, the transaction uses the *restore* system call to copy the updated scratch record onto the original. During *restore*, the OS unguards the page containing the original, performs the copy, and reguards it.

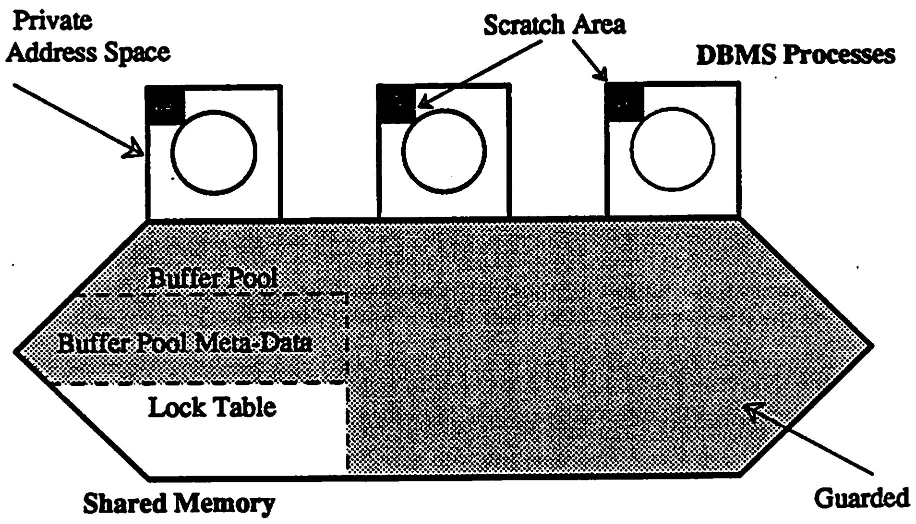


Figure 3: TPS Process Structure

The cost of deferred write will differ from simple page guarding in several respects. First, one system call is required to replace the old record with the new instead of two (there are still two TLB flushes). Second, deferred write must copy the record twice: once to make the scratch copy and once during *restore*. Third, extra memory is required to store the scratch copies. Added memory use may increase the system's page fault rate. Again, the system call costs can be reduced if all of the modified records are restored at once. In deferred write, all pages written can be unguarded *and* guarded in a single system call. For the page guarding technique, each page had to be unguarded with separate system calls.

3.2.1. Costs and Benefits

If we look again at the process model for the DBMS (figure 3), we can see how deferred write helps insulate transactions from one another. The DBMS processes share the write protected data in the shared memory buffer pool. Writable scratch copies of records are in individual DBMS address spaces. Thus, if one DBMS process makes an error, it normally cannot hurt data belonging to another process. One transaction can only propagate errors to another transaction by:

(a) committing without detecting the error,

(b) damaging the write-shared lock table,

(c) damaging the buffer pool meta-data (e.g. reallocating buffers incorrectly),

(d) damaging disk files,

(e) violating the two phase locking protocol (e.g. copy before acquiring a lock).

So, for example, if the transaction writes to the wrong record, the entire transaction has to complete successfully before the error propagates. This will help ensure that existing error checkers have a chance to detect an error before any damage either propagates or is committed. With page guarding, the erring transaction would unguard the wrong page and propagate the error immediately.

Deferred write will also make unguard keys more effective. Since data is only unguarded after the modifications are ready, the buffer manager can check over the proposed update before allowing it to proceed. If the buffer manager has semantic information about the data to be modified, it can examine the before and after image of the record and decide if the update makes sense. It can, for example, check that a modification to the system catalog make sense, or check that a data record update hasn't changed the record's unique object identifier. When *restore* only happens at the end of a transaction, the buffer manager can check that the transaction has produced correct log records before allowing the updated data to be installed.

Deferred write has several other advantages over page guarding:

(1) The TLB consistency problem in shared memory multiprocessors may be simplified. Pages are unguarded only during *restore* and only one processor's MMU need reflect the change.

(2) The DBMS does not have to know the page size of the underlying hardware. It copies and restores records of whatever size is convenient.

(3) The page-level granularity of write protection causes less trouble for deferred write than simple page guarding. Because scratch records are kept on a separate page from the one containing the original data, other records in the original page are only unguarded during the *restore* system call.

(4) Deferred write helps ensure that each *unguard* is followed by a *guard*. In page guarding, the transaction might forget to reguard the pages it has unguarded.

(5) Aborting transactions may be less likely to result in error. Since transactions never modify real data until commit, there is little to undo when the transactions abort.

Deferred write does have the disadvantage that pointers to the original record must be thrown away after a writelock is obtained. All software must have enough indirection that the scratch copy is always used in place of the original.

3.2.2. Synchronized Write: Using Deferred Write to Check Synchronization

Deferred write introduces several potential errors not present in simple page guarding. The transaction must now ensure that:

(1) updated data is returned to the same place it came from,

(2) all scratch records modified by a transaction are restored, and

(3) the granularity of the copy is no larger than the granularity of locking.

Deferred write also changes the nature of synchronization errors. The *restore* system call serializes all updates to a given record, even illegal ones. If a locking protocol error causes two transactions to write same record, one update will be lost, but the two transactions cannot intermingle writes.

Because of the enforced serialization in *restore*, a technique called **synchronized write** can help detect synchronization errors. In synchronized write, the writer takes a checksum of its scratch record before the record is modified. Here, checksum could simply be a parity calculation. At *restore*, a checksum of the original record is taken. If the new checksum is not the same as the stored checksum, either we are not restoring the record to the place it came from or someone else has written the data since the writer copied it. The cpu overhead for synchronized write can be less than twice that of deferred write since the parity calculation will be cheaper than copy.

3.3. Delayed Commit: Trade Response Time for Reliability

Most software fault tolerance measures, including the ones we have discussed so far, are designed to help detect errors quickly. Deferred write is an exception. It ensures that an error in one transaction cannot propagate before the transactions error checker has a chance to run. Our last fault tolerance technique, delayed commit, is another means of increasing the opportunity to detect errors.

In delayed commit, an artificial delay D is inserted between the time a transaction completes and the time it is considered committed. When a transaction commits, it puts a timestamp in its log records, writes the log records to disk, releases its locks, and goes to sleep. After D seconds, the system writes a second timestamp to the log and the transaction commits. During recovery, the system reads the log to find L, the latest timestamp before the failure. Only transactions with timestamps less than L-D are recovered, effectively increasing the propagation latency by D. If another transaction's log records are ready to go to the disk, the second timestamp will be unnecessary.

Delayed commit allows the delayed transaction to take advantage of error checking in other transactions and in the OS. It helps when

(a) an error strikes the transaction's data after the transaction has finished error checking it (e.g. the checker itself might have bugs),

(b) an OS error damages the transaction's data in a way the transaction alone could not detect (e.g. synchronized write detects errors after the offending transaction has committed),

(c) one transaction using a shared data structure has less thorough error checking than others,

(d) errors occur in clusters. Several errors might be propagated from the same source. If any error from the cluster causes system failure, none of the errors cause disaster.

Delayed commit can also decrease error checking overhead by allowing many independent transactions to be error checked together. Suppose the DBMS checks system catalog pages for structural integrity. These checks can happen in the background rather than as part of every transaction.

Delayed commit will obviously increase the response time of transactions by D seconds. In most applications, response time is crucial, however, even very small D (one percent of response time) can give a lot of time for error checking. If D is small relative to the variation in response time, the delay will probably not be noticed by end users.

Delayed commit will only affect throughput when the need to write extra timestamps to the disk affects throughput. If the throughput rate is high, no extra timestamps are needed. The timestamp written by the transaction at time I+D commits the transaction finishing at time I. If some bottleneck other than the disk is causing throughput to be low, writing extra timestamps to the disk won't affect throughput. However, if the workload is disk-bound because transactions go to disk for data so often, delayed commit will worsen throughput. Finally, delayed commit causes more work to be lost on failures than would be lost in the crash of a normal system. All of the delayed transactions must be redone.

3.3.1. Artificial Delay in Meta-Data Management

A technique similar to delayed commit may be used to reduce the opportunity for meta-data errors to cause disaster. In existing systems, meta-data is often used immediately after it is changed. For example, file system meta-data is changed when a file is extended. Usually a file is extended only when the DBMS has to write to the end of the file. Thus, immediately after modifying the meta-data for the file, the meta-data is used. If an error caused the file to be extended incorrectly, there is no delay before a disaster occurs. Simple tricks, like extending the DBMS log file before it needs to be written, can permit a delay period after meta-data errors in which disaster might be prevented.

4. Evaluation: Performance and Reliability

In the previous sections, we have sketched some of the performance and reliability effects of several fault tolerance techniques. In order for the techniques to be useful, system managers must be able to weigh the techniques' performance costs against their reliability benefits. Therefore, a more quantitative analysis of costs and benefits is required to complete our evaluation of these techniques.

The performance evaluation is relatively straightforward. The next section presents simulation results for transaction processing systems which use page guarding and deferred write. Ultimately, we will build and measure implementations of these techniques for the POSTGRES [28] data manager and Sprite [26] operating system.

Quantitative reliability evaluation is considerably more difficult. We would like to project the number of disasters that would occur in a target system with and without a given combination of fault tolerance techniques. An implementation alone will not give us enough information. In general, it takes years of operation to acquire sufficient data to generalize about system failures. Even then, observations would not necessarily be generalizable if only a few programmers developed the program, or if it was not widely used.

A more practical way of quantifying the fault tolerance increase provided by these techniques is to derive reliability measurements using observed errors from existing systems. The error data is broken into classes according to the fault tolerance techniques that prevent them. In the second section that follows, we take that approach and describe some of the difficulties with it.

A third method for reliability evaluation is called error seeding. In error seeding, known errors are inserted into either the program code [12] or into both code and data [8] according to some fixed distribution. By counting the number of times the seeded system fails (or has a disaster) during a test run, we can measure its reliability. Reseeding and rerunning the test program many times increases the stability of the measurements. By using the same error distribution to seed systems which use each fault tolerance technique, we can compare the effectiveness of different techniques. We discuss some analytic models based on error seeding and some of the methods' weaknesses in the last subsection.

4.1. Simulation Study for Performance Analysis

To estimate the performance impact of page guarding and deferred write, we have simulated a shared memory multiprocessor TPS. The simulator uses the TPS model described in the first section. Transactions read and write pages, produce a transaction log, and commit. The program simulates lock activity, memory management, and queueing behavior at disk and cpu. It uses an artificial workload generated using the following parameters:

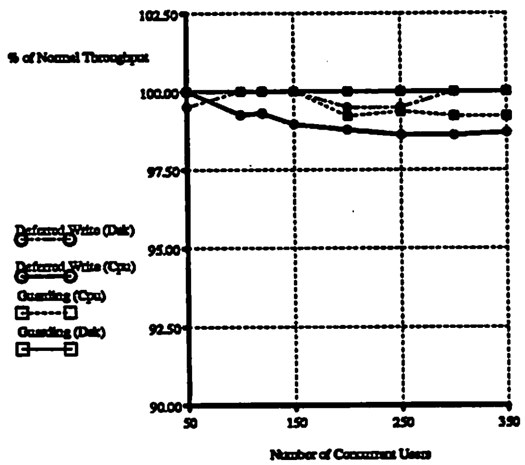
Processors	10, 6-10 MIPS each
Disks	30-120, ~28 ms per I/O
Database	500MB, 50-350 users
Memory	50MB, 4KB pages, 128 byte records
Transactions	mean size 8 records, 20% writes
Hot Spots	80% of references to 20% of database
System Call	500 instructions
TLB Shootdown	4000 instructions
Copy Cost	4 instr/byte, 512 instr/record
DBMS processing	20,000 instr/page

The parameters were chosen to emphasize the differences between page guarding, deferred write, and a traditional system under varying degrees of resource contention. For each fault tolerance technique, the parameters were adjusted to create disk-bound and cpu-bound runs. The cases were constructed by modifying the cpu speed and number of disks until the desired resource became the bottleneck.

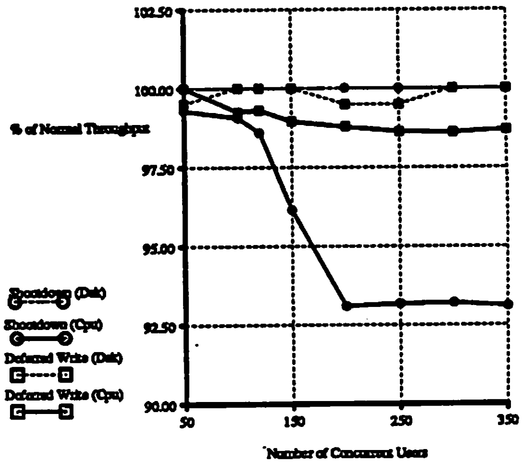
For our evaluation of page guarding, the TPS unguards a page when a transaction locks it for writing. Similarly, deferred write copies the record when a write lock is acquired. In both cases, the transaction makes a single system call at commit time to reguard pages and/or install the writes. Each guard, unguard, or restore operation paid the cost for a system call and deferred write paid the cpu cost for two copies. In our initial simulations of page guarding, we did not consider the cost of MMU operations in a shared memory multiprocessor. We consider guarding both in the case when MMU consistency is slow (shared memory multiprocessor, TLB Shootdown algorithm) and fast (uniprocessor, single system call).

4.1.1. Simulation Results: Throughput and Response Time

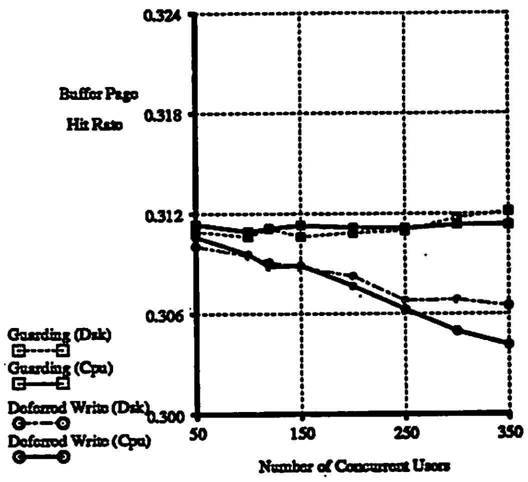
Graph 1 compares the throughput of the deferred write and the guarding TPS to that of an unmodified system. The results are normalized so that the throughput for each modified TPS is expressed as a percent of the normal system's throughput. The curves have been smoothed slightly. Any time the difference between normal and modified system throughput was less than the standard deviation of the



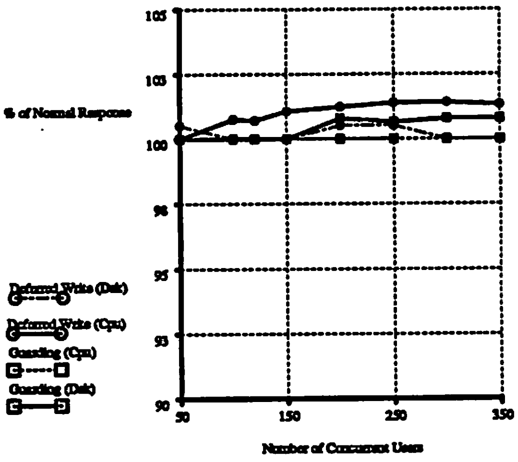
Graph 1: Normalized Throughput Rate



Graph 2: Guarding Throughput with TLB Shootdown



Graph 3: Buffer Hit Rate



Graph 4: Normalized Response Time

simulation runs, they were recorded in the graph as equal (100%).

In graph 1, throughput degradation never exceeds two percent. Not surprisingly, the techniques have no impact at all in the disk-bound case. Deferred write and guarding only increase cpu demand so do not affect throughput when the cpu is not heavily utilized. Even in the cpu-bound case, the throughput effects of the algorithms are not significant. The normal processing cost in the DBMS dominates the cost of the extra copies and system calls. In general, guarding fares slightly better than deferred write when the MMU update costs are not taken into consideration. The TLB shootdown algorithm (graph 2) dramatically increases the cost of guarding but reduces throughput (over the normal system) only by seven percent.

Finally, we had expected that deferred write might worsen the page fault rate, hence worsen performance. Although the page fault is higher in deferred write, the difference did not have much effect on performance. Graph 3 shows how buffer hit rate changes as the degree of multiprogramming. The amount of scratch space required by deferred write increases linearly with multiprogramming level.

Graph 4 shows the response time differences in the algorithms, normalized in the same way as throughput. The response time difference between the three systems almost negligible. Even in the cpu-bound case, the response time is dominated by disk latency. For these parameters, a one percent delay added by delayed commit provides 100,000 to 400,000 instructions per processor during which an error might be detected (4 MIP processor, cpu bound).

4.2. Reliability Analysis Through Error Studies

There are two sources of existing system failure data: (unpublished) bug report databases and published error studies. In general, neither source provides enough information to evaluate our techniques. A bug report might describe the immediate cause of a failure (segmentation fault), the original cause (race condition in network driver), or describe symptoms (garbage characters appeared on the screen). Often, specifics about system behavior after the error are missing (e.g "then the system eventually crashed"). It is this post-error behavior which both causes disaster and determines which fault tolerance techniques are successful. Published studies summarize and categorize error data, hence provide less detailed information than bug reports. The researchers conducting a study group errors into classes that may not have much relation to our techniques. In spite of these concerns, some useful information can be garnered from published and unpublished error studies.

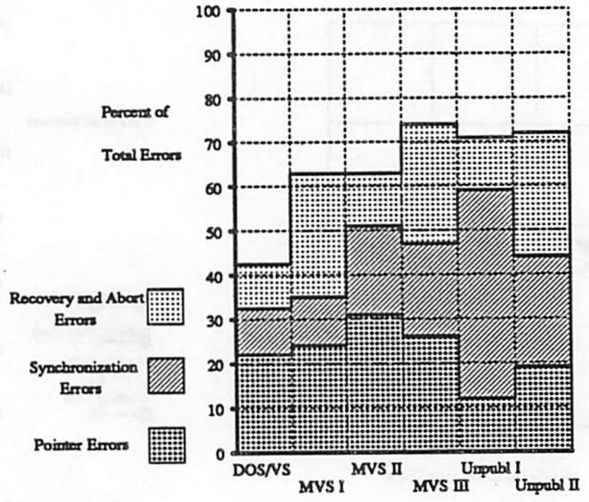


Table 3: Summary of Error Study Data

Several published studies examine the frequency of error clusters. The existence of small error clusters increases the effectiveness of delayed commit. Unfortunately, the interval between errors in the cluster must be extremely short (a small fraction of average transaction response time) for delayed commit to be useful. In two MVS studies ([32][24]), error clusters were observed in 60% and 20% of all failures, respectively. In the first study, 35% of these clusters were less than a minute apart on an IBM 3081 (finest grain reported was one minute). Two sources for clustered errors were (a) error handlers failed (making the original and the error spawned by the bad handler into an error cluster) (b) several transient hardware errors occurred together. As far as delayed commit is concerned, the results are inconclusive.

Other observations from the studies confirm that software errors correlate with high system load, hardware errors, and highly interactive workload [18][32]. Also, Gray reports 99.25% of all software errors are handled successfully by retry [13]. In an MVS retry study [24], retry was much less effective (0-70%) but these retries were operating system recovery routines instead of full system rollbacks.

Table 3 matches the error classifications from six operating system error studies to our error classes. In some cases, the match is crude. For example, all bad address trap errors are assumed to fit into our pointer error class. Some of our error classes (value and operand) could not be discerned from the published information. The published studies in the table are from MVS ([32] [24], [24], and DOS/VS [10].

While not enough to compare the mechanisms fully, the studies indicate that some of the error classes handled best by our mechanisms are, in practice, important error classes (pointer, synchronization, and abort). Of course, the errors observed the most often in these studies are not necessarily the ones that have long error latencies. The failure information does not indicate how often disasters occurred as a result of these errors or how long the error latency was. So, this information, in and of itself, does not indicate that our mechanisms are necessary.

4.3. Error Seeding Analysis

In this section, we make a partial evaluation of page guarding using a family of analytic models based on the error seeding idea. Guarding will detect pointer errors only if the page struck by the error happens to be guarded. If we use error seeding to simulate pointer errors, a location is chosen from the database buffer pool according to some error distribution. The chosen location is either in a guarded page (and is detected immediately) or in an unguarded page (and may be detected if examined by a transaction error checker). Only permanent (buffer pool) pages are considered because pointer errors which hit transient data pages can only contribute to disaster indirectly (through error propagation). Error propagation is assumed to be represented in the seeded error distributions.

By considering the likelihood that the error distribution will select guarded pages, we can get much of the same information that would have come from the error seeding study. We will not, however, know whether the seeded error would have caused a disaster. The models will also give us no new information about the relative likelihood of pointer errors and decision errors. Also, only single errors are considered. Multiple (independent) errors would increase guarding's effectiveness since recovering from any of the errors would cause all to be repaired.

Different error distributions will make different pages candidates for error insertion. Some reasonable error distributions are:

(1) **Uniform:** every page is equally likely to be hit by an error. This is the best case for guarding since a large fraction of the buffer pool is guarded at any time.

(2) **Locality-biased:** the error will only affect one of the last K pages the system has used. The assumption here is that pointer errors damage the buffer pool when a recently used pointer is reused in an incorrect way. For example, an uninitialized pointer variable on the stack is probably most likely to contain a small integer, and to crash immediately when the pointer is used. Disasters are more likely if the uninitialized pointer reuses another pointer value. These pointer values are most likely to point to recently-used data.

(3) **Write-locality-biased:** As above, but read-only data is never chosen. Pointer errors only occur when the erring program writes to a bad pointer location. This model assumes that data that is normally never written is less at risk of accidentally overwrite.

(4) **IO-biased:** errors affect pages only as they are being read in or written out (or soon after). Disk IO has synchronization and buffer management (page replacement) associated with it and may be a source of errors. Sometimes transient hardware failures cause software errors, as well.

(5) **Point-of-control:** only the page currently being written is ever affected. This is a lower bound since, here, guarding gives no increase in reliability.

Each error distribution implies a probability P_g that errors strike guarded pages. There is also a probability P_c that, if the error strikes an unguarded page, the transaction error checker eventually looks at the page and detects the error. The models described below use the following symbols:

M is the degree of multiprogramming
P is the average number of pages used by a transaction
W is the fraction of pages written
C is the fraction of pages in disk cache, but not in use

4.3.1.1. Uniform Distribution

In this distribution, all pages are equally likely to be hit by an error. Pages in the cache and read-only pages are guarded, so

$$P_g = C + (1 - W)*(1 - C).$$

In our simulations, most of the pages in memory are not in use because contention for locks, disks, and cpu always outweighs contention for memory. Thus, P_g ranges from 97 to 99 percent for simple page guarding. Deferred write unguards W *records* instead of W *pages*, so its P_g will be slightly larger.

4.3.1.2. Locality-Biased and Write-Locality-Biased Error Distributions

In this distribution, the system has a K page working set of recently used pages. Only pages from the working set can be damaged by wrong page errors. A page from the working set is only unguarded if written. Even the pages that were written are now guarded again if the transaction that wrote them has committed. To determine P_g, we must determine what fraction of the last K pages used belong to uncommitted transactions. Suppose for simplicity that a transaction touches its P pages in order one after another with no rereferences.

In the steady state, a transaction completes after every 1/P page references. The chance that a page touched i references ago belongs to a committed transaction is then min(($\lceil i/M \rceil$),P)/P, and, the chance that the page belongs to a transaction that is still alive is L_i = 1 - min($\lceil i/M \rceil$,P)/P. The average L_i from 1 to K is the average number of the last K references that went to transactions that are still uncommitted at the time of the error.

Using these assumptions, we can compute P_g. In simple page guarding, W percent of the references to uncommitted pages were writes. Thus, $P_g = 1 - (W^*(\sum_{i=1}^{K} L_i))$. In the write-locality distribution, an error can only strike pages that have been written, so the page is only guarded if the transactions is finished. Hence, $P_g = 1 - (\sum_{i=1}^{N} L_i)$ For (P=10,M=100,W=.20) and K=50,300,1000, P_g = 0.82, 0.84, and 0.89, respectively, using the locality-biased distribution. In the write-locality-biased distribution, P_g is 0.10, 0.30, and 0.45 for the same parameters.

In deferred write, scratch pages belonging to the transaction do not become guarded when the transaction commits. These pages are reused (or at least are never guarded). Hence, for the locality-biased distribution, P_g = 1 - W. For the write-locality-biased distribution, P_g = 0.

4.3.1.3. IO-biased Distribution

The link between device failures and software errors suggests that pages may be more subject to error when they are read from or written to the disk. Of course, nothing suggests that device failures cause *pointer* errors or that those pointer errors would primarily affect the data being transferred.

For this distribution, P_g is the ratio of disk reads to total disk operations. Data being written is guarded while data being read cannot be. Suppose the cache hit rate is H. Then P*(1-H) pages must be read during every transaction. Because W percent of the cache is dirty, the transaction flushes W*P*(1-H) pages to disk. Assuming that transactions are small enough that only one page is written out on commit, P_g = (P*W*(1-H) + 1)/(P*(1-H)*(1+W)+1). Using the parameters (and hit rate) from our simulations, P_g = 0.26.

4.3.1.4. Summary of Error Seeding Analysis Results

The analysis shows that the reliability increase provided by guarding varies significantly with the error distribution. In all but the most pessimistic assumptions, however, the P_g is substantial (10% to 99%). Even if P_g is small, these are errors that transaction error checkers would normally not detect. Suppose P_c is 50% (the checker is 50% effective against the pointer errors that it sees). Then, P_g of, say, 10% in the write-locality-biased case implies an overall increase in pointer error detection from (90*50) 45% to 55%.

4.4. Performance and Reliability Evaluation: Conclusions

The simulation results make the performance of deferred write look promising. Since it provides more protection than simple page guarding at nearly equal cost, deferred write will be preferable to guarding in most cases. Only when records are very large or when whole pages are being initialized by a transaction (hence nothing in the page needs to be protected during the transaction) might page guarding alone be worthwhile.

Our reliability results are less conclusive. The error study shows that our techniques are attacking the right problems. Concurrency, error recovery, and pointer management cause most failures. Intuitively, this makes sense. Concurrency and error handling code is often complex and difficult to test. Pointer errors may simply be an early manifestation of many different kinds of control and data errors. These errors are also difficult to correct since different parts of the system can be damaged each time the error arises. Although our techniques are designed to protect against these kinds of errors, we have not been able to quantify their fault tolerance impact.

5. Fast Recovery: Main Memory as Recovery Store

The techniques we have proposed to protect against long latency errors can also be used to improve recovery speed. The POSTGRES Storage System has a fast recovery mechanism that depends on non-volatile main memory [30]. Even a traditional database storage system could recover faster if the database log and frequently-used database pages did not need to be reread from disk during recovery. The hardware challenges in building reliable non-volatile main memory have for the most part been solved [17][29]. Unfortunately, concern about software errors still forces TPS designers to use Disk Recovery Store (DRS) instead of Main Memory Recovery Store (MMRS). With a few modifications, the techniques discussed so far can tip the scales toward using MMRS.

5.1. Issues: What makes MMRS different from DRS?

Our model of MMRS is as follows: A region of non-volatile main memory is guarded for use as recovery store. The DBMS maps this region into its shared memory. When a transaction commits, its log records and any modified operands are copied to this region. The MMRS is still managed as a cache so log records and data pages are aged out to disk eventually. On a failure, all main memory data except that in MMRS is thrown away. Figure 4 shows how DBMS shared memory might take advantage of MMRS.

From the standpoint of software error protection, MMRS poses several problems not found in DRS:

(1) **Error Propagation Speed:** Broken software can damage memory faster than it can damage disk. If a transaction decides to overwrite a location in main memory, it can do so in one instruction. The advent of page guarding can only slow a determined overwrite by a few more instructions. Destroying data on disk takes several thousand instructions (from beginning to end) After the instructions are complete, there is a delay of about 18 ms (180,000 instructions on a 10 MIP machine) while the disk seeks to the correct location. The extra work and extra latency in the DRS case give the system a chance to detect that it is in error and fail.

(2) **Data Error Buffering:** Recovery causes the DRS system to throw away its cache. During recovery, pages in the cache damaged by latent data errors will be thrown away as well. In DRS systems, data errors are buffered in memory for seconds before being flushed to disk. As memories become larger, this extra latency becomes even larger. MMRS moves data errors into recovery store immediately upon commit, hence, the data error becomes a disaster immediately.

(3) **Meta-Data Risks:** MMRS faces a larger threat from meta-data errors for three reasons. First, disk meta-data can be checked for errors on every write to the disk. Main memory meta-data is used by

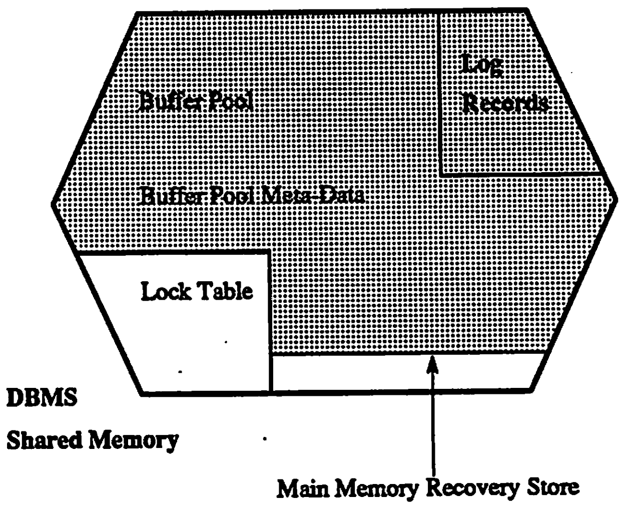


Figure 4: Allocation of MMRS

hardware and is on the critical path for every instruction execution. The meta-data structures are relatively inflexible (making error checker design hard) and error checking on use is severly restricted for performance reasons. Second, in DBMS systems, the disk meta-data for critical files does not usually change very often. Memory allocation changes regularly, as pages are brought in and out of memory. If the chance of errors is proportional to the frequency with which something is updated (quite possible for propagated errors), main memory meta-data is more susceptible to error. Third, MMRS is managed as a cache, so it requires some recoverable data structures unnecessary for DRS. The dirty bits and buffer-to-disk-block mappings for the cache must survive failures. Latent errors in these data structures are thrown away with the data structures during DRS recovery.

(4) **Independent Failure Modes:** If disk management and memory management software fail independently, DRS has an advantage. Failure of disk management software can cause a disaster for either system. Memory management failure (directly) causes a disaster only in MMRS. Conversely, though, there is a correlation between load on a device and failure rate in both the hardware and software related to the device. MMRS may produce a reliability advantage by lessening the disk load.

In the section that follows, we reexamine the effects of each error type on MMRS and make some suggestions to handle the increased danger of meta-data errors.

5.2. Error Analysis: What Causes MMRS Disasters?

In order for MMRS to be acceptable, errors must be no more likely to damage MMRS than DRS. Earlier, we discussed three ways that errors could cause disaster: damaging the log, damaging data, and damaging recovery store meta-data. In this section, we are going to compare DRS and MMRS according to the danger posed to each system by each of these kinds of disasters. Log errors were defined earlier as errors whose effects are propagated to the transaction log. These become disaster on transaction commit. For this analysis, we break data errors into three types: direct data errors, pointer errors, and indirect data errors. Direct data errors are those that can affect MMRS directly -- meta-data and decision errors. Pointer errors are a special case of direct errors, since, if a few precautions are taken, they can be neutralized with guarding. Indirect data errors are those that affect data while it is writable and are eventually aged into recovery store.

5.2.1. Log Errors

Once a damaged log record is stored in recovery store, the kind of recovery store used (MMRS or DRS) will not matter. Log errors are more of a concern to MMRS than DRS only because the latency of a disk write operation allows the system a chance to fail before the log write is complete. Log errors are equivalent in MMRS and DRS if delayed commit is used to simulate the latency between the decision to write to the log and the completion of the write.

5.2.2. Pointer Errors

In order for MMRS to be effectively protected against pointer errors, all committed data in MMRS must be guarded. The simple page guarding technique will not be sufficient since permanent records are left unguarded during a transaction. Deferred write is better, but during *restore*, it, too, leaves committed data records unguarded. If a record is being restored into page P, a pointer error during the restoration could damage the other records on P. The risk to the unprotected pages can be reduced by (1) unprotecting one page at a time so interrupts can be suspended during each copy, (2) only allowing the record to be unprotected for the single process doing the copy, and (3) carefully coding and testing the OS *restore* routine.

In environments in which a carefully-coded *restore* is impossible, the OS can maintain a guarded backup copy of the page being updated. Before unguarding the original page, the system copies the page into a writable backup page. After the copy, the backup is unmapped (from the OS virtual address space), and the original is made writable. When *restore* is done, the original page is reguarded. No matter when a pointer error occurs during *restore*, either the backup or the original is unwritable. The system must add to its MMRS meta-data, however, the mapping between backup page and data page undergoing *restore*. Below, we discuss the performance implications of copying a full hardware page for each record modified by the transaction.

5.2.3. Indirect Data Errors

As we mentioned above, indirect data errors have a longer propagation latency in DRS than they do in MMRS. If the system recovers between the time a data page is damaged and the time it is flushed to recovery store, the damaged page is thrown away without causing a disaster. By saving backup copies from *restore*, MMRS can manufacture its own indirect error buffer. Backups for the last K buffers updated could be saved (unmapped) in MMRS. On recovery, the backup and the transaction log would be used to reconstruct the current state of the database. As in DRS, a checkpointing mechanism is required to bound the number of log records touched during recovery. Also a second update to the same buffer shouldn't require a new backup.

Depending on how much space is allocated for the buffer, MMRS can approach the fault tolerance of DRS for indirect errors. The cost, however, is that less memory is available for disk cache.

5.2.4. Direct Data Errors

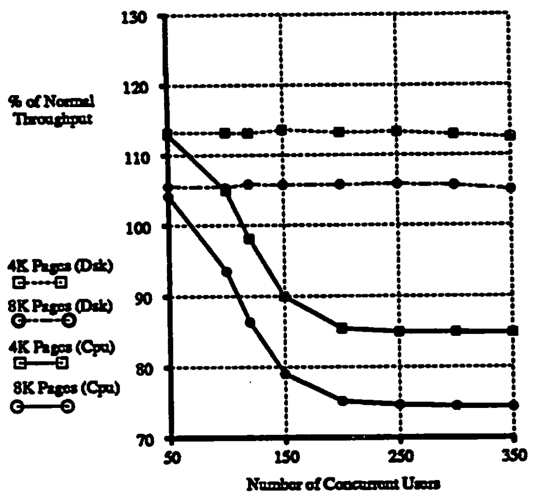
Given guarding and the new restore algorithm, the only remaining software fault tolerance concern is that errors somehow subvert the guarding mechanism and overwrite MMRS pages. The two types of errors that might behave that way are decision errors and meta-data errors. We have already discussed a capability-based mechanism for reducing the effects of decision errors. It would have to be extended to limit OS access to the DBMS buffer pool.

For meta-data errors, we described three important differences between DRS and MMRS: meta-data cannot be checked on use, meta-data is updated more often, and MMRS requires cache management meta-data. The solution for all but the last of these is to treat MMRS meta-data differently from other main memory meta-data. Distinguishing MMRS meta-data can allow the system's MMU to check for bad meta-data at run time efficiently.

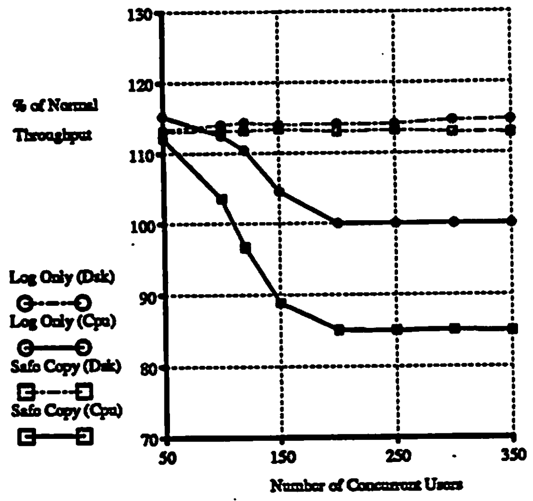
In current systems, the protection bits in a PTE are assumed to be the only information available about whether or not a page is writable. In fact, those bits could be wrong either because they have been recorded incorrectly or because a writable PTE is accidentally mapped to data that is supposed to be protected. If, however, a database buffer pool uses a fixed size region of physical memory and that region is always guarded, additional information is available. In this case, pages at the physical addresses associated with the buffer pool should never be writable except during *restore*. Furthermore, since the buffer pool is never paged, the addresses can be mapped into the DBMS process address space in sequential order. Now, the system has additional information that can be used to determine whether or not a page table entry has been damaged.

An MMU could be modified to take advantage of this information without slowing the critical path of instruction execution. The modified MMU would have two registers for the base and bound of the protected region of physical memory. When the MMU is loaded with a writable page mapping, it can quickly check whether the physical address maps to the buffer pool. Because the checks happen when the MMU is loaded, they are not in the critical path for most instructions. In order to allow the pages to be written during restore, the MMU would have to have a restore register containing the physical address of the *one* page which is currently being copied. If the MMU detects an invalid PTE load, it checks the restore register to see if the PTE being loaded refers to a record being restored. If the MMU is software-loaded (as in the DECstation and Sun architectures), the mechanism just described can be built into software rather than the MMU hardware.

This kind of MMU management is advantageous for several reasons. First, PTEs can be checked on reference. If a buffer pool page is ever accidentally mapped into writable memory, it is still protected. In the background, the system can also check that the buffer pool addresses are always mapped in the correct order. Second, the PTEs for the buffer pool are never modified. That removes the possibility that the PTE will be damaged during an update. The downside is that the shared memory region must be fixed size and contiguous in order for this mechanism to be efficient and safe. This limits the operating system's freedom to allocate main memory.



Graph 5: Normalized Throughput



Graph 6: Safe vs. Normal Restore Algorithm

Performance of Main Memory Recovery Store

Making cache management meta-data (dirty bits and mappings) recoverable requires updates to this data to be logged. Delayed commit can help reduce MMRS vulnerability to errors in this meta-data. Buffers loaded during the last D seconds could not contain data for committed transactions. Targeting the buffers for replacement more than D seconds ahead of time reduces the effectiveness of LRU replacement, but will guarantee that the D second old mapping meta-data is enough information for recovery. That allows buffers loaded during the last D seconds to be marked unoccupied and allows the new mappings to be thrown away. The buffer used to delay indirect errors also delays the time until dirty bits must be recoverable. Dirty bits do not need to be recoverable until the backups for the buffers they represent are thrown away.

5.3. MMRS Performance Impact

MMRS improves the recovery speed by reducing the amount of data that must be read from disk during recovery. The tail of the database log and much of the most often used data is in protected memory, hence is available after a crash. In order to determine the performance impact of MMRS on normal system operation, we modified our TPS simulator.

Performance of an MMRS system is different from that of a traditional system in two ways. MMRS systems do not write log data to disk on transaction commit, but, because of the safer *restore* operation, they must copy a full page for every data record the transaction modifies. Graphs 5 and 6 show the effects of these two differences on performance. In the graphs, we show MMRS systems with 8K and 4K pages.

In all cases, the advantage of not having to go to disk on commit makes MMRS faster, during normal operation, than a traditional system. In the cpu-bound case, the copying costs of the *restore* algorithm start to dominate as the cpu becomes saturated. The original *restore* algorithm could be used instead of the safe one in order to lessen the cpu cost of a transaction. Here, MMRS beats DRS unless the cpu is so saturated that the difference in disk load does not matter.

6. Related Work

6.1. Writing Fault Tolerant Software

Strategies for improving software fault tolerance fall into three classes: fault prevention, error correction, and retry. Errors can be prevented through modular design, exhaustive testing, and formal software verification (techniques surveyed in [23]). Although all software designs incorporate one or more of these techniques, the complexity and size of concurrent programs like the OS and DBMS make error prevention alone an insufficient approach.

Correction, or **forward recovery**, techniques use redundancy to detect and correct damage caused by errors. One common forward recovery technique is N-version programming [2], in which several versions of a program are designed independently by different programmers. The N versions run simultaneously, comparing results and voting to resolve conflicts. In theory, the independent programs will fail independently. In practice, multiple version failures are caused by errors in common tools, errors in program specification, errors in the voting mechanism, and commonalities introduced during bug fixes. Furthermore, experimental work [19] has indicated that while N-version programming can increase reliability, often independently constructed programs fail on the same input. Not surprisingly, different programmers often find the same hard tasks difficult to code correctly.

Retry, or **backward recovery**, eliminates an error's effects and then reattempts the computation. Recovery techniques which use data stored on disk for backward recovery are surveyed in [15]. Normally in commercial systems, the retry uses the same software that just failed. There is evidence that, in concurrent system software (OS and DBMS), timing differences cause the software to execute differently during retry, hence avoid repeating the error [13]. Some research systems attempt to avoid repeated failures by using different code to execute the retry [27]. This technique only works if both versions of the code are equally well tested and debugged.

6.2. Fast Recovery

Several systems have been designed so that recovery time is not bounded by magnetic disk access speeds. One suggestion is to have two processors: one general purpose cpu for transaction processing and one recovery cpu with access to non-volatile storage [21]. The main cpu does not have access to MMRS; at commit time, it hands log records to the recovery cpu. The recovery cpu copies log records into non-volatile storage, collects them into batches, and writes the batches to disk. Although the approach requires specialized hardware, the second cpu runs less complex software so it may be less vulnerable to software errors. The two cpu approach increases danger of synchronization errors during commit, and does not protect the buffer pool.

The usual technique is to contain software errors so that only part of the system needs to be recovered. Auragen and Tandem systems run on non-shared memory multiprocessors replicating critical software components on more than one processor [5][3]. Their recovery mechanism assumes that OS and DBMS errors will be contained on the machine that causes them. Each critical OS and DBMS process has a backup process that can take over if the primary process fails. Recovery is faster than in normal backward recovery because the takeover mechanism is faster than rollback and restart.

Unfortunately, process-pair techniques involve a lot of communication overhead. The primary must continually pass state information to the backup so that it will be able to take over on a failure. By trusting more of the low-level system software, we can trade off reliability against increased performance. For example, the primary and backup could run on different virtual machines on the same physical machine [6]. Object-oriented languages [31] or modular systems with typesafe languages [20] could use the compiler to prevent errors from affecting both primary and backup. Guarding and deferred write put more confidence in the OS virtual memory management software than is required by process pair technique, but less confidence in the message passing system. We expect these techniques to outperform process pairs since communication goes through shared memory instead of an interprocessor bus.

6.3. Measuring Software Fault Tolerance

Analytic models for judging improvements in software reliability are sometimes used in commercial systems ([12] surveys some of these techniques). Existing models, however, are oriented towards finding trends in reliability improvement over a single system's lifetime. The most common models [25][22] assume that effort spent debugging a system will show diminishing returns over time. Given current and past rate of bug fixes, these models project the increase in reliability expected from continued debugging.

Normally, the models cannot be used to weigh two alternative strategies for detecting errors.

Researchers evaluating fault tolerance techniques have used error seeding and randomized input models. In [1] and [19], thousands of random test cases were generated for test applications built with different fault tolerance mechanisms (recovery blocks and N-version programs). The mechanisms were compared on the basis of the number of the random test cases that they failed on. These cases all involved sensor data for flight management applications. For a TPS, random input would not be as interesting.

Error seeding is generally used to measure software reliability in the face of hardware faults (e.g. [9][14]). Chillarege and Bowen [8] simulated pointer errors with error seeding to evaluate the recovery capabilities of an IMS data manager and an MVS operating system. Errors were simulated by choosing a random page of physical memory and overwriting its contents with garbage. Error latencies have been observed using a fault injection model in [7]. All of these models used what we have called the uniform error distribution.

7. Conclusions

As hardware becomes more reliable and non-volatile memory becomes more common, fail-stop errors will become less and less the norm. We have suggested several techniques for reducing data unavailability due to long latency software errors.

(1) Page guarding is a passive protection technique in which unused and read-only disk cache blocks are write-protected. Page guarding allows hardware to prevent damage due to corrupted pointers.

(2) Unguard keys increase the effectiveness of guarding by disallowing one module from unprotecting another module's data.

(3) Deferred write helps keep damaged DBMS processes from propagating errors to healthy ones. Because most shared data is protected until transaction commit, propagation only occurs if the error remains undetected during the entire transaction.

(4) Synchronized write allows deferred write's to detect violations of the buffer locking protocol.

(5) Delayed commit gives the system extra time to detect errors before the affected data becomes irrecoverable.

All of these methods can be used in conjunction with the application-dependent error checking and safe coding practices commonly used in high-availability systems.

We have evaluated the performance and reliability impact of these techniques using both analytic models and simulations. The overhead from our fault tolerance techniques is only noticeable in cpu-bound systems. Even then, the effect on throughput is not significant unless the DBMS itself does very little processing. Although little is known about error distributions, our reliability analysis shows that page guarding is effective for all but the most pessimistic assumptions. A survey of existing system error data indicates the most common errors to occur in practice are the ones our techniques handle best. These common errors involve (a) concurrency and synchronization, (b) recovery and error handling, and (c) bad pointers.

We also looked at the ways in which the fault tolerance techniques could be used to support a main memory recovery store. MMRS improves recovery speed by obviating the need to read the log and reload the buffer pool from disk after a failure. The software fault tolerance differences between main memory and disk recovery store come largely from (a) faster propagation of errors in memory, and (b) more complex meta-data for memory management. Our proposed techniques offer many ways to reduce or delay software error propagation. Modifications to the software or hardware that loads the MMU allows MMRS meta-data to be safer than that used for the rest of memory.

8. Acknowledgements

Members of the POSTGRES, Sprite, and RAID groups have provide comments and suggestions during the course of this work. Ramon Caceres, Peter Danzig, Anant Jhingran, Margo Seltzer, and Shin-Yuan Tzou have been especially helpful.

9. References

1. T. Anderson, P. Barrett, D. Halliwell and M. Moulding, "Software Fault Tolerance: An Evaluation", *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985).
2. A. Avizienis, "The N-version approach to Fault Tolerant Software", *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985).
3. J. Bartlett, "A NonStop Kernel", *Proc Eighth Symposium on Operating Systems Principles*, 1981.
4. D. Black, R. Rashid, D. Golub, R. Hill and R. Baron, "Translation Lookaside Buffer Consistency: a Software Approach", *Proc. Third Architectural Support for Programming Languages and Operating systems*, April 1989.
5. A. Borg, W. Blau, W. Graetsch, F. Herrman and W. Oberle, "Fault Tolerance Under UNIX", *ACM Transactions on Computer Systems 7*, 1 (February 1989).
6. J. Buzen and U. Gagliardi, "The Evolution of Virtual Machine Architectures", *AFIPS Conference Proceedings*, June 1973.
7. R. Chillarege and R. Iyer, "Measurement Based Analysis of Error Latency", *IEEE Transactions on Computers C-36*, 5 (May 1987), 529-537.
8. R. Chillarege and N. Bowen, "Understanding Large System Failures -- A Fault Injection Experiment", *International Symposium on Fault Tolerant Computing Systems*, 1989.
9. A. Damm, "The Effectiveness of Software Error Detection Mechanisms in Real-Time Operating Systems", *International Symposium on Fault Tolerant Computing Systems*, 1986.
10. A. Endres, "Software Errors in Systems Programs", *IEEE Transactions on Software Engineering SE-1* (June 1975).
11. R. Fabry, "Capability-based Addressing", *Communications of the ACM 17*, 7 (July 1974), 403-412.
12. A. Goel, "Software Reliability Models: Assumptions, Limitations, Applicability", *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985).
13. J. Gray, "Why Do Computers Stop and What Can Be Done About It?", *Proc. Sixth Symposium on Reliability in Distributed Software and Database Systems*, 1987.
14. U. Gunneflo, J. Karlsson and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation", *International Symposium on Fault Tolerant Computing Systems*, 1989.
15. T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *Computing Surveys 15*, 4 (December 1983).
16. M. Hill, S. Eggers, J. Larus, G. Taylor and al., "Design Decisions in SPUR", *IEEE Computer 19*, 11 (November 1986).
17. R. Horst, "Reliable Design of High Speed Cache and Control Store Memories", *International Symposium on Fault Tolerant Computing Systems*, 1989.
18. R. Iyer and D. Rossetti, "Effect of System Workload on Operating System Reliability: A study on IBM 3081", *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985).
19. J. Knight, N. Leveson and L. StJohn, "A Large-Scale Experiment in N-version Programming", *International Symposium on Fault Tolerant Computing Systems*, 1985.
20. B. W. Lampson and D. D. Redell, "Experiences with Processes and Monitors in Mesa", *Comm. of the ACM 23*, 2 (February 1980), 105-117.
21. T. Lehman and M. Carey, "A Recovery Algorithm for a High-Performance Memory-Resident Database System", *Proc. Thirteenth Very Large Data Bases*, December 1987.
22. B. Littlewood, "How to Measure Reliability and How Not To", *IEEE Transactions on Reliability R-28*, 2 (June 1979).
23. D. Morgan and D. Taylor, "A Survey of Methods for Achieving Reliable Software", *IEEE Computer 10hhhjj*, 2 (February 1977).
24. S. Mourad and D. Andrews, "On the Reliability of the IBM MVS/XA Operating System", *IEEE Transactions on Software Engineering SE-13*, 10 (October 1987).

25. J. Musa, "Theory of Software Reliability", *IEEE Transactions on Software Engineering SE-1*, 3 (September 1975).

26. J. Ousterhout and al., "The Sprite Network Operating System", *IEEE Computer 21*, 2 (February 1988).

27. B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975), 220-232.

28. L. Rowe and M. Stonebraker, "The POSTGRES Data Model", *Proc. Thirteenth Intl. Conference on Very Large Data Bases*, December 1987.

29. D. Sarrazin and M. Malek, "Fault-Tolerant Semiconductor Memories", *IEEE Computer 17*, 8 (August 1984).

30. M. Stonebraker, "The POSTGRES Storage System", *Proc. 1987 VLDB Conf. Conference*, September 1987.

31. B. Stroustrup, "The C++ Programming Language", *Addison-Wesley*, 1986.

32. P. Velardi and R. Iyer, "A Study of Software Failures and Recovery in the MVS Operating System", *IEEE Transactions on Computers C-33*, 6 (June 1984).