Using Write Protected Data Structures To Improve Software Fault Tolerance in Highly Available Database Management Systems

Mark Sullivan, Michael Stonebraker

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

Abstract

This paper describes a database management system (DBMS) modified to use hardware write protection to guard critical DBMS data structures against software errors. **Guarding** (write-protecting) DBMS data improves software reliability by providing quick detection of corrupted pointers and array bounds overruns. Guarding will be especially helpful in an extensible DBMS since it limits the power of extension code to corrupt unrelated parts of the system. Read-write data structures can be guarded as long as correct software is able to temporarily unprotect the data structures during updates. The paper discusses the effects of three different update models on performance, software complexity, and error protection. Measurements of a DBMS which uses guarding to protect its buffer pool show two to eleven percent performance degradation in a debit/credit benchmark.

1. Introduction

Today, software errors are the largest cause of failure in fault tolerant transaction processing systems [Gray90]. Between 1985 and 1990, software was at fault in 62% of Tandem system outages. The second and third largest contributors, operations and hardware, were at fault 15% and 7% of the time, respectively. In order to improve the reliability of these systems, we must improve the reliability of the software they run.

One factor that limits the reliability of software is *software error propagation*. Using redundancy, hardware components can detect their own errors and recover without disturbing the system. Software errors, on the other hand, often cause damage that is not detected immediately. The damaged system can initiate a sequence of additional software errors as it executes, eventually causing the system to corrupt permanent data or fail. Error propagation complicates software failure modes, making the code difficult to reason about, test, and debug. Reproducing propagation-related failures during debugging is difficult since error propagation is often timing dependent.

For some systems, software error propagation reduces DBMS availability as well as reliability. In multi-process DBMS architectures, a software error in one process might propagate damage to shared data structures. An uninitialized pointer used in one process, for example, could propagate damage to DBMS shared data before the error is detected. Even if no propagation has occurred, all DBMS processes may have to go through recovery simply because the extent of propagated damage is not known.

Unfortunately, the advent of extensible data managers will make the error propagation worse in the future than it is now. Extensible DBMS include extended relational systems [Stonebraker87], object-oriented systems [Bannerjee87], and DBMS toolkits [Carey86]. An extensible DBMS lets users or database administrators add access methods, operators, and data types to manage complex objects. Moving functionality from DBMS clients to the DBMS itself improves application performance but could worsen system failure behavior. Extensibility allows different object management packages with varying degrees of trustworthiness to run together in the data manager. Every time one user on the system tries to use a new object manager or combine existing ones in a new way, there is a risk of uncovering new errors. Because of error propagation, this risk is not confined to the person using the new feature, but affects the reliability and availability achieved by everyone.

The most common approach to software fault tolerance is to write additional code that checks for errors. By detecting errors quickly, systems limit the chance that minor errors will propagate into worse ones. However, checking for errors increases processing

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

costs. No published figures are available regarding the cost of error checking in the DBMS, however, run time checks for array bounds overruns in Fortran programs can double program execution time [Gupta90]. Furthermore, the checkers themselves can have defects. They have to be maintained as the software they check is maintained. Implementing and testing them increases development cost.

To address software error propagation, we have modified a DBMS to use hardware write protection to protect some of its data structures from propagated errors. Several system calls were added to the Sprite operating system [Ousterhout88] to allow the DBMS to **guard** (write protect) regions of its address space. The DBMS uses these services to protect data in its buffer pool. To provide read-write data with protection against errors, the DBMS must support an *update model* that allows correct software to modify protected data, but prevents accidental updates by incorrect software. Different update models will make different tradeoffs regarding software complexity, performance, and the kind of error protection offered.

We have experimented with three models for updating guarded data structures: *expose page*, *deferred write*, and *expose segment*. A single DBMS can use different update models in different program modules, if necessary. The *expose page* model is the simplest one. The DBMS must recognize that it is about to update a protected record, unprotect the page containing the record, and reprotect the record after it is updated. In the *deferred write* model, the DBMS copies a record it intends to update into unprotected memory and updates the copy. At the end of transaction, a system call recopies the updated record into protected memory. Finally, the *expose segment* model lets the DBMS make a system call to unprotect all guarded data at once. After the update, a second system call reprotects the guarded data.

In all three models, guarding DBMS data allows the hardware to detect illegal attempts to write to protected pages. As a debugging tool, guarding can help eliminate pointer management errors earlier in the software development cycle. Even after product release, guarding lessens the impact of addressing-related errors by detecting errors at the time propagation occurs rather than after the damaged data is used. Because guarding detects a class of errors not well-covered by data consistency checkers, it complements existing fault tolerance techniques. For multi-process DBMS architectures, guarding can prevent one DBMS process' errors from corrupting data structures used by the other processes -- improving overall DBMS availability. In an extensible data manager, guarding is a compromise between running application code in a separate process and running it as a full fledged part of the DBMS. Much of the protection of the separate address space model is retained at a cost much closer to the single-address space model.

Initial performance measurements indicate that guarding the DBMS buffer pool has a relatively small performance impact. For a debit/credit benchmark, guarding caused roughly two to eleven percent degradation in performance, depending on update model and workload. This cost is comparable to the costs of data structure consistency checking normally included in fault tolerant software. Guarding all of shared memory was more expensive, five to eighty-seven percent overhead for the same benchmarks, but even full shared memory protection may be worthwhile in some environments. Changes to the DBMS required to support the update models have been small. To support *deferred write*, a few hundred lines were added to a roughly 50,000 line DBMS; the other models required less than a hundred lines.

The paper is divided into five sections. The first introduces the DBMS and operating system testbeds on which we have implemented guarding. The second details the update models and describes their implementations. The third section presents some performance results and evaluates the reliability effects of guarding based on previously published statistics about system software errors. The fourth and fifth sections give previous work and conclusions.

2. System Assumptions

In this paper, we assume an extensible DBMS architecture with multiple backend processes. Each DBMS backend process has its own private address space, but all of them share a single common memory region. The shared region contains a lock table, buffer pool, and some in-memory meta data structures used by all of the backend processes. DBMS application programs run in separate address spaces and communicate with the DBMS using messages. POSTGRES, the DBMS used to evaluate guarding, has a process-per-user architecture, but that does not have an impact on guarding. Record-level locking is assumed.

The DBMS used in this study has an unconventional storage manager [Stonebraker87], but the results should still be applicable to more traditional DBMS designs. The POSTGRES storage system has a ''no overwrite'' policy in which data records are not updated directly. An ''update'' marks the current version of the record as invalid and inserts a new version of the record on the same page as the old one. Out-of-date records are removed (or archived) by a background garbage collector process. Guarding is implemented below the level of the POSTGRES storage system and does not take advantage of its no-overwrite property.

POSTGRES is extensible, so code implementing user-defined operators, access methods, and data types can be added to the DBMS. Most extension code will access shared data through a lower-level interface. Locks are set through the POSTGRES lock manager and disk data is accessed through the POSTGRES buffer manager. Normally, the extension code will not have to know about the existence of guarded pages. Some extensions, such as user-defined access methods which have their own page formats, would have to know about and use guarding directly. For example, B-tree access methods had to be modified to unprotect pages before adding or deleting keys.

The Sprite operating system, which we modified to support guarding, is a Unix-based distributed operating system being developed at Berkeley. We chose Sprite as a testbed because the source code was available and well-documented. The guarding implementation assumes that the processor has a software-loaded Translation Lookaside Buffer (TLB).

3. Models for Updating Protected Data

3.1. The Expose Page Update Model

In the *expose page* update model, a DBMS process unguards a record before writing to it and reguards the record after the write. Because write-protection is enforced in hardware at page granularity, unguarding one record also unguards all of the records on the same page. The page granularity of guarding does not imply page granularity for transaction locks, since transaction locks are enforced by software.

Managing protected data in the buffer pool using this model is straightforward. When the data manager updates, inserts, or deletes a record on a buffer page, it unprotects the page with a system call. While the page is unprotected, the page header can be manipulated directly (for updates and deletes) or the data in the record can be changed.

After the DBMS has updated a record, it does not necessarily have to reguard the record immediately. If the record is not immediately reguarded, subsequent updates avoid the cost of unguarding and reguarding the data. Deferring the reguard operation reduces the protection offered to the data, however, and increases the opportunity for the DBMS to ‘‘forget’’ to reguard the page. Our implementation unguards one record at a time, reguarding each record before updating the next.

In the Sprite shared memory implementation, unguarding a page for one DBMS process unguards it for all of the others as well. The guard and unguard system calls change the software page table and modify the TLB (Translation Lookaside Buffer) entry for the affected page. A single software page table is used for a shared memory segment, so, if a second process refers to the unprotected page, it will be allowed access. Sprite has also been modified to include a *guarded_read* system call which allows the DBMS to read a page from disk without leaving it unprotected during the entire I/O operation.

Expose page is best for detecting pointer errors affecting pages containing infrequently updated records. ‘‘Hot’’ pages containing frequently updated records will be unprotected much of the time, so they will receive less benefit from guarding than cold pages. The major costs associated with expose page are an increased number of system calls and the additional TLB operations required to change page protections. If guarding were implemented on a processor with a virtually-addressed cache, changing page protection status from read-write to read-only would require a cache flush. Virtually-addressed caches normally require cache flush operations to change the protection bits for cached data.

3.2. The Deferred Write Update Model

The second model of DBMS data structure protection is designed to leave the record guarded until the end of transaction. When a DBMS process needs to update a record, it copies the record into writable memory and updates the copy rather than update the record in place. After the update is complete, an *InstallData* system call copies the new record value into the protected page. *InstallData* takes as an argument an array of (source address, destination address, length) triples, so several records can be installed with a single system call.

InstallData combines an *unguard* operation, a copy, and an *guard* operation into a single system call. The operating system unprotects the page in the processor’s TLB, copies in the updated record, and reprotects the page in the TLB. If a record must be installed in a page that is no longer in the buffer pool, the DBMS reads the page back into memory before installing the data. Unlike the *expose page* model, *InstallData* never changes process page tables so unprotecting the record for one DBMS process does not unprotect it for the others. *InstallData* changes only the TLB entry and changes it only for the duration of the copy. Since Unix-based operating systems disallow context switches during system calls, no other processes can see the unprotected page.

As in the *expose page* model, *deferred write* offers the DBMS programmer some latitude in deciding when to complete the guarded update. The updated record could be reinstalled immediately after the update. It could also be installed after several updates, at the end of transaction, or after several transactions. In our implementation of the *deferred write* model, guarded records are installed at the end of transaction. Writable records in POSTGRES are kept in unshared process memory, so the data must be installed before transactions running in other DBMS processes can see it.

Some modifications to the POSTGRES buffer manager were required to support deferred write. If the DBMS asks for a record on a page during a scan, the buffer manager has to see if there is already a writable copy of the record. If the record has not been copied, the scan returns a pointer to the protected record. Otherwise, the copy is returned. A hash table tells the buffer manager whether or not there is currently an unprotected copy of the record. If the DBMS decides to update a record, it first tells the buffer manager to make sure the record is writable. The request to make a record writable is logically at the same place the DBMS would upgrade a read lock to a write lock. Hence, the existence of copies did not cause radical changes to the DBMS software.

Deferred write is similiar in some respects to the shadow paging technique used in System R [Lorie77]. Shadow paging is a no-overwrite transaction management technique in which a new block on the disk is allocated for every page modified by a transaction. When the page is evicted from memory or forced to disk, it goes to the new location. The update is committed by remapping the new page into the original page's position in its home relation. Shadow paging was not used in conjunction with write protection in System R and did not provide the error detection benefits of deferred write. Also, unlike shadow paging, deferred write uses in-memory copies and does not affect the allocation of the protected data on the disk.

An in-memory variation of shadow paging could be used in conjunction with guarding to limit copying costs. The deferred write implementation could copy the entire page containing a target record instead of simply copying the record. When the update is complete, the copy could be protected and remapped into the (main memory) position occupied by the original version of the page. In general, this technique will be cost effective only if records are very large. When records are small, making two small copies is faster than copying and remapping a full page.

The *deferred write* update model provides more protection to guarded records than the *expose page* model does. Because *deferred write* updates protected records during a system call, the protected page is never directly addressable to the DBMS process. In *deferred write*, software errors can damage the writable copy of a record, but other records on the same page are less at risk. Installing the update to the wrong place on the page is the only way to corrupt them.

Deferred write has an additional advantage over both *expose page* and conventional DBMS transaction management. When bad software corrupts data, often the damage is not detected immediately. After an error is detected, the DBMS never knows how much data has been corrupted. The detected error could be part of a larger cluster of undetected errors. With guarding and deferred write, however, the DBMS knows that protected data cannot be corrupted until the *InstallData* system call. If a transaction detects that it has corrupted some of its data, it can simply throw away all uninstalled data. If the same transaction also caused undetected damage, that damage will be thrown away (assuming data is installed at the end of transaction). The pages from which the data came (and other buffer pool pages, for that matter) are guaranteed not to have been damaged by this transaction because those pages were never unprotected.

A conventional DBMS handles this situation by aborting the transaction and hoping that the DBMS transaction support removes the effects of undetected errors. Aborting the transaction will remove the damage only if the erring software accurately recorded its updates in the log. Some errors, like those caused by corrupted pointers, are not remedied by recovery protocols. The most practical way for a conventional DBMS to get the same guarantee as the deferred write update model is to invalidate the entire buffer pool after detecting an error.

3.3. The Expose Segment Update Model

The *expose segment* update model is similar to the expose page model, however, protection is added to or removed from all guarded pages at once. When the DBMS makes an *ExposeData* system call, all protected data becomes visible. A second system call, *HideData* returns the protection to all exposed data.

Expose segment provides less protection than the other two models since nothing is protected from the routines which update critical data structures. The reason for using the expose segment model is that it simplifies the management of guarded data in some modules. Using the expose segment model, a DBMS programmer can unprotect data for a procedure and its descendants in the call tree without knowing exactly which protected pages will be written. For POSTGRES, we found the *expose segment* model to be convenient for small, fast, and trustworthy operations that needed access to data on several pages. For example, we use it in a shared memory hash table in the implementation of record-level locking.

To further simplify programming in the *expose segment* model, we use a pre-processor to place calls to *ExposeData* and *HideData* in procedures. The DBMS programmer flags with a keyword any procedure which is to update protected data. The pre-processor adds *ExposeData* and *HideData* calls at the first line and before all return statements in the targeted procedures. The pre-processor eliminates a class of errors in which data is never hidden again after an *ExposeData* call. It also makes adding protection to new data structures very easy.

To implement the *expose segment* update model in Sprite, we modified the part of the operating system that loads the processor's TLB. Normally, the TLB loader forces the DBMS process to take a protection fault if it tries to write to protected data. After an *ExposeData* system call, the Sprite TLB loader allows writes to guarded data. When the data is hidden again, the mappings for any guarded pages still in the TLB must be returned to read-only status.

The implementation is optimized for the case in which few pages are written while the guarded data is exposed. After an *ExposeData* system call, the TLB loader records the page number of the first few guarded pages that are updated in a trace buffer. If only a few pages are ever updated, the trace buffer allows fast reprotection of these pages during *HideData*. If the trace buffer overflows, the entire TLB must be flushed to reprotect the exposed pages.

The expose segment model of guarded update is similar to a conventional protected subsystem. Other protected subsystems (the operating system kernel, for example) require more complicated mechanisms since they are expected to prevent malicious as well as accidental damage.

4. Performance Impact of Guarded Data Structures

Because the DBMS and operating system have to do extra work during updates of guarded records, guarding will decrease DBMS performance for update-intensive workloads. The extra costs involved in guarding include the additional system calls and TLB operations required to change page protections. In the *deferred write* update model, additional processing is required to create and keep track of record copies.

In order to measure the performance impact of guarding, we compared several different versions of POSTGRES using a workload based on the TP1 debit/credit benchmark [Anon85]. In our version of this benchmark, two thousand transactions were run against a small database. Each transaction retrieves a tuple from an account relation, updates the account relation and two other smaller relations (branch and teller), and appends a record to a fourth relation (history). Account is 200 pages long and branch and teller are each one page.

The benchmark database is small in order to allow the DBMS to store the entire database in main memory. We wanted to measure guarding under both a CPU-bound and a disk-bound workload. Since the POSTGRES storage system is optimized for battery-backed main memory, it includes a ‘‘no disk write on commit’’ option which we used for the CPU-bound benchmark. In the CPU-bound case, POSTGRES never writes updated pages to disk so the CPU is saturated. When POSTGRES runs on a system with volatile memory, it must write all modified data pages to disk at transaction commit. In volatile-memory mode, POSTGRES runs at about 25 percent CPU utilization. The benchmarks were run single-user on a DECStation 3100 implementation of the Sprite operating system.

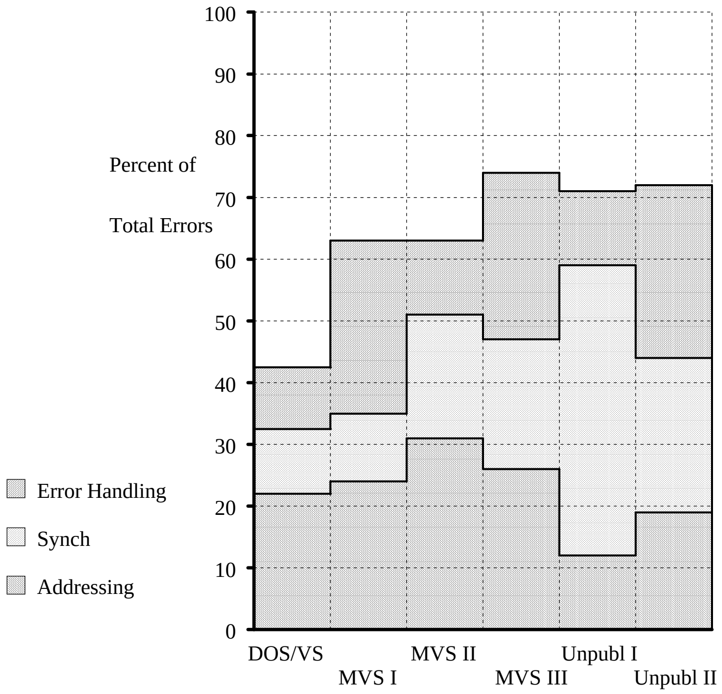
We compared six different versions of POSTGRES. The **normal** version is a vanilla DBMS with no guarding support. The **unprotected copy** version used the deferred write update model but did not protect the pages. Comparing the unprotected copy POSTGRES to normal POSTGRES shows the overhead in deferred write attributable to copy management, but not to write protection. The next three POSTGRES versions each use a different one of the update models described in the paper.

Table 1: Debit/Credit Performance for Guarding In-Memory Database, CPU-Bound

Update Model	Protection Overhead
Normal POSTGRES	0%
Expose Page Guarding	7%
Expose Segment Guarding	10%
Unprotected Copy (no guarding)	6%
Deferred Write Guarding	11%
Full Protection	87%

Table 2: Debit/Credit Performance for Guarding 25% CPU Utilization, Write-Through on Commit

Update Model	Protection Overhead
Normal POSTGRES	0%
Expose Page Guarding	2%
Expose Segment Guarding	3%
Unprotected Copy (no guarding)	2%
Deferred Write Guarding	3%
Full Protection	5%



Graph 1: Summary of Error Study Data

The last POSTGRES version, **full protection**, protects all of shared memory -- including the lock table, some shared memory lookup tables, and the buffer pool. The **full protection** version uses the *expose page* update model to update data in the buffer pool and *expose segment* to update all other data structures.

Tables one and two compare the protection overhead for each of the six program versions. Each benchmark run of two thousand transactions was repeated five times to get an average elapsed time. If the standard deviation of the five elapsed times was greater than one percent of the average, all five runs were repeated. The tables present their results as the percent increase in the average elapsed time caused by the protection mechanism.

The tables show that the least expensive of the three update models for the guarded buffer pool is *expose page*. *Expose segment* is slightly more expensive, probably because *expose segment* requires both a system call and a TLB fault to access protected data while *expose page* only requires a system call. In the disk-bound case, the costs of the different models are roughly the same. Since guarding does not affect disk accesses, it has a large impact only when there is high CPU utilization.

Deferred write has about the same cost as *expose segment*. This cost is divided between the cost of managing record copies and the cost of making guarding-related system calls. Comparing the unprotected copy DBMS to the *deferred write* DBMS shows that much of the expense is related to copy management. *Deferred write* makes only one system call per transaction, so it would be expected to have less guarding-related overhead than the other two techniques. From profile data, we have seen that nearly all of the copy management costs come from allocating, freeing, and searching for record copies in the copy hash table. Because records are small in the benchmark, physical copying does not affect performance.

In the CPU-bound case, the full protection DBMS is significantly slower than the versions that only protected the buffer pool. The difference was more pronounced on read-only transaction workload, since buffer pool protection alone caused *no* measurable decrease in performance. Full shared memory protection caused a 70% to a 116% increase in average elapsed time, depending on the placement of the system calls.

5. Reliability Impact of Guarded Data Structures

In order for guarding to increase reliability, failing software must try to update protected data illegally. If broken software always managed to unguard data structures before corrupting them, guarding would not be effective. Guarding would also have no impact if software failures simply cause the program to halt or produce incorrect results without ever overwriting any data (e.g. deadlock).

We could measure the reliability impact of guarding by running an extensive test suite against the protected DBMS, however, the results of such a test are unlikely to reflect the impact of guarding in a commercial system. Guarding has been implemented for a single research DBMS. The types of errors experienced at our site will not be the same as the commercial fault tolerant systems experience in the field. Also, a measurement study would have to compare one system with guarding to another system without guarding in order to get meaningful results. Comparison is important since, to be cost effective, guarding must detect errors that would not be detected by less expensive means.

A second evaluation alternative is to estimate the effectiveness of guarding using existing software error studies. Graph 1 summarizes some of the results from six studies of software failures in operating systems. The published studies in the table are from MVS ([Velardi84] and two from [Mourad87]), and DOS/VS [Endres75]. These studies are difficult to compare since they were taken at different phases of the development cycle and had classifications which obscured information we need to evaluate guarding. Each of the studies classified errors in slightly different ways.

In graph 1, we have regrouped the categories from each study into a few categories that could be compared across all of the studies. The result is four classes of errors: addressing errors, synchronization errors, error handling errors, and miscellaneous. Addressing errors are the ones of most interest for evaluating guarding. Error handling errors come about when the system failed after being unable to handle an error in a lower level subsystem. The Endres study is different from the others largely because it classified many errors as ‘‘specification errors’’ without going into detail about how they affected the execution of the program.

The studies show that addressing errors make up twenty to thirty percent of the recorded software errors. In these studies, errors are assigned a primary cause (based on available data and the interests of the research team). The secondary effects of the error may involve addressing failures as well, so thirty percent is not necessarily an upper bound.

While this initial evidence is promising, more work is required to show a strong relationship between guarding and reliability. We are currently collecting failure data from a commercial DBMS in order to better characterize DBMS errors. Using this data, we intend to conduct a more complete study using fault injection techniques such as those used in [Chillarege89].

6. Using Guarding to Improve Data Availability

Guarding is a relatively inexpensive way of isolating processes in a multi-process DBMS from one another; this isolation can be used to improve the availability of data after one process fails. Software errors from one DBMS process sometimes destroy data structures in shared memory, forcing all DBMS processes to recover. By reducing the need for multi-process recovery, guarding can improve recovery speed. Because data is either unavailable or less available during recovery, improving recovery speed improves data availability.

To see the impact of multi-process recovery, consider three levels of recovery:

(1) **Single-Process Recovery**: One DBMS process aborts its current transaction and exits. To recover, the process must restart and reinitialize its in-memory data structures. The transaction in progress on the failed process must be restarted and the transaction’s work must be redone.

(2) **Multi-Process Recovery**: All DBMS processes must reinitialize. In conventional write ahead logging systems, some undo/redo log processing is required. The DBMS buffer pool is discarded, and must be reloaded from disk. Communications with the client processes must be reestablished.

(3) **Media Recovery**: As above, but the contents of disk must be restored from dump tape before multi-process recovery begins.

Each level of recovery removes the effects of a different class of errors. Media recovery affects errors which corrupt the disk. Multi-process recovery is required when shared memory is corrupted. Single-process recovery can cleanup from errors affecting process local memory. If all DBMS shared memory were guarded, many of the errors that normally require multi-process recovery could be repaired by faster single-process recovery.

For some systems, using single process recovery in place of multi-process recovery will increase the risk posed by undetected errors. The additional risk comes when a transaction manages to commit data with undetected errors. If a second error occurs, multi-process recovery reinitializes the buffer pool and discards the buffer damaged by the first error. If the second error is cleaned up with single process recovery, the damaged buffer is not discarded.

In storage systems based on shadow paging and in the POSTGRES storage system, even this situation cannot occur. POSTGRES uses a no-overwrite update policy instead of a conventional log [Stonebraker87]. As a consequence, any updated buffer page must be written to stable main memory or to disk before the end of transaction. Once written to stable store, the corrupted buffer will be used in recovery whether the buffer pool is discarded or not.

In a write ahead logging system, multi-process recovery is more reliable than single process recovery only when the system log is not corrupted. If both the data and the log record are corrupted, multi-process recovery will not remove the damage since data in the log must be used for recovery. For example, if the DBMS miscalculated a data value, the corrupted value would be written into the log. Addressing errors can obviously corrupt buffer pool data without generating bad log records, but most of these errors are detected by guarding.

In summary, guarding shared data structures makes it possible to use single process recovery in place of multi-process recovery. Using faster single-process recovery will increase data availability during recovery. In some storage systems, single process recovery increases the risk of unrecoverable damage, but the increased risk is small. The exact increase in risk depends on how effective guarding is at preventing errors and how long errors remain undetected after they occur.

7. Previous Work

An alternative to protecting shared data structures with guarding is to keep those data structures in one address space and the clients of the data structures in another. In order to make such an architecture practical, a fast cross-address-space procedure call mechanism like that of the Taos operating system [Bershad89] is required. The Taos Lightweight Remote Procedure Call (LRPC) is optimized for RPC-style communication in which only a few parameters are passed between caller and called routine. The Service Request Block (SRB) mechanism in the MVS/XA [IBM] operating system is similar to LRPC. An SRB is a high priority thread of control which can be created in a remote address space. Both LRPC and SRB use a fast path through the scheduler and some shared memory to reduce overhead.

Guarding provides the same kinds of protection against non-malicious damage as does an address space boundary. However, access to read-only records is faster than would be possible in a separate address space implementation. Since database workloads often require the DBMS to scan through large amounts of data before selecting some for update, faster read performance is a distinct advantage.

Tandem's process pair mechanism [Bartlett81] also relies on multiple address spaces to prevent propagation of software errors. The Tandem data manager has a primary and ''hot spare'' process executing at the same time on different machines. The primary executes all transactions and sends checkpoint messages to the spare. If the primary fails, the spare can reconstruct the data manager's state from the checkpoint messages. While errors might propagate within the primary, they are less likely to propagate to the spare.

While process pair prevents the same kinds of errors as guarding does, it is much more expensive. Keeping the spare up to date requires resources for sending and processing checkpoint messages. Worse, the implementation of the checkpoint protocol is non-trivial. Modifications to the DBMS may affect the checkpoint protocol, making them expensive to implement and test. Finally, the model does not help detect errors. The primary and spare both have large, unprotected buffer pools. An undetected pointer error can damage a buffer without making the primary turn over control to the spare. The corrupted buffer will eventually corrupt permanent data.

The 801 System [Chang88] uses page protection bits to provide operating system support for DBMS locking and logging, rather than using page protection to increase fault tolerance. A data manager running on the 801 does not set locks explicitly. Memory management hardware detects a read or a write to an unlocked buffer and the DBMS traps to the operating system. The operating system then sets locks and implements physical logging of 128 byte subpages. To support fine-grain locking, the 801 memory management unit provides write-protection at subpage granularity. The same hardware would support subpage granularity guarding.

Unlike a system using guarded data structures, the 801 treats any attempt to write to one of its buffers as legitimate. By moving responsibility for locking from the DBMS to the operating system, the 801 is losing information available to the DBMS about which data is updated erroneously. If a bad pointer causes a write to an unlocked buffer, the 801 locks the buffer and logs it normally. Under the same circumstances, a guarded system would immediately halt the transaction.

Implementing protected operations such as locking in the operating system is one alternative to the *expose segment* model of guarding. However, installing the DBMS code in the operating system makes the operating system vulnerable to errors in the installed code. Guarding gives the DBMS implementor more freedom to decide what code is reliable enough to have access to protected data. More debugging support is available for user programs than for the operating system, so implementing protected subsystems in the DBMS is more practical than implementing them in the OS.

The *expose segment* update model implementation provides some of the same protections as a protected subsystem mechanism without requiring any special hardware or restricting the designer's choice of programming environment. Existing protected subsystem mechanisms often rely on special memory management hardware [Schroeder72], [Wulf74], or type-safe languages [Lampson80]. The *expose segment* update model can be implemented on any processor which uses a software-loaded TLB. Of course, guarding is designed to protect against accidental damage not malicious damage. Existing protected subsystem mechanisms were designed to protect against both.

We chose to implement the virtual memory support required for guarding by modifying the operating system. It would also be possible to support guarding using the Mach external pager [Young87]. Implementing guarding directly in the operating system should make guarding more efficient.

8. Conclusions and Future Work

We have modified the operating system and data manager in order to limit software error propagation in DBMS shared memory. Write protecting the data manager's buffer pool allows early hardware detection of addressing-related software errors. Guarding reduces the complexity of software failure by preventing errors from propagating to protected data structures. Guarding techniques can also improve recovery speed since limiting potential error propagation decreases the amount of work required at recovery time. While any DBMS could use these techniques, they are especially important to a extensible DBMS such as POSTGRES. With a guarded system, one person using (or developing) new access methods or data types has smaller impact on the availability and reliability achieved by his or her peers.

In general, the performance impact of guarding is comparable to the impact of other software techniques for detecting software errors, such data structure verifiers or array bounds checks. Guarding can be implemented efficiently by taking advantage of processors with software-loaded TLBs. For read-only workloads, guarding provides the DBMS with additional protection at no extra cost. For update-intensive workloads, experiments have shown that the additional CPU demand caused by guarding is only a few percent when small records are updated. In the future, we will use page remapping techniques as a method for reducing copy cost for large records.

In deciding whether or not to guard data structures, system designers face a tradeoff between potential reliability and availability improvement and a small but measurable performance loss. For some systems, no reliability gain will be worth any loss in performance. Others may be willing to accept the small performance loss in order to achieve any reliability improvement. Still other systems may want the option of switching from guarded to normal operations at different points in the system lifetime or for different customers. An important second area of future work is the development of techniques for quantifying the reliability impact of guarding. These techniques will help system designers or administrators make an informed decision about whether or not to use guarding.

Over time, trends in system cost will tilt the performance/protection tradeoff in the favor of guarding. Falling memory prices are increasing the sizes of disk caches like the DBMS buffer pool. Some data in the cache will remain unused for long periods of time. It is essential that bad writes into this data be caught at the time of the error rather than the first time the data is used. Meanwhile, as processors become faster, the additional processing demands caused by guarding will become less of a concern.

Acknowledgements

Margo Seltzer, David Bacon, Mike Olson, and Ramon Caceres read early drafts of the paper and provided useful suggestions. Discussions with Ram Chillarege and with members of the Sprite group were also extremely helpful.

References

[Anon85] Anon et. al, ‘‘A Measure of Transaction Processing Power,’’ Technical Report 85.1, *Tandem Corporation*, January 1985.

[Bannerjee87] J. Bannerjee, W. Kim, H. Kim, H. Korth, ‘‘Semantics and Implementation of Scheme Evolution in Object-Oriented Databases,’’ *Proc. ACM SIGMOD Conference*, pages 311-322, December 1987.

[Bartlett81] J. Bartlett, ‘‘A NonStop Kernel,’’ *Proc. Eighth Symposium on Operating Systems Principles*, pages 22-29, December 1981.

[Bershad89] B. Bershad, T. Anderson, L. Lazowska, H. Levy, ‘‘Lightweight Remote Procedure Call,’’ *Proc. Twelfth Symposium on Operating Systems Principles*, pages 102-122, December 1989.

[Carey86] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, E. Shekita, ‘‘The Architecture of the Exodus Extensible DBMS,’’ *Proc. IEEE International Workhop on Object-Oriented Systems*, September 1986.

[Chang88] A. Chang, M. Mergen, ‘‘801 Storage: Architecture and Programming,’’ *ACM Trans. on Computer Systems*, 6(1):28-50, February 1988.

[Chillarege89] R. Chillarege and N. S. Bowen. ‘‘Understanding Large System Failure -- A Fault Injection Experiment.’’ *Digest 19th International Symposium on Fault Tolerant Computing*, pages 356-363, June 1989.

[Endres75] A. Endres. ‘‘An Analysis of Errors and Their Causes in Systems Programs.’’ *IEEE Trans. on Software Engineering*, SE-1(2):140-149, June 1975

[Gray90] J. Gray. ‘‘A Census of Tandem System Availability between 1985 and 1990.’’ *IEEE Trans. on Reliability*, 39(4):409-418, October 1990.

[Gupta90] R. Gupta. ‘‘A Fresh Look at Optimizing Array Bounds Checking,’’ *Proc. of ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*, pages 272-282, June 1990.

[IBM] *MVS/Extended Architecture Overview*, Publication Number GC28-1348, IBM Corporation.

[Lampson80] B. Lampson, D. Redell, ‘‘Experiences with Processes and Monitors in Mesa,’’ *Comm. of the ACM*, 23(2):105-117, February 1980.

[Lorie77] R. Lorie, ‘‘Physical Integrity in a Large Segmented Database,’’ *ACM Trans. on Database Systems*, 2(1):91-104, March 1977.

[Mourad87] S. Mourad, D. Andrews, ‘‘On the Reliability of the IBM MVS/XA Operating System,’’ *IEEE Trans. on Software Engineering*, SE-13(10):1135-1139, October 1987.

[Ousterhout88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, B. Welch, ‘‘The Sprite Network Operating System,’’ *IEEE Computer*, 21(2):23-36, February 1988.

[Schroeder72] M. Schroeder, J. Saltzer, ‘‘A Hardware Architecture for Implementing Protection Rings,’’ *Comm. of the ACM*, 15(3):157-170, March 1972.

[Stonebraker86] M. Stonebraker, L. Rowe, ‘‘The Design of POSTGRES,’’ *Proc. ACM SIGMOD Conference*, June 1986.

[Stonebraker87] M. Stonebraker, ‘‘The POSTGRES Storage System,’’ *Proc. Very Large Data Bases Conference*, pages 289-300, September 1987.

[Velardi84] P. Velardi, R. Iyer, ‘‘A Study of Software Failures and Recovery in the MVS Operating System,’’ *IEEE Transactions on Computers*, C-33(6):564-568, June 1984.

[Wulf74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack, ‘‘HYDRA: The Kernel of a Multiprocessor Operating System,’’ *Comm. of the ACM*, 17(6):337-345, June 1974

[Young87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron, ‘‘The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System,’’ *Proc. Eleventh Symposium on Operating Systems Principles*, pages 63-76, December 1987.