Using Codewords to Protect Database Data from a Class of Software Errors

Philip Bohannon[1] **Rajeev Rastogi**[1] **S. Seshadri**[1]
Avi Silberschatz[1] **S. Sudarshan**[2,*]

[1] Bell Laboratories, Murray Hill, NJ
{bohannon,rastogi,seshadri,avi}@research.bell-labs.com
[2] Indian Institute of Technology, Bombay, India
sudarsha@cse.iitb.ernet.in

Abstract

Increasingly, for extensibility and performance, special-purpose application code is being integrated with database system code. Such application code has direct access to database system buffers, and as a result, the danger of data being corrupted due to inadvertent application writes is increased. Previously proposed hardware techniques to protect from corruption require system calls, and their performance depends on details of the hardware architecture.

We investigate an alternative approach which uses codewords associated with regions of data to detect corruption and to prevent corrupted data from being used by subsequent transactions. We develop several such techniques which vary in the level of protection, space overhead, performance, and impact on concurrency. These techniques are implemented in the Dalí main-memory storage manager, and the performance impact of each on normal processing is evaluated. Novel techniques are developed to recover when a transaction had read corrupted data caused by a bad write, and gone on to write other data in the database. These techniques use limited and relatively low-cost logging of transaction reads to trace the corruption, and may also prove useful when resolving problems caused by incorrect data entry and other logical errors.

1. Introduction

As hardware gets more reliable, software errors are often the greatest threat to database system availability [20, 7], even in standard systems where database data is protected by process boundaries from errors in application programs. Increasingly, however, for extensibility and performance, special-purpose application code is being integrated with database system code. Extensible databases allow third party vendors and users to add new data types and storage methods to the database engine [19]. Performance-critical applications may require the performance which can be achieved by directly accessing the data stored in a main-memory database [9, 4]. In either case, due to the high cost of inter-process communication, direct access to database internal structures such as the buffer cache is critical to meeting the performance needs of these applications. Thus, database availability can be affected not only by software errors in the DBMS, but also by errors in application programs. One class of software error which has been shown to have a significant impact on DBMS availability is the "addressing" error [20]. This class of error includes copy overruns and "wild writes" through uninitialized pointers.

Software fault tolerance techniques (see, for example, [15, 18]) attempt to mitigate the damage done when software errors occur in a production environment. One approach to avoiding addressing errors is the use of type-safe languages for user applications. Similar results can be achieved at runtime with the software fault tolerance technique of *sandboxing* [23]. However, type-safe languages have yet to be proven in high-performance situations, and sandboxing may perform poorly on certain architectures (see Section 6, Related Work, for more details). Finally, communication across process domain boundaries to a database server process provides protection, but such communication is orders of magnitude slower than access in the same process space, even with highly tuned implementations [1]. With multi-gigabyte main-memories now easily affordable, one can expect many OLTP databases to be fully cached, decreasing the impact of disk latency on performance and consequently increasing the relative impact of inter-process communication.

[*] The work of S. Sudarshan was performed in part while at Bell Labs.

In [21], Sullivan and Stonebraker investigate the use of hardware memory protection to improve software fault tolerance in a DBMS environment by guarding data in the buffer cache. For example, calls were added to POSTGRES [19] to unprotect the page containing a tuple before it is updated and to reprotect it afterwards. In performance experiments, they found that this protection was relatively inexpensive. The overhead amounted to 7-11% of the processing time using a CPU bound workload which ignored disk latency, and to 2-3% of processing time when disk latency was included. However, a number of factors have motivated us to consider other possible techniques for protecting data in the DBMS. First, memory protection primitives must be accessed through system calls which may be slow [17]. An informal test of several systems available to us confirmed that the performance of mprotect can vary widely among comparable workstations (see Figure 1 in Section 5.1). Second, in a threaded application, threads can access pages unprotected by other threads, decreasing the effectiveness of the scheme. Finally, in a main-memory DBMS, information such as allocation information or other control information need not be stored on the same page as user data. This is the case in the Dalí system in which our experiments are implemented, and it can significantly increase the number of pages which must be protected and unprotected during a single operation.

In this paper, we refer to bytes modified in the course of an addressing error as *direct physical corruption*. Once data is directly corrupted, it may be read by a process, which then issues writes based on the value read. Data written in this manner is *indirectly corrupted*, and the process involved is said to have *carried* the corruption. While this process could be a database maintenance process, we focus on *transaction-carried corruption*, in which the carrying process is an executing transaction. As noted in [21], for a DBMS to effectively guard data from direct physical corruption, it must expose an *update model* by which correct updates can be distinguished from unintended or erroneous updates. In our update model, all updates are in place, and correct updates are ones which use a prescribed interface.

We investigate techniques which protect data by dividing the database into *protection regions* and associating a *codeword* with each such region. Thus, using the prescribed interface ensures that when data in a region is updated, the codeword associated with the region is also updated. (This activity is referred to as "codeword maintenance".) When a wild write or other addressing error updates data, with high probability the codeword value computed from the region will no longer match the codeword maintained for that region.

We present several codeword-based techniques for the prevention or detection of corruption. The first scheme we describe, Read Prechecking, prevents transaction-carried corruption by verifying that the codeword matches the data each time it is read. The Data Codeword scheme, a less expensive variant of Read Prechecking, allows detection of direct physical corruption by asynchronously auditing the codewords.

For detecting indirect corruption, we introduce the Read Logging scheme in which a limited amount of information about each data item read by a transaction is added to the log. Since the data logged consists of the identity of the item and an optional checksum of the value, *but not the value itself*, the performance impact of this logging is limited. The addition of information about reads allows the database log to function as a limited form of audit trail [2] for the DBMS.

Since none of these techniques prevents direct physical corruption, techniques for *corruption recovery* must be employed to restore the database to an uncorrupted state. We introduce the *delete-transaction model*, a model of recovery which focuses on removing the effects of corruption from the database image, and present an algorithm which implements this model as a modification of the Dalí recovery algorithm. Recovery from errors when the error is not immediately detected (for example, not detected until after the transaction commits) is discussed in [5], but the delete-transaction model algorithm presented in this paper is the first concrete proposal we are aware of for defining and implementing corruption recovery in a transaction processing system.

To ascertain the performance of our algorithms for detecting and recovering from physical corruption, we studied the impact of these schemes on a TPC-B style workload implemented in the Dalí main-memory storage manager. Our goal was to evaluate the relative impact of the schemes described above on normal transaction processing. In addition to our schemes, we include a hardware-based protection technique similar to that of [21]. For detection of direct corruption, the overheads imposed cause throughput of update transactions to be decreased by 8%. Prevention of transaction-carried corruption with Read Prechecking costs between 12% and 72%, with the space overheads increasing as performance improves. Detection of transaction-carried corruption with Read Logging costs between 17% and 22%. Using hardware protection decreases throughput in Dalí by about 38%, even on a platform with relatively fast protection primitives.

The remainder of the paper is organized as follows. Our system model is discussed in Section 2. Section 3 describes schemes which prevent or detect corruption. Section 4 presents techniques for recovery when corruption is detected after the fact. Section 5 describes a performance study of several of the algorithms presented in the paper, and Section 6 describes related work. Section 7 concludes the paper and discusses future work.

2. System Model

This paper assumes a database model in which database data is directly mapped into the address space of applications. In addition to user data, control information such as lock tables and log buffers may also be mapped into the address space of the application. Note that even if user applications are not allowed direct access to database data, any multi-threaded database engine will follow a similar model in that database server processes will access the buffer cache in shared memory [8].

Certain of the algorithms we develop depend on details of the logging and recovery model of the DBMS, and in these cases, the logging and recovery model of the Dalí main-memory storage manager [3] is used. The logging and checkpointing code in Dalí is modified to implement the protection schemes for the performance study in Section 5. Updates in Dalí are done in-place, and updates by a transaction must be bracketed by calls to the functions beginUpdate and endUpdate. Each physical update to a database region generates an undo image and a redo image for use in transaction rollback and crash recovery. Undo and redo logs in Dalí are stored on a per-transaction basis (*local logging*). When a lower-level operation is committed, the redo log records are moved from the local redo log to the system log tail in memory, and the undo information for that operation is replaced with a logical undo record. Both steps take place prior to the release of lower level locks. A copy of the logical undo description is included in the operation commit log record for use in restart recovery.

As a main-memory system, Dalí is only page-based to the extent that it is convenient for tracking storage use and for the efficient layout of fixed-size records. For example, allocation information is not stored on the same page as tuple data, and extra free-space for expansion need not be reserved on each page. Benefits of this approach include efficient use of space, and the ability to store objects larger than a page contiguously, and thus access them directly without reassembly and copying.

2.1. The Dalí Multi-Level Recovery Algorithm

Dalí implements a main-memory version of *multi-level recovery* [12]. A multi-level transaction processing system consists of n logical *levels of abstraction*, with operations at each level invoking operations at lower levels. Transactions themselves are modeled as operations at level n, with level 0 consisting of physical updates.

The contents of the system log tail are *flushed* to the stable system log on disk when a transaction commits, or during a checkpoint. The *system log latch* must be obtained before performing a flush, to prevent concurrent access to the flush buffers. The stable system log and the tail are together called the system log. The variable end_of_stable_log stores a pointer into the system log such that all records prior to the pointer are known to have been flushed to the stable system log. While flushing physical log records, we also note which pages were touched ("dirtied") by the update which generated the log record. This information about dirty pages is noted in the *dirty page table* (dpt).

All redo actions are physical, but when an operation commits, an operation commit log record is added to the redo log, containing a logical undo description for that operation. At system recovery, these records are used so that logical undo information is available for all committed operations whose enclosing operation (which may be the transaction itself) has not committed. For transaction rollback during normal execution, the corresponding undo records in the transaction's local undo log are used instead.

Since the database is assumed to fit in main-memory, Dalí does not have a buffer manager and does not write pages back to disk except during a *checkpoint* operation. During a checkpoint, dirty pages from the in-memory database image are written to disk. In fact, two checkpoint images, Ckpt_A and Ckpt_B, are stored on disk, as is the checkpoint anchor, cur_ckpt, which points to the most recent valid checkpoint image for the database. During subsequent checkpoints, the newly dirty portions of the database are written alternately to the two checkpoint images (this is called ping-pong checkpointing [6]).

Information about active transactions is stored in an *active transaction table*, referred to as the ATT. Due to local logging, the entry for each transaction in the ATT contains local undo and redo logs. In addition to the database image, a copy of the ATT with the local undo logs and a copy of the dirty page table (dpt) are stored with each checkpoint.

Note that physical undo information is moved to disk only during a checkpoint. The undo information is taken by the checkpointer directly from the local undo logs of each transaction. (Thus, physical undo log records are never written to disk for transactions which take place between checkpoints.)

Restart recovery starts from the last completed checkpoint image, and replays all redo logs, repeating history physically. When the end of the log is reached, incomplete transactions (those without transaction commit or abort records) are rolled back, using the logical undo information stored in either the checkpointed ATT or operation commit log records. Due to multi-level recovery, the rollback is done level by level, with all incomplete operations at level i being rolled back before any at level $i + 1$.

3. Codeword Protection

In this section, we introduce two codeword-based schemes for detection and prevention of corruption: Read Prechecking and Data Codeword. We also discuss the Hard-

ware Protection scheme which is included in the performance study in Section 5. When Hardware Protection prevents an addressing error, a trap is issued to the process and the offending write is not completed. Thus, this scheme prevents direct physical corruption. By contrast, the codeword schemes can only detect direct physical corruption during a subsequent audit, and this is the strategy taken with the Data Codeword scheme. However, direct corruption only does damage when the corrupted data is read by a subsequent process, and this indirect corruption can be prevented using the Read Prechecking scheme.

Schemes for computing codewords for data are well known, and selection of a good scheme is not a topic addressed in this paper. In our implementations, the codeword is the bitwise exclusive-or of the words in the region. Thus the i'th bit of the codeword represents the parity of the i'th bit of each word on the region.

Control Structures

Corruption can occur not only on the database image, but also on transient database control structures such as lock information, etc. Since the interfaces to such data are typically less uniform than the interface to user data, using the techniques described below to protect control structures would entail a significant individual implementation effort for each structure covered. For this reason, we do not include protection of these control structures in this study.

Protecting and unprotecting an entire segment was used to provide protection for control structures in [21]. While we are aware of no segment-level protection mechanisms in the UNIX platforms available to us, were such a mechanism available, it could easily be combined with the schemes for protection of persistent data studied in this thesis. However, even with such a mechanism, protection of control structures in [21] was found to cost approximately ten times as much as protection of data buffers in CPU-bound tests.

Hardware Protection

While not a codeword scheme, the hardware protection scheme is included in our performance study as a point of comparison. Since all updates in Dalí are in-place, the hardware protection scheme we implemented most closely resembles the Expose Page Update Model of [21]. On a call to beginUpdate, the page (or possibly pages) being updated are unprotected, and are reprotected at the call to endUpdate.

3.1. Read Prechecking

An alternative to preventing direct corruption of data is preventing the use of that corrupted data by a transaction. To accomplish this, the consistency between the data in a protection region and its codeword is checked during each read of persistent data. We now present the details of this scheme.

A protection latch is associated with each protection region and acquired exclusively when data is being updated, or when a reader needs to check the region against the codeword. At endUpdate time, the undo image stored in the log and the current value of the updated region are used to update the codeword before the protection latch is released. A flag, codeword-applied, is stored in the undo log record for a physical update to indicate whether the associated change to the codeword has been applied. This flag is set at beginUpdate and reset at endUpdate. If a rollback is necessary when the flag is set, the undo image for this update should be applied without updating the codeword. When reading data, the protection_latch is taken in exclusive mode and the codeword for the contents of the region which contains the data to be read is computed and compared with the stored codeword.

3.2. Data Codeword

Detecting (but not preventing) direct physical corruption can be accomplished with a variant of the Read Prechecking scheme described in Section 3.1. The maintenance of the codewords is accomplished in the same manner; however, the check of the codeword on each read is dropped in favor of periodic audits. The process of auditing is nothing more than an asynchronous check of consistency between the contents of a protection region and the codeword for that region. This can be carried out just as if a read of the region were taking place in the Read Prechecking scheme.

Since prechecks are not being performed, and audits are asynchronous, it makes sense to use significantly larger protection regions. In this case the protection latch may become a concurrency bottleneck. If so, a new latch, the codeword latch, may be introduced to guard the update to the actual codewords, and the protection latch for a region need only be held in shared mode by updaters. During audit, the protection latch must be taken in exclusive mode to obtain a consistent image of the protection region and associated codeword. In particular, data is audited during the propagation to disk by the checkpointer (or at page-steal time in a page-based system).

4. Corruption Recovery

In this section, we consider how to recover from physical corruption which is detected rather than prevented. Since recovery is necessarily more complicated than prevention, it may seem an unnecessary effort to design recovery algorithms for this case, even if the corruption detection mechanism is significantly more efficient than comparable corruption prevention mechanisms. However, in the fault injection study performed by Ng and Chen to study the reliability of a non-volatile buffer cache in a DBMS, the addi-

tion of hardware protection only reduced the incidence of data corruption from 2.7% to 2.3%, indicating that a significant risk exists that physical corruption will go undetected [16]. Since they used a hardware protection scheme following [21], some of this corruption may have occurred during the time the page was unprotected. Such errors may be detected by the codeword schemes in this paper if the area being corrupted, while on the same page, is not specified as part of the update when the prescribed interface (beginUpdate/endUpdate) is called.

In other cases, however, the corruption may have been introduced through the prescribed routines, in which case it does not meet the definition of physical corruption used in this paper. For these errors, codeword audit procedures will not be able to automatically detect the corruption. However, if other audit mechanisms such as those described in [10] or other asserts within the DBMS are available to determine the location and a lower bound on the time of the error, the recovery mechanisms described in this section can aid in the subsequent recovery.

4.1. Models of Corruption Recovery

We define three models of corruption recovery: the *cache-recovery* model, the *prior-state* model, and the *delete-transaction* model. Using the cache-recovery model with the Read Prechecking scheme and the two Data Codeword schemes is discussed in Section 4.2, and an algorithm which implements the delete-transaction model is given in Section 4.3.

In the cache-recovery model, direct physical corruption is removed from cache pages, assuming that indirect corruption has not occurred, and because of that, corrupt data values are not reflected in any log records. This form of recovery is invoked when a precheck fails in the Read Prechecking scheme or when an audit detects a codeword error in one of the Data Codeword schemes. By auditing pages before they are propagated to disk, we ensure that the disk image is free of corruption and thus repairing the corrupted cache image can be accomplished by applying standard recovery techniques to the region of data corrupted. We omit the details due to lack of space.

In the prior-state model, the goal is to return the database to a transaction consistent state prior to the first possible occurrence of corruption by replaying logs which were generated prior to that point. Most commercial systems support this model (see, for example, [13]). We do not discuss it further.

In our final model, the delete-transaction model, we assume that corruption is dealt with by deleting the effects of certain transactions from the database image. Any transaction that read corrupted data must be deleted from history, and any data that such a transaction wrote after reading corrupt data is treated as being corrupted by the transaction. The identity of deleted transactions is then returned to the user to allow manual compensation for the effects of these transactions. Thus we assume that corruption is detected relatively quickly, and that the amount of corrupted data is limited. We do not attempt to analyze the speed at which corruption may spread, since it is dependent on the details of the application, the DBMS implementation, and the initially corrupted data. Furthermore, it is up to the user to deal with any real-world actions or communication with external systems which has occurred during the execution of the deleted transactions. (Note that in the prior-state model, it is up to the user to deal with compensating for *all* transactions which have occurred after the corruption, rather than just the ones determined to be possibly affected.)

To implement a recovery algorithm for this model, it must be clearly understood what it means to "delete a transaction from history". One possible interpretation would be to allow any serializable execution of the remaining, undeleted, transactions. However, this definition is not acceptable, since the values read by other transactions, and thus the values exposed to the outside world, might change in the modified history.

To define correctness in this model, we consider two transaction execution histories, the original history, H_o, and the *delete history,* H_d. Each history is specified by the reads and writes issued by each transaction. In H_d, all reads and writes of certain transactions no longer appear. These histories include the values read or written by each operation, and for a given operation, the value read or written in H_d is the same as for the corresponding operation in H_o. A delete history is *conflict-consistent*[1] with the original history if any read in H_d is preceded by the same write which preceded it in H_o. Similarly, H_d is *view-consistent* with H_o if each read in H_d returns the value returned to it in H_o. A correct recovery algorithm in the delete-transaction model recovers the database according to a delete history which is conflict- or view-consistent with the original history. Note that it follows from this definition that in a *conflict*-consistent delete history, the final state of any data item written by a transaction in the delete set will have the value it had before being written by the first deleted transaction.

4.2. Read Logging

In order to trace the indirect physical corruption as required by recovery in the delete-transaction model, we introduce the idea of *limited read logging*. When a data item is read, the identity of that item is added to the transaction log. Should it be determined through an audit or other means that certain data is corrupt, the read log records can help determine if any subsequent transactions have in fact read the corrupt data by serving as a form of audit trail [2]. The

[1] Note that the notions of *conflict-* or *view-consistency* are distinct from the standard notions of *conflict-* or *view-equivalence*.

read log records combined with the rest of the log allow the transaction log to be used as a mechanism for tracing the flow of indirect corruption in the database. Since log records in Dalí are physical, we log reads at the physical level also. Thus when data is read, the identity of that data is logged as a start point and a number of bytes. For efficiency, the data logged as read may overestimate the amount actually read.

Generating Checkpoints Free of Corruption

Since Read Logging supports recovery from indirect corruption, it becomes crucial that the disk image be free not only of direct corruption, but indirect corruption as well, so that a correct recovery does not require loading an archive image of the database. Thus, when propagating dirty pages from memory to disk, it is not sufficient to audit the pages being written. Even if none of the dirty pages has direct physical corruption, it is possible that a "clean" page has direct corruption, and a transaction has carried this corruption over to a page that was written out. Thus the checkpoint would have data that is indirectly corrupted.

The correct way of ensuring that the checkpoint is free of corruption is to create the checkpoint, and after the checkpoint has been written out, audit every page in the database. If no page in the database has direct corruption, no indirect corruption could have occurred either. We can then certify the checkpoint free of corruption.

This technique cannot be directly applied to page flushes in a disk-based system, since it amounts to auditing all pages in the buffer cache before any write of a dirty page to disk (at page steal time). However, a similar strategy can be followed if a set of pages are copied to the side, and then an audit of all pages is performed before writing them to the database image on disk. To ensure that direct physical corruption does not escape undetected, a clean page which is being discarded to make room for a new page must also be audited.

4.3. Delete-Transaction Model

The delete-transaction model of corruption recovery is tightly integrated with restart recovery. On detecting an error, we simply note the region(s) failing the audit, and cause the database to crash, allowing corruption recovery to be handled as part of the subsequent restart recovery. For our delete-transaction model recovery algorithm, we need a checkpoint which is update-consistent in addition to being free from corruption. However, in Dalí, a checkpoint being used for recovery is not necessarily update-consistent until recovery has completed (that is, physical changes may only be partially reflected in the checkpoint image, and certain updates may be present when earlier updates are not).

The algorithm to obtain an update-consistent checkpoint in Dalí is similar to the audit procedure for the Deferred Maintenance codeword scheme, and uses a portion of the redo log and ATT to bring the checkpoint to a consistent state before the anchor is toggled and the checkpoint made active. Once performed, the checkpoint is update-consistent with a point in the log, CK_end. We omit the details.

Recovery Algorithm

The main idea of the following scheme is that corruption is removed from the database by refusing during recovery to perform writes which could have been influenced by corrupt data. In order to do this, the transactions which performed those writes must, at the end of the recovery, appear to have aborted instead of committed. Certain other transactions may also be removed from history (by refusing to perform their writes) in order to ensure that these "corrupt" transactions can be effectively removed, and thus ensuring the final history as executed by the recovery algorithm is consistent with a delete history obtained from removing the "corrupt" transactions from the original execution (see Section 4.1).

Recovery must start from a database image that is known to be non-corrupt. Note that since errors are only detected during checkpointing or auditing, we may not know exactly *when* the error occurred; the error may have been propagated through several transactions before being detected. The algorithm below conservatively assumes that the error occurred immediately after Audit_LSN, the point in the log at which the last clean audit began.

Two tables, a CorruptTransTable and a CorruptDataTable are maintained. A transaction is said to have *read corrupt data* if the data noted in a read or write log record of that transaction is in the CorruptDataTable.

Restart recovery consists of the redo phase followed by the undo phase, as follows:

Redo Phase: The checkpointed database is loaded into memory and the redo phase of the Dalí recovery algorithm is initiated, starting the forward log scan from CK_end.

During the forward scan, the following steps are taken (any log record types not mentioned below are handled as during normal recovery):

- If a read or write log record is found, then if this record indicates that the transaction has *read corrupted data*, then the transaction is added to CorruptTransTable (where it may already appear).
- If a log record for a physical write is found, then there are two cases to consider:
  1. The transaction that generated the log record *is not* in CorruptTransTable: In this case, the redo is applied to the database image as in the Dalí recovery algorithm.
  2. The transaction that generated the log record *is* in the CorruptTransTable: In this case, the data it would have written is inserted into CorruptDataTable. However, the data is *not updated*.

- If a begin operation log record is found for a transaction that is not in CorruptTransTable, then it is checked against the operations in the undo logs of all transactions currently in CorruptTransTable. If it conflicts with one of these operations, then the transaction is added to CorruptTransTable. This ensures that the earlier corrupt transaction can be rolled back. If it does not conflict, then it is handled as in the normal restart recovery algorithm.
- If a logical record such as commit operation, commit transaction or abort transaction is found, the record is ignored if the transaction that generated the log record is in CorruptTransTable. Otherwise, the record is handled as in normal restart recovery.
- When Audit_LSN is passed, all data noted to be corrupt by the last audit is added to CorruptDataTable.

Undo Phase: At the end of the forward scan, incomplete transactions are rolled back. As in the normal Dalí algorithm, undo of all incomplete transactions is performed logically level by level. Note that at the end of the redo phase, each transaction in CorruptTransTable has a (possibly empty) undo log, containing actions taken by the transaction before it first read corrupted data. During the undo phase, these portions of the corrupt transactions are undone as if the corrupt transactions were among those in progress at the time of the crash.

Checkpoint: The recovery algorithm is completed by performing a checkpoint to ensure that recovery following any further crashes will find a clean database free of corruption. If the checkpoint were not performed, a future recovery may rediscover the same corruption and in fact additionally declare transactions that started after this recovery phase to also be corrupted. Note that this checkpoint invalidates all archives. The log may be amended during recovery to avoid this problem, but this scheme is omitted for simplicity.

Discussion

Following Section 4.1, the database image at the end of the above algorithm should reflect a delete history that is consistent (in this case, conflict-consistent) with the original transaction history. To see (informally) that this is the case, first observe that all top-level reads of non-deleted transactions read the same value in the history played during recovery as in the original history. This is because any data that could possibly have been read with different values was previously placed in CorruptDataTable, and top-level reads must be implemented in terms of reads at the physical-level where corruption is tracked. The second observation is that the database image is consistent and contains the original contents plus the writes of those transactions which do not appear in the delete set. This follows from the correctness of the original recovery algorithm, and the fact that the initial portion of corrupted transactions can be rolled back during the undo phase along with normal incomplete transactions to produce a consistent image. This is ensured since we do not allow any subsequent operations which conflict with these operations to begin.

Extension: Codewords in Read Log Records

If codewords are stored in read log records, then detection of indirect corruption becomes more precise. In particular, the CorruptDataTable can be dispensed with, and instead, the definition of *reading corrupt data* given above is replaced by a definition in which a transaction read corrupt data if either of the following cases hold:

1. A codeword is stored in a read log record, and it does not match the computed codeword for the corresponding region in the database being recovered.
2. A codeword is stored in a write log record (indicating that it should be treated as a read followed by a write) and the codeword does not match the computed codeword for the corresponding region in the database.

A second benefit of storing codewords in read log records is that in the case of a true system failure (as opposed to one caused by a failed audit) it is possible to detect physical corruption which occurred after the last audit but before the crash. More precisely, physically corrupt data will be detected if any transaction read it, since during recovery the codeword for these transactions will not match the database image being recovered. Thus, if codewords are present, the corruption recovery algorithm should be executed not only when an error is detected, but also on every restart.

Note that the modified algorithm produces a recovery schedule which is *view-consistent* with the original history, thus not propagating corruption when the corrupt transaction wrote the same data to a data item as it would have had in the delete-history.

5. Performance

The goal of our performance study was to compare the relative cost of different levels of protection, for example detection versus prevention, as well as comparing different techniques for obtaining the same level of protection. In each case, we are interested in the impact of the scheme on normal processing as opposed to the time taken for recovery. Corruption recovery is expected to be relatively rare, and the time required is highly dependent on the application and workload. The algorithms studied were implemented in the DataBlitz Storage Manager, a storage manager being developed at Bell Labs based on the Dalí main memory storage manager.

Platform	pairs/second
SPARCstation 20	15,600
UltraSPARC 2	43,000
HP 9000 C110	3,300
SGI Challenge DM	8,200

Table 1. Performance of Protect/Unprotect

5.1. Performance of mprotect

Before describing the results of our study of protection schemes, we begin by looking at the relative performance of memory protection primitives on commonly available UNIX platforms. In Table 1, we evaluate the basic performance of the memory protection feature on a number of hardware platforms locally available to us. In each case, 2000 pages were protected and then unprotected, and this was repeated 50 times. The number reported is the average number of these *pairs* of operations which were accomplished per second. Even this limited sample indicates the variability in the performance of mprotect: the HP 9000 C110, which has about twice the integer performance of the SPARCstation 20 (170.2 SPECint92 for the HP as opposed to 88.9 for the Sun[2]), gives less than one fourth the performance of the SPARCstation when performing mprotect/unprotect operations.

5.2. Workload

The workload examined is a single process executing TPC-B style transactions. The database consists of four tables, Branch, Teller, Account, and History, each with 100 bytes per record. Our database contained 100,000 accounts, with 10,000 tellers and 1,000 branches. The ratios between record types are changed from those specified in TPC-B, in order to increase the size of the smaller tables and thus limit the effects of CPU caching on these tables. The benchmarks were run on an UltraSPARC with two 200Mhz processors, and 1 gigabyte of memory. All tables are in memory during each run, with logging and checkpointing ensuring recoverability. In each run, 50,000 operations were done, where an operation consists of updating the (non-key) balance fields of one account, teller and branch, and adding a record to the history table. Transactions were committed after 500 operations, so that commit times do not dominate.[3] Each test was run six times, and the results averaged. The results are reported in terms of number of operations completed per second.

[2]From the database at http://performance.netlib.org. It was not possible to compare these numbers for all machines, as only SPECint95 numbers are available for the newer systems.

[3]The alternative was to design a highly concurrent test with group commits, introducing a great deal of complexity and variability into the test.

5.3. Results

In Table 2, a representative selection of the algorithms discussed in this paper are shown, along with the average number of operations per second the algorithm achieved in our tests, and the relative slowdown of the algorithm compared to the baseline algorithm, which is just the system running with no corruption protection. Our experiments show that detection of direct corruption can be achieved very cheaply, with a 8% overhead, using data codeword protection.

The Read Prechecking scheme appears in the table several times due to a time-space tradeoff between space used for codewords and the size of protection domains. for this scheme. We present the performance of Prechecking with a small domain size is economical at a 12% cost, depending on the acceptability of a 6% space overhead. Read logging lowers the space overhead, but raises the cost to 17%, which is significant, but may be worthwhile, since automatic support for repairing the database can then be employed. Logging the checksum of the data read, which increases the accuracy of the corruption recovery algorithms, adds 5% to the cost, bringing it to 22%. Memory protection using the standard *mprotect* call on an UltraSPARC costs 38%, more than double the performance hit of codeword protection with read logging. Finally, prechecking with large domain sizes fares very poorly.

By monitoring the number of mprotect calls for the hardware-based scheme, we determined that on average operations updated about 11 pages. Only 4 tuples are touched by an operation, and the extra page updates arise from updates to allocation information and control information not residing on the same page as the tuple. Thus, this number may be significantly smaller for a page-based system, which would improve the performance of Hardware Protection and Read Prechecking relative to the detection schemes. However, even if a factor of three improvement is realized, the variability in performance of mprotect described in Section 5.1 will more than erase this gain on some systems.

Our conclusion from these results is that some form of codeword protection should be implemented in any DBMS in which application code has direct access to database data. Detection of direct corruption is quite cheap, and while limited, is still far better than allowing corruption to remain undetected in the database. Other levels of protection may be implemented or offered to users so that they may make their own safety/performance tradeoff.

6. Related Work

Sullivan and Stonebreaker [21] use the hardware support for memory protection to un-protect and re-protect the page accessed by an update. By writing special system calls into

Algorithm	Corruption		Ops/Sec	% Slower
	Direct	Indirect		
Baseline	None	None	417	0%
Data CW	Correct	None	380	8.5%
Data CW w/Precheck, 64 byte	Correct	Prevent	366	12.2%
Data CW w/ReadLog	Correct	Correct	345	17.1%
Data CW w/CW ReadLog	Correct	Correct	323	22.4%
Data CW w/Precheck, 512 byte	Correct	Prevent	311	25.4%
Memory Protection	Prevent	Unneeded	257	38.2%
Data CW w/Precheck, 8K byte	Correct	Prevent	115	72.4%

Table 2. Cost of Corruption Protection

the Sprite operating system, and making use of the Translation Lookaside Buffers on their hardware platform, they found that page protection could be turned on and off relatively cheaply. In contrast, the new techniques introduced in this paper do not require special operating system or hardware support, easing portability of the DBMS. More importantly, the performance of our schemes is not limited by the operating system's implementation of the mprotect system call (see Section 5.1).

Presumably, type-safe languages could be used to provide protection from direct physical corruption However, C and C++ are still dominant for CAD and other high-performance uses of memory-mapped database systems. Sandboxing (see, for example, [23]) provides an alternate technique for protecting data by rewriting object modules, and with only a minor performance impact. However, the object module rewriting must be redesigned for each target architecture, which may be a significant limit on portability, and since the technique requires a number of free registers to perform well, it may not be applicable on architectures without a large register set, such as the Intel x86 architecture. By contrast, our techniques are language and instruction-set independent.

Ng and Chen [16] study the reliability of three different interfaces to a persistent cache – (1) based on the I/O model, (2) based on direct read/write (without memory protection) and (3) based on direct read/write with memory protection, using the POSTGRES database system. They inject a variety of faults (hardware and software) and then check if persistent data has been corrupted. They conclude that the three interfaces provide a similar degree of protection. What is more important to note is that about 2.5% of the crashes due to the injected faults resulted in persistent data being corrupted, a rather high number for a database system. Thus, techniques to detect and recover from physical corruption are important, even in the absence of application code with direct access to the database buffers.

Küspert [10] presents a number of specific techniques for detecting corruption of DBMS data structures. These techniques are *ad-hoc* in the sense that they are designed for specific DBMS storage structures. Taylor, Morgan and Black [22] provide some theoretical structure to the design of data structures that can recover from certain failures. However, this work is not in the context of a DBMS, and does not apply to corruption of general application data, since it handles only components such as pointers and counts.

Finally, we note that commercial databases may well have implemented techniques for detecting corruption; however, such information is not publicly available. Our page-based direct corruption detection scheme is inspired by a codeword scheme for protecting telephone switch data [11], but as far as we are aware, neither codeword based detection and recovery techniques nor read logging techniques are present in any published description of either a research or commercial database system.

7. Conclusions and Future Work

We have described a variety of schemes for preventing or detecting physical corruption using codewords, and an algorithm for tracing and recovering from physical corruption under the delete-transaction model. We have presented a performance study comparing alternative techniques of corruption detection and recovery. Our study demonstrates that detection of direct physical corruption is economical to implement, that transaction-carried corruption can be prevented cheaply if enough space is available for small protection domains, and that detection of transaction-carried corruption for later correction through read logging imposes about a 17% overhead on update transaction performance. However, this technique opens up interesting possibilities in tracing errors through the database system and aiding in their correction. For a non-page-based system such as Dalí, the new schemes are much cheaper than using the UNIX memory protection primitives for every update. Furthermore, since the codeword schemes depend on simple integer operations, we expect them to be easily portable, to perform consistently over all platforms, and to scale in speed

with the integer performance of new machines, while hardware based protection may be much slower on certain systems with expensive system calls.

We believe our techniques for physical corruption will be of increasing importance since applications are increasingly being provided direct access to persistent data. Since it is relatively cheap, we believe implementors of database systems in which application code has direct access to database buffers should always provide detection of direct physical corruption as a minimum.

We believe the delete-model recovery algorithm is the first concrete proposal for integrating recovery from cross-transaction errors into the crash recovery subsystem of a DBMS. The model of tracing the indirect effects of errors is applicable for other forms of corruption. When an error in the database results from incorrectly coded application programs or due to incorrect user input, we define this to be *logical* corruption, based on the intuition that the error is introduced at a higher level of abstraction than addressing errors. Unlike physical corruption, direct logical corruption cannot be efficiently detected (unless the violation of an integrity constraint causes the transaction which would enter the corruption to roll back). Thus, logical corruption may remain in the database until a user or auditor notices the error. Recovery from such corruption can be very difficult, with problems ranging from accurate tracing of corruption over much longer times to dealing with external actions associated with transactions. We are nevertheless convinced that 1) automatic tools can be a significant aid in this process, following the outline of [5], and 2) as with the implementation of read logging for the delete-transaction model, such tools can benefit significantly from support within the DBMS.

Acknowledgments We would like to thank Dennis Leinbaugh for insightful discussions on codeword maintenance.

References

[1] B. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[2] L. A. Bjork, Jr. Generalized audit trail requirements and concepts for data base applications. *IBM Systems Journal*, 14(3):229–245, 1975.

[3] P. Bohannon, J. Parker, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. Distributed multi-level recovery in a main-memory database. In *Proceedings of the Fourth International Conference on Parallel and Distibuted Information Systems, Miami Beach, Florida*, 1996.

[4] P. Bohannon, R. Rastogi, D. Lieuwen, S. Seshadri, A. Silberschatz, and S. Sudarshan. The architecture of the dali main memory storage manager. *Journal of Multimedia Tools and Applications*, 4(2):115–151, Mar. 1997.

[5] C. Davies, Jr. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[6] D. J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 1–8, Boston, Mass., June 1984.

[7] J. Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Trans. on System Reliability*, 39(4):409–418, Oct. 1990.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[9] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main-memory storage manager. In *Proc. of the Int'l Conf. on Very Large Databases*, 1994.

[10] K. Kuspert. Principles of error detection in storage structures of database systems. *Reliability Engineering*, 14:275–290, 1986.

[11] D. Leinbaugh, November 1994. Personal communication.

[12] D. Lomet. MLR: A recovery method for multi-level systems. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 185–194, 1992.

[13] K. Loney. *ORACLE8 DBA Handbook*. Osborne McGraw-Hill, 1998.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, Mar. 1992.

[15] D. Morgan and D. Taylor. A survey of methods for achieving reliable software. *IEEE Computer*, 10(2), Feb. 1977.

[16] W. Ng and P. Chen. Integrating reliable memory in databases. In *Procs. of the International Conf. on Very Large Databases*, pages 76–85, Aug. 1997.

[17] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware. In *USENIX Summer 1990 Conference Proceedings*, pages 247–256, 1990.

[18] B. Randell. System structure for software fault tolerance. *IEEE Computer*, 10(2), Feb. 1977.

[19] M. Stonebraker and L. Rowe (eds). The POSTGRES papers. Technical report, UCB, Elec.Res.Lab, Memo No.M86-85, rev. Jun.1987., Nov. 1986.

[20] M. Sullivan. *System Support for Software Fault Tolerance in Highly Available Database Management Systems*. PhD thesis, University of California, Berkeley, Jan. 1993.

[21] M. Sullivan and M. Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Procs. of the International Conf. on Very Large Databases*, pages 171–179, 1991.

[22] D. Taylor, D. Morgan, and J. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, 6(6):585–594, Nov. 1980.

[23] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, Asheville, North Carolina, Dec. 1993.