TANDEMCOMPUTERS

Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach

Andrea Borr

Technical Report 84.2
September 1984
PN87607

Robustness To Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach

Andrea Borr

September 1984

Tandem Technical Report 84.2

Tandem TR 84.2

Robustness To Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach

Andrea Borr

ABSTRACT

Since attention first turned to the problem of database recovery following system crash, computer architectures have undergone considerable evolution. One direction such evolution has taken is toward fault-tolerant, highly available, distributed database systems. One such architecture is characterized by a single system composed of multiple independent processors, each with its own memory. This paper examines the inadequacy of both the traditional definition of system crash and the conventional approaches to crash recovery for this architecture. It describes an approach to recovery from failures which takes advantage of the multiple independent processor memories and avoids system restart in many cases.

This paper appeared in the Proceedings of the Tenth International Conference on Very Large Data Bases, Singapore, Aug. 1984, pp. 445-453.

TABLE OF CONTENTS

INTRODUCTION

With the emergence of on-line update in transaction processing applications, log-based database recovery techniques have evolved to provide robustness to crash or system failure. Log-based crash recovery techniques have received considerable attention in the literature [4,5,8,9,10].

The strategies adopted by the proponents of these techniques fall into two basic categories. Both postulate the existence of two types of memory [4]:

i. main memory, which is volatile, hence does not survive system failure;

ii. secondary storage, which is stable or non-volatile, hence usually survives system failure.

In the first strategy, a transaction writes an intentions list rather than updating database pages in real-time. The application of a transaction's intended updates to the actual database pages is deferred until transaction commit, at which time the transaction's intentions list is written to a secondary storage log, following which the updates are applied to the actual database pages. If a failure occurs during the application of the intentions list, the recovery procedure consists of restarting the application of the intentions list from the beginning. This technique has been described by Lampson

and Sturgis in [8].

In the second strategy, a transaction effects its database updates in real-time, but a so-called write-ahead log protocol governs the migration of the updated database pages from a memory buffer pool to secondary storage. According to this protocol, described by Gray in [4], no updated data page is permitted to be written to secondary storage before the log records describing the updates to that page have been written to the secondary storage log. At commit time, transaction recoverability is achieved by forcing to stable storage all log records related to the committing transaction.

Using either of the above strategies, database recovery following a crash is characterized by having recourse to the log stored on secondary storage in order to ensure that committed transactions are applied and uncommitted transactions are removed from the database. A difference between the two strategies lies in the type of log information required for crash recovery. In the case of deferred update, only redo information need be logged. In the case of real-time update with write-ahead log, both undo and redo information must be logged [6].

A RE-EXAMINATION OF THE TERM "CRASH"

Central to the strategies used in the conventional approaches to crash recovery is the definition of a crash or system failure as the loss of the contents of main memory [9]. The inadequacy of this definition of system failure becomes evident when applied to a non shared-memory multi-processor architecture. The concept of "main memory" as a unique and shared resource constituting a single point of failure is inappropriate for multi-computer systems. In a system architecture in which multiple independent processors, each with its own memory, are connected to form a single system or node via interprocessor buses or local area network, the use of the term "crash" to denote an all-or-nothing state of the system loses its validity. The term becomes even less meaningful when applied to a long-haul network consisting of multiple shared-memory nodes, or even of multiple multi-computer nodes. Such configurations raise the possibility of partial crashes caused by individual processor failures within a node or caused by node failures within a network. A fault-tolerant system design may allow certain failures within a node to be handled without requiring system restart. If a partial failure does not require system restart, neither should it require full database restart. However, the problem of the total failure or crash of a multi-computer node still remains and must be handled.

A corollary to the generalization of the concept of crash is the generalization of the concept of crash recovery. If, as in the above definition, secondary storage is viewed as the only storage which

survives failures, then crash recovery must be based on a secondary storage log and system restart is required. If, on the other hand, a processor failure does not imply the failure of other processors, then recovery techniques not requiring system restart or recourse to secondary storage are possible. If a portion of the "log" were copied from the memory of one processor to that of another during normal processing, and one of these processors survived the failure of the other, then recovery from the partial system failure could be effected using the "log" information from the memory of a surviving processor while system operation continued "on-line".

Tandem Computers has implemented a multi-processor architecture using the above concepts. The next section presents a brief description of Tandem's system architecture in order to motivate a more general approach to identifying and recovering from both partial and total system failures. Subsequent sections define robustness to single and multiple processor failures in a Tandem system. A discussion of Tandem's implementation of fault tolerance and the evolution of its design follows.

ARCHITECTURAL OVERVIEW

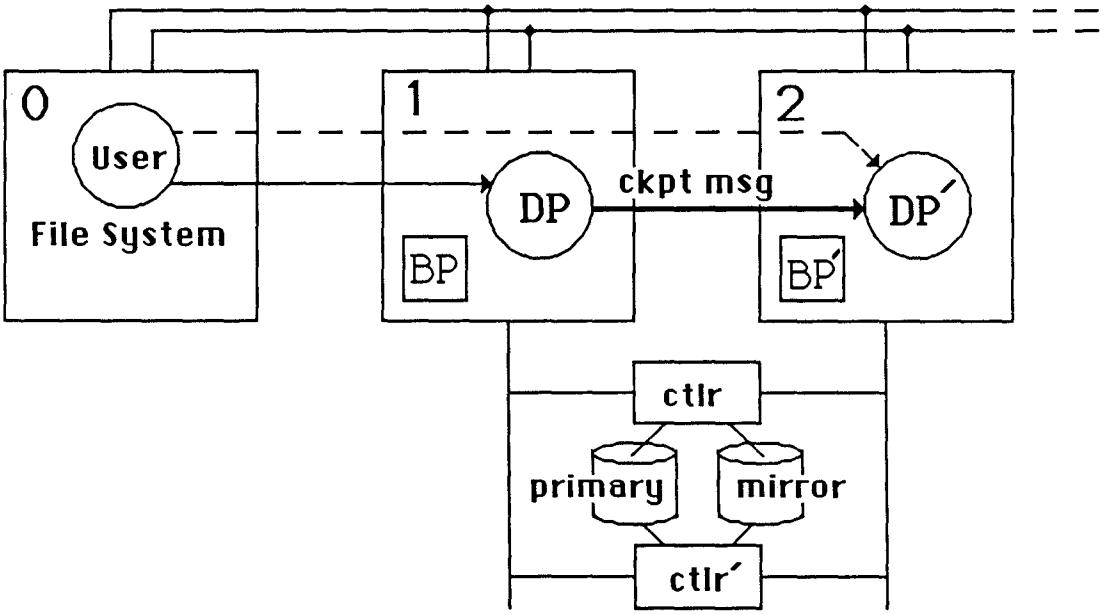
The hardware architecture of a Tandem [TM] system is described in [7]. Illustrated in Figure 1, it is based on multiple independent processors which are interconnected by dual high-speed buses to form a single system (node). The goals of the architecture are fault-tolerance, high availability, and modularity. Hardware redundancy is provided such that the failure of a single module does not disable any other module or disable any inter-module communication. Normally, all components are active in processing the workload. However, when a component fails, the remaining system components automatically take over the workload of the failed component. Each of the (up to 16) processors in a system has its own power supply, memory, and I/O channel. Memory has battery backup power capable of saving system state for several hours in the event of power failure. Each I/O controller is connected to the I/O channels of two processors, and each I/O device, such as a disc drive, may be connected to two controllers. A given disc volume is directly accessible from two processors. Disc volume availability, despite media failures, is provided by optional duplication, or mirroring of drives.

System resources are managed by a message-based operating system, described in [2]. The Message System, a component of the operating system, provides communication between processes executing in the same or different processors, making the distribution of hardware

---

[TM] Tandem and ENCOMPASS are trademarks of Tandem Computers Incorporated.

SYSTEM ARCHITECTURE



- Non - Shared Memory Multiprocessor
- Message Based Operating System
- Distributed Data Management System
- Transaction Concept
- Audited and Non Audited Files

Figure 1: A Tandem node of 2 to 16 processors.

components transparent to processes. Through its Message and File Systems, the operating system makes the multi-computer structure appear as a unified multiprocessor to higher levels of software.

Built on this architecture is a distributed data management and transaction management system called ENCOMPASS [TM]. Described in [1], ENCOMPASS allows data to be distributed across multiple processors and discs within a single node, or even within multiple nodes of a Tandem long-haul network. It supports the transaction concept [6] in this distributed environment. The transaction concept is implemented by means of a log and real-time (as opposed to deferred) update. Transactions can span multiple discs (connected to multiple processors) within the same node or on multiple nodes of a Tandem long-haul network.

Updates to a file may or may not be protected by transaction auditing, depending on the value of the file attribute audited. (Henceforth, the terms "log/logging" and "audit trail/auditing" will be used interchangeably).

ENCOMPASS supports three kinds of structured file organizations: (1) key-sequenced; (2) relative-record; and (3) entry-sequenced. A key-sequenced file is organized as a B-tree on the primary key field. All three file organizations can have alternate keys. Alternate keys are implemented as separate key-sequenced files which "point" to primary file records via a field which contains the value of the primary key. Alternate key files and the primary files which they index can reside

on separate disc volumes. Partitioning files -- by primary key value range -- across multiple disc volumes (possibly on multiple nodes) is also supported. Locking is at file or record granularity.

One of the basic implementation components of ENCOMPASS is a process which acts as a server for files on a particular disc volume. This process, designated the Discprocess, is an example of an I/O process-pair [3]. An I/O process-pair is a mechanism which provides fault-tolerant system-wide access to I/O devices. It consists of two cooperating processes which run in the two processors physically connected to a particular I/O device. One of these processes, designated the primary process, controls the I/O device, handling all requests to perform I/O on the device. The other process, designated the backup process, functions as a stand-by, ready to take over control of the device in case of failure of the primary path to the device. The processor in which the primary I/O process resides is an integral constituent of the primary path to the device. Should the primary's processor crash, the backup process must have information sufficient to take over control of the device. This critical information is sent from the primary process to the backup process during the course of normal processing in the form of so-called checkpoint messages. The process-pair which controls a disc volume is called the Discprocess-pair, or simply Discprocess. Its primary and backup members run in the "primary" and "backup" processors for the disc volume, respectively. The Discprocess has an active rather than a passive backup process. The term active backup process refers to the fact that the information which it receives via checkpoint

messages drives its execution control flow. This is in contrast to a possible alternative design in which the backup process passively receives copies of recently-dirtied portions of the primary process' memory. The active backup concept is central to the design of single fault tolerance, as described below.

From the point of view of a given Discprocess, a "file" is a single partition of an ENCOMPASS "file" (if, indeed, the latter is partitioned). Partitions of key-sequenced primary data files and of alternate key files look alike to the Discprocess: each is structured as a single B-tree. The higher-level concept of a "file" with partitions and/or alternate keys is implemented by the File System. The File System is a set of user-callable procedures which execute in the environment of the user process. These procedures (e.g. OPEN, READ, KEYPOSITION, LOCKREC, WRITE, etc.) accomplish an operation by sending one or more request messages to the appropriate Discprocess(es). In a requester-server model, the invoker of the File System is the requester and the Discprocesses are servers.

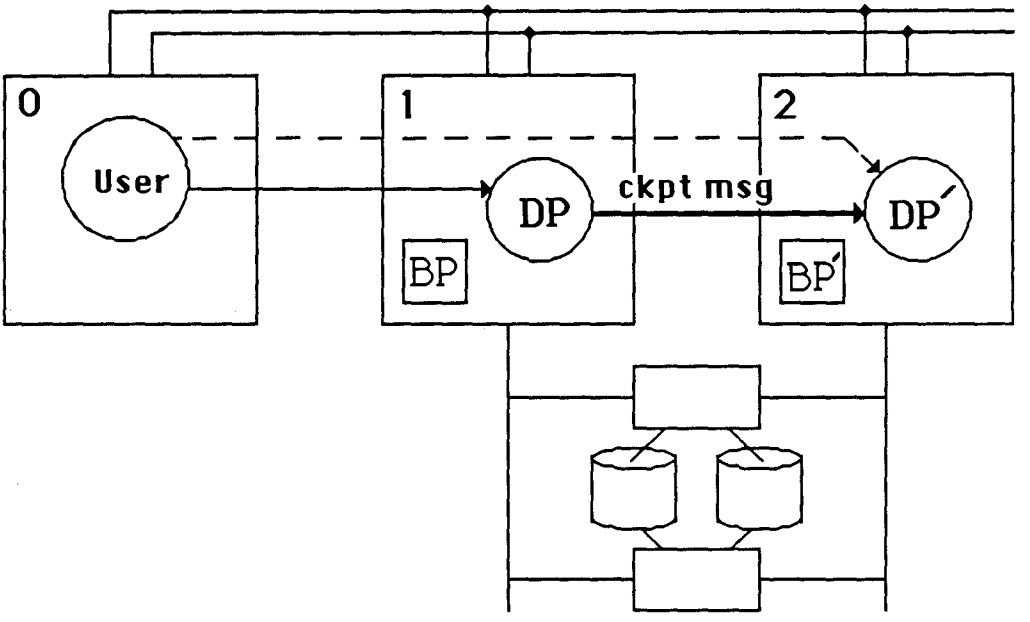
The primary interface to the Discprocess is record-oriented, although a block-oriented interface is also provided. Most update requests result in the updating of a single record within a single block of a given file. In the case of key-sequenced files, however, the possibility that a single request message from the File System could cause a B-tree split or collapse means that the request may be executed as a series of micro update steps. Since an incomplete micro update step series leaves a file structurally inconsistent, robustness

to crash requires a method of assuring its atomicity. This atomicity is provided for both audited and non-audited files, but the means differ, as explained later.

FAILURE MODES

The system architecture described supports fault tolerance for a variety of failure modes other than processor crash. Fault tolerance extends from failures of single hardware components (discs, I/O channels, I/O controllers) to failures of system or application software (programmatic processor halt, user process error, transaction abort). The current discussion, however, will be limited to failures which result in the loss to a single multi-processor node of one or more of its constituent processors. "Loss" in this context means the invalidation of everything stored in the failed processor's memory. This could actually be caused by the failure of any hardware or software component associated with that processor.

The failure model supported can be characterized as fail fast. Consistency checks are an integral part of the system hardware and software. If such a check fails, the bad component is halted. This approach makes failures "clean" and makes it unlikely that a failed component will contaminate other components [3,6].



FAILING CPU(S)	ACTION
CPU 0	Transaction Abort Automatic User Process Restart Automatic Transaction Restart
CPU 1	File System Session Survives Failure DP´ Transparently Takes Over
CPU 2	Lose Tolerance to CPU 1 Failure
CPUs 0-1	No System Restart
CPUs 0-2	No System Restart
CPUs 1-2	Disc Process Pair Crash

Figure 2: Failure modes of a Tandem system.

DEFINITION OF ROBUSTNESS TO SINGLE PROCESSOR FAILURE

The failure of a single processor in the described environment results in the takeover of its functions by the remaining processors. In particular, the failure of a primary Discprocess' processor results in the takeover of its function by the backup Discprocess' processor. If the failed processor contained other primary Discprocesses with different backup processors, then the failed processor may have its work taken over by several other processors.

The Discprocess is designed to provide robustness to single processor failure. This robustness is implemented by means of checkpoint messages sent from the primary process to the backup process during normal processing and a takeover algorithm described later.

The following elements constitute robustness to single processor failure:

(1) "Sessions" between the Discprocess and requesters calling the File System survive the failure of the Discprocess' primary processor. Thus, any file open before takeover still appears open after the takeover.

When updates are not protected by transaction auditing (i.e. updates to non-audited files), a mechanism of tagging messages between the File System and the Discprocess with sequence numbers can optionally be used to guarantee that a request

message is never lost during the takeover and that a non-idempotent operation is never duplicated [3].

When updates are protected by transaction auditing (i.e. updates to audited files), the file open session survives the takeover, but updates executed under that open by a given transaction survive the takeover if and only if that transaction committed before the takeover.

The tolerance of sessions to single processor failure obviates the need to perform system restart in the event of such a failure. For non-audited files, the takeover is transparent to the caller of the File System. For audited files, the takeover is not transparent to the caller of the transaction management system (since transactions may be aborted), but higher-level software makes the abort and restart of such a transaction transparent to the end-user [1].

(2) The structural integrity of both audited and non-audited files on the volume is guaranteed. Thus, if the primary's processor fails in the middle of performing a series of micro update steps to a file, takeover processing restores the file's structure to a consistent state by backing out the steps performed before the failure.

(3) The transactional consistency of the database as a whole is guaranteed. Thus, if a transaction which was uncommitted at the

time of takeover had updated audited files on the failed primary Discprocess' volume, takeover processing aborts the transaction and backs out its changes everywhere (on other volumes on this or other nodes). It should be noted that transaction backout does not include undoing a completed B-tree index operation. In this sense, transaction backout is logical rather than physical.

This is summarized in Figure 3.

Definition of Robustness to SINGLE PROCESSOR FAILURE

IF DP's Primary CPU Fails Then:

- File "Open" Sessions Survive
  - No System Restart
  - No Application Restart
  - Committed Transactions Survive
  - Uncommitted Transactions Aborted and transparently restarted
- File Structural Integrity Restored
  - Non - Audited Files
  - Audited Files
- Global Database Transactional Consistency Restored

Figure 3: Definition of Robustness.

DEFINITION OF ROBUSTNESS TO DISCPROCESS-PAIR CRASH

A Discprocess-pair crash is defined as the simultaneous failure of both its primary and backup processors. The crash of a Discprocess-pair and the failure of its primary and backup processors are viewed as equivalent because the Discprocess is an integral part of the operating system, and as such becomes operational whenever the the processor is restarted. Conversely, whenever a Discprocess primary or backup process detects an internal consistency check failure, it halts its processor in accordance with the fail fast principle. While such a measure might be deemed Draconian in a conventional architecture, this aspect of the design is predicated on the principle that system availability is not compromised by the loss of a single processor. The underlying assumption is that processors fail independently, and that the primary and backup Discprocesses have independent failure modes. Of course, this assumption would be invalidated by the presence of a "hard" (i.e. non timing-dependent) algorithmic bug present in code which would inevitably be executed by either member of the process-pair. The elimination of such bugs has not proven to be an impractical goal, however. This might not be so were the primary and backup processors running in lock-step, or were the backup process passively receiving copies of recently-dirtied portions of the primary process' memory.

When a Discprocess-pair crashes, the situation is similar to the state described earlier as the crash of a shared-memory system. Information stored in memory (in this case the memories of both primary and backup

processors) is lost. Any method of recovery must resort to secondary storage. Furthermore, since "sessions" between the crashed Discprocess-pair and requesters calling the File System have been broken, there is the operational requirement of "restart". The analogy between the elements of robustness to single processor failure and robustness to Discprocess-pair crash is as follows:

(1) "Sessions" between the Discprocess and requesters calling the File System do not survive the Discprocess-pair crash.

(2) The structural integrity of both audited and non-audited files on the volume is guaranteed. Thus, if the Discprocess-pair crashes in the middle of performing a series of micro update steps to a file, crash recovery restores the file's structure to a consistent state.

(3) The transactional consistency of the database as a whole is guaranteed. Thus, if a transaction which was uncommitted at the time of the Discprocess-pair crash had updated audited files on the crashed Discprocess-pair's volume, crash recovery backs out that transaction's changes everywhere (on other volumes on this or other nodes). Conversely, if a transaction which was committed at the time of the Discprocess-pair crash had updated audited files on the crashed Discprocess-pair's volume, but those updates were still in memory buffers (rather than reflected in the corresponding database pages on secondary storage) at the time of the crash, crash recovery retrieves

those updates (from the log) and applies them to the database pages on secondary storage. As in the single processor failure case, transaction backout does not undo completed B-tree index operations.

Definition of Robustness to DISCPROCESS PAIR CRASH

If DP's Primary and Backup CPUs Fail Then:

- File Open Sessions **DO NOT** survive
  - Restart of Processors
  - Database Crash Recovery
- File Structural Integrity Restored
  - Non Audited Files
  - Audited Files
- Global Database Transactional Consistency Restored

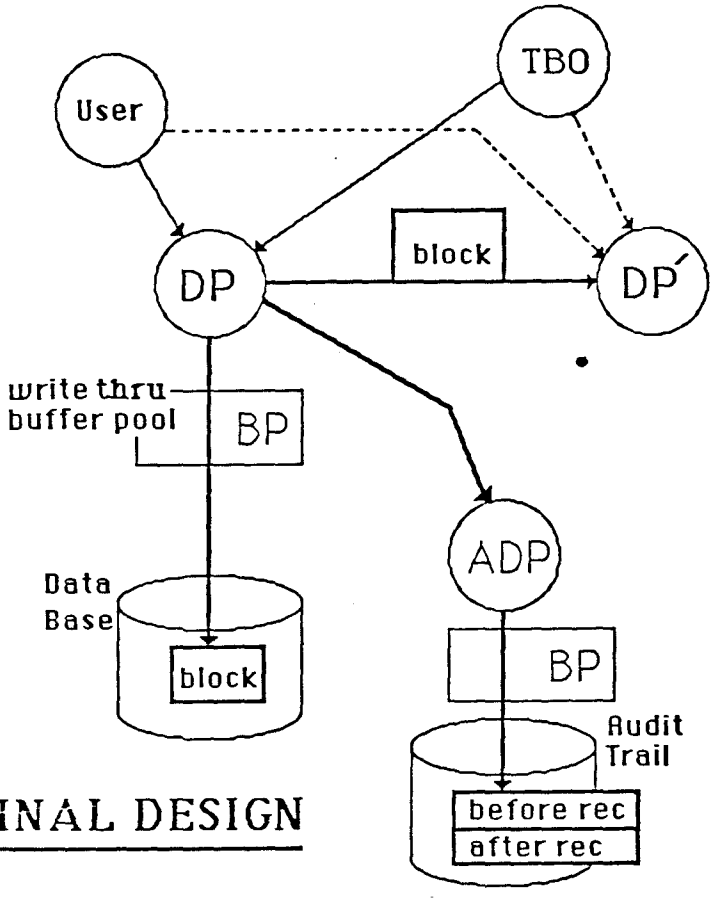
Figure 4: Definition of robustness to discprocess crash.

EVOLUTION OF THE DISCPROCESS DESIGN

The above description reflects a re-architecture of the Discprocess. The goals of the new design were to provide quick recovery from Discprocess-pair crash and less costly tolerance of single-processor failure. The old Discprocess provided robustness to single processor failure as described above. However the old implementation of single processor failure tolerance made a tradeoff in favor of fast takeover recovery from single processor failure at the expense of long recovery in the event of Discprocess-pair crash. The only method of recovery from Discprocess-pair crash was the time-consuming technique of re-loading previously-archived copies of audited database files and "rolling forward" these files to a state of transactional consistency by the application of after-images from the audit trail. The duration of volume unavailability implied by this procedure was justified by the assumption that double processor failure is rare. In actual fact however, double failures are more common than would be predicted by consideration of hardware mean-time-between-failures. Most processor failures are in fact caused by software bugs or operational errors.

Two characteristics of the original design dictated the "roll forward" approach to crash recovery and tolerated single processor failure at the expense of extra disc I/O's and extra checkpoint messages during normal processing. These were as follows (see figure 5):

i. the decision to synchronously write through to disc all updated database pages rather than buffering them in memory;



- Write Through Buffer Pool for simplicity
- Incremental Checkpointing for Single Failure Tolerance
- Carry FORWARD for Single Processor Failure
- Log Forced only by Transaction Commit
- Roll FORWARD for Discprocess Pair Crash

Figure 5: Original DiscProcess design.

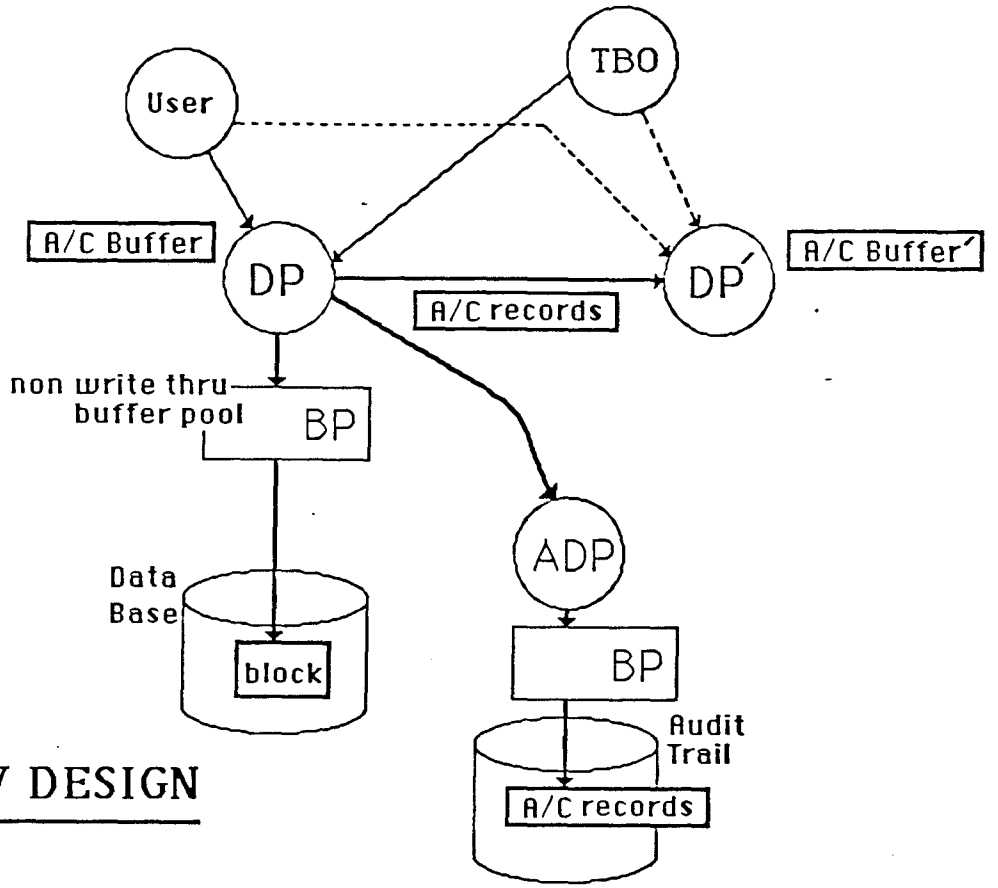
ii. the technique of incremental checkpointing (sending messages from primary to backup Discprocess during normal processing), which provided the backup process with the information needed in the event of the primary processor's failure to carry forward any interrupted series of micro update steps and to continue forward processing on transactions active on the disc volume.

The "write-through cache" was originally conceived as a means of simplifying the implemention of single processor failure tolerance. However it made the write-ahead log protocol [4] infeasible because unacceptable performance would result if every database update resulted in two writes: first, the before-image log necessary for undo in case of failure; second, the modified database page. The absence of write-ahead log made the fast crash recovery technique of in-place rollback of crashed transactions impossible. Writing-through every database update also had negative implications for throughput and response time. Rather than allowing the "piggy-backing" of several in-memory modifications on the same I/O, it meant that each time a page was "dirtied" in memory, it would be written out synchronously (while the application process waited).

Incremental checkpointing is necessary if the backup process is to be prepared -- in the event of the primary processor's failure at any instant -- to carry forward an interrupted series of micro update steps or an interrupted transaction. In the re-architected Discprocess, the approach to takeover is to provide the backup process with enough information to enable it to back out rather than to carry

forward any interrupted series of micro update steps, and to abort rather than to continue processing forward any transaction active on the disc volume. With this approach, deferred checkpointing is possible. According to this technique, the information which would have been sent as synchronous incremental checkpoint messages in the old architecture is instead buffered on the primary process' side and sent as a batch at such times as transaction commit. This technique reduces considerably the cost of single processor failure tolerance in terms of number of messages. In particular, it saves sending checkpoint messages which inform the backup process of memory-only changes in the primary's processor which will not reach secondary storage and which will be backed out anyway in case of takeover. An example of such a change is a buffer dirtied in memory by a transaction which has not yet committed and for which the audit has not yet been forced. The backup process need have no knowledge of such a change since the transaction which caused it will be aborted and backed out (globally) in case the primary Discprocess' processor fails.

In order to explain the takeover algorithm used by the new Discprocess to recover from single processor failure, it is useful to draw an analogy between the use of log records by conventional crash recovery algorithms [4] and the use of checkpoint records during takeover processing. Checkpointing for the new Discprocess is analagous to logging to the backup process. Audit and checkpoint records have a common format; for this reason, they are known as audit/checkpoint records. A typical audit/checkpoint record contains identification of



NEW DESIGN

- Non Write Through Buffer Pool for Write-Ahead-Log*
- Deferred Checkpointing for Single Failure Tolerance*
- BACKOUT and Retry for Single Processor Failure
- Log Forced by Write Ahead Protocol, as well as Transaction Commit
- Auto-Rollback for Discprocess Pair Crash

(*performance)

Figure 7: The new DiscProcess design.

the file, page number within file, record number within page, and the before and after content of the changed record. A version number of the change is stored in both the page header and the audit/checkpoint record to provide idempotence during recovery. Just as conventional log-based crash recovery algorithms use the redo information in the log to bring the database pages up to date with the information which had been logged by the time the system crashed, so the takeover algorithm uses the redo information from checkpoint records received to bring the backup process' memory buffer pool up to date with the information which had been checkpointed by the time the primary's processor failed. Similarly, just as conventional crash recovery proceeds to use logged undo information to back out incomplete requests and uncommitted transactions, so the takeover algorithm uses checkpointed undo information to back out any incomplete series of micro update steps.

At this point, the takeover algorithm terminates and the former backup process begins operation as a primary process by accepting new request messages. Uncommitted transactions have not yet been recovered, however. Any partial micro update step series belonging to an uncommitted transaction has been backed out, and the transaction is prevented from continuing forward processing; but at the completion of takeover such a transaction's updates have not yet been backed out. Locks needed for backout are still held, however. Such a transaction will eventually be backed out by means of logically compensating Discprocess request messages sent by the Backout Process, a system process which extracts the information needed for such requests from

the log. The logically compensating operations requested by the Backout Process are made idempotent by tolerating a "record not found" condition when deleting a record (compensating for an insert) or a "duplicate key" condition when inserting a record (compensating for a delete). Compensating update operations are automatically idempotent.

COMPARISON OF DESIGNS

	Original Design	New Design
Check-Point Content	Physical After Image Of ENTIRE BLOCK (Index or Data)	REDO and UNDO DELTA associated with a Physical Index/Data Block
When Check-Pointed	Prior to Writing each Index or Data Block *	• Phase 1 commit ** • Write ahead log • A/C Buffer Full
Audit Record Content	LOGICAL before and After Images of a changed data record	REDO and UNDO DELTA associated with a Physical Index/Data Block
When sent to audit DP	• Phase 1 Commit • Audit Buffer Full	• Phase 1 commit • Write ahead log • A/C Buffer Full
When Forced to audit disc	• Phase 1 Commit • Audit Buffer Full	• Write Ahead Log • Piggy-backed on commit record (unless previously written)
	* Incremental Checkpointing	** Deferred Checkpointing

Figure 8: Comparison of the two DiscProcess designs.

CRASH RECOVERY FOR THE RE-ARCHITECTED DISCPROCESS

The new Discprocess uses separate mechanisms to provide robustness to Discprocess-pair crash for non-audited and audited files. As previously stated, robustness to crash for non-audited files implies the restoration of structural integrity. For audited files, on the other hand, it implies not only the restoration of structural integrity to individual audited files, but in addition the guarantee of transactional consistency for the database as a whole.

In the case of non-audited files, updates are not protected by transaction auditing. However, loss of structural integrity due to a micro update step series interrupted by Discprocess-pair crash is prevented by use of the so-called Undo Area on the disc volume. This is a small pre-allocated area on the volume which is re-useable for every request. Before beginning a series of micro update steps on a non-audited file (e.g. B-tree block split), a highly-compacted encoding of the intended steps is written to the Undo Area using one I/O. Then if the Discprocess-pair crashes before the operation completes, this undo information is used to back out the incomplete operation when the volume's processors are restarted.

The algorithm used to recover audited files from Discprocess-pair crash is summarized below. It is analagous to typical database crash recovery algorithms used for conventional architectures [4].

Following Discprocess-pair crash, the user first restarts the volume's primary and backup processors. He then initiates the Crash Recovery Process. Crash Recovery obtains a list of those audited files on the crashed volume which were open for write access at crash time. These are the files which are recovered from the log. Log processing during crash recovery consists of a forward and a backward pass.

The forward pass begins at the redo start point. This is a location in the log prior to which all logged updates (redo images) are guaranteed to be reflected in the database. Existence of such a point within a short distance of the end of the log is guaranteed by the periodic execution by each volume's Discprocess of control points. At each control point, currently dirty buffers are flagged. During any spare time between control points, flagged buffers are written out. At the occurrence of the next control point, any flagged buffers not yet written are forced out and newly-dirtied buffers are flagged. (Other systems term this mechanism a "checkpoint"; see [5]). The locations in the log of the latest two control point records are remembered at a known place on the disc volume.

When recovering a given crashed disc volume, Crash Recovery finds that volume's redo start point by obtaining the pointer to its next-to-last control point. When recovering a set of crashed volumes, Crash Recovery starts its forward pass of the log at the earliest redo start point for any of the crashed volumes. Crash Recovery then sends to the Discprocess of a crashed volume all redo log records it finds from that volume's redo start point through the end of the log.

After the redo phase, the backward pass begins. Reading the log backwards from the end, Crash Recovery sends to the appropriate Discprocess those undo log records which represent incomplete micro update step series. When all of the changes represented by these log records have been physically undone, all audited files open on the crashed volume(s) will have been restored to a state of structural integrity. During the same backward pass, Crash Recovery sends to the appropriate Discprocess those undo log records which represent logical operations on data blocks (e.g. record insert, modify, or delete) which were executed by transactions which were uncommitted at crash time. When all of the changes represented by these log records have been logically backed out (i.e. using compensating operations at Discprocess request level), global transactional integrity will have been achieved.

Checkpointing: Normal Processing

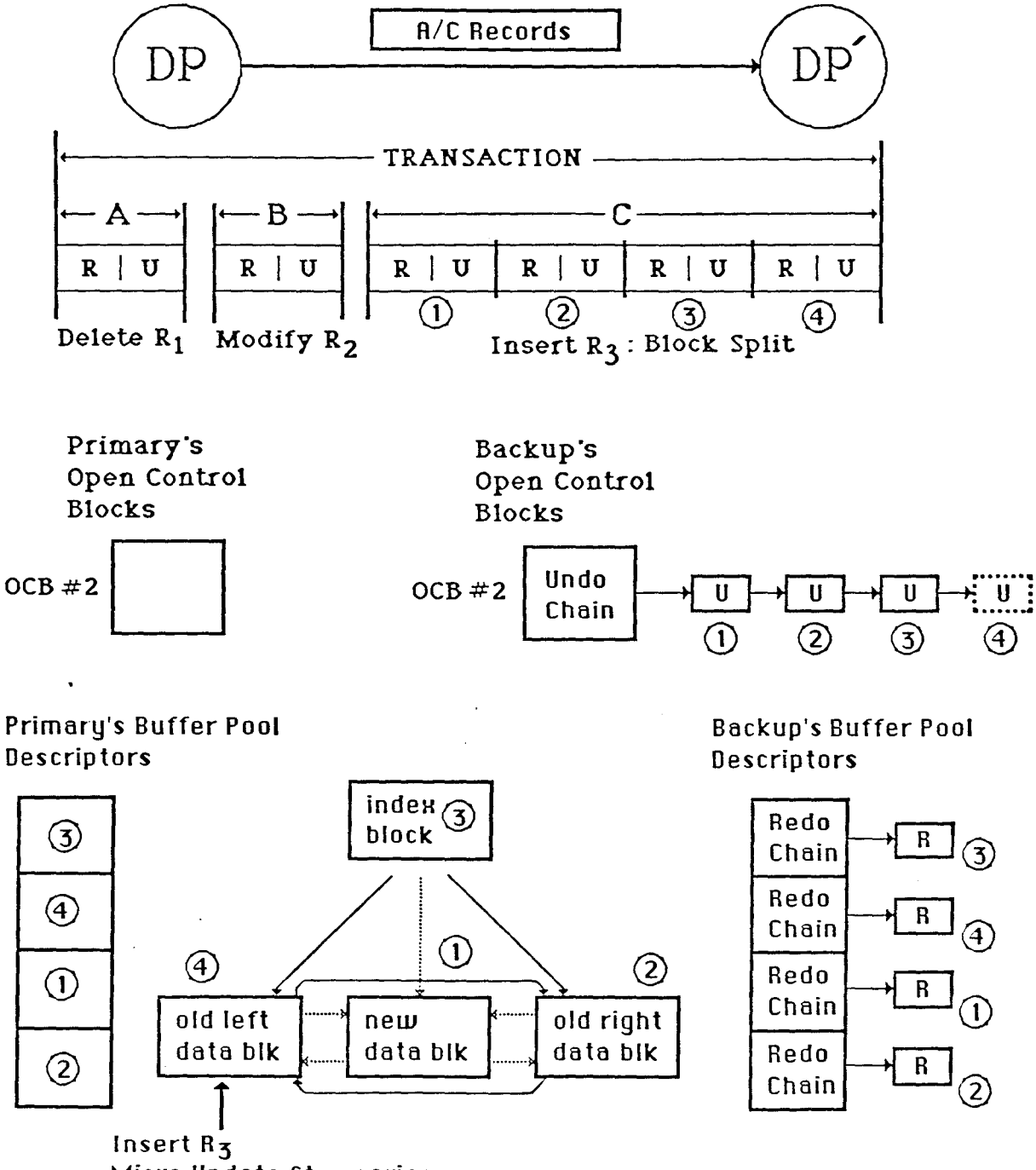
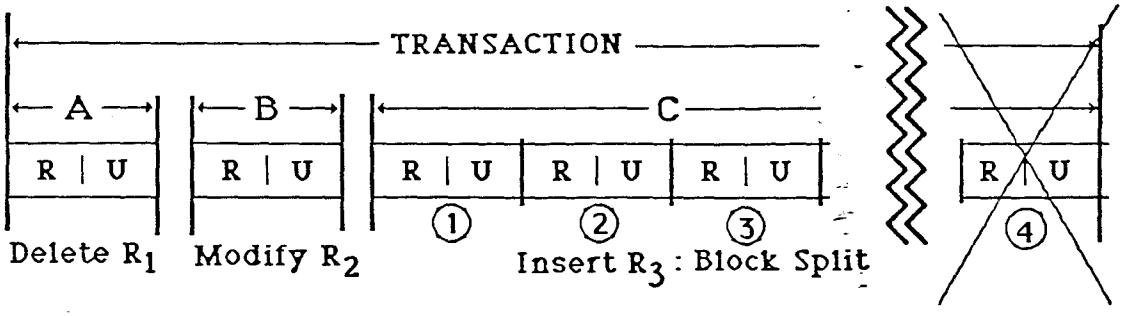


Figure 9. New DiscProcess checkpointing durning normal operation.

CRASH RECOVERY FROM AUDIT TRAIL



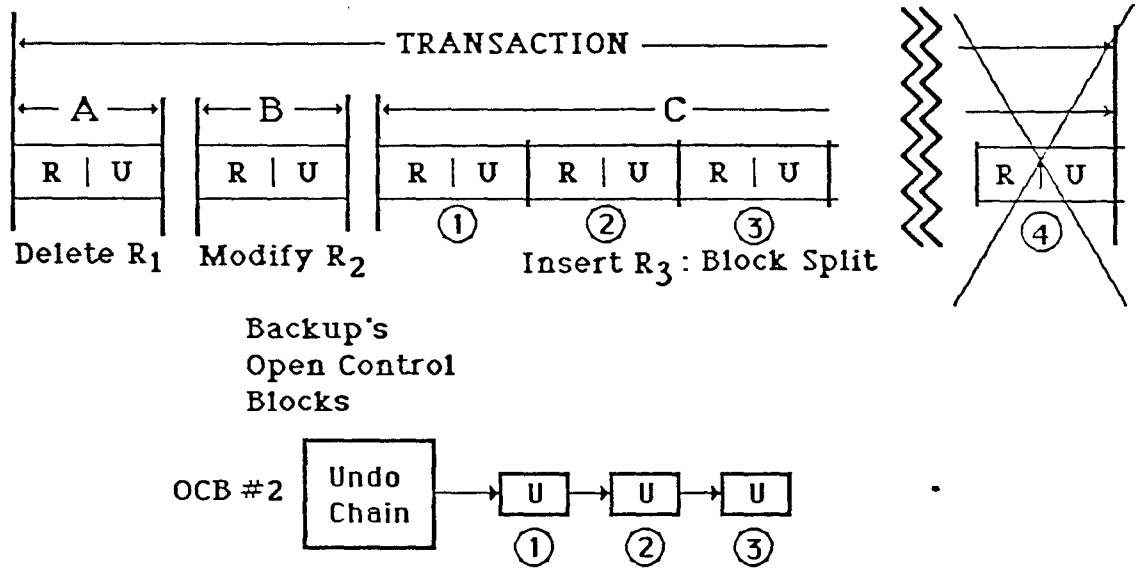
AUDIT TRAIL

CRASH RECOVERY SEQUENCE

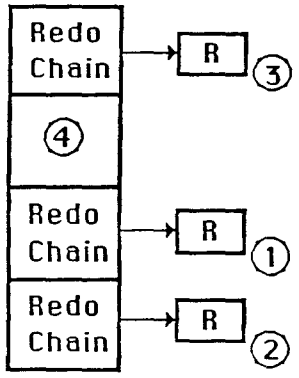
Physical Operations to restore Structural Integrity	1st: Traverse Audit Trail Forward Apply REDO steps to bring DB up to date with Audit 2nd: Traverse Audit Trail Backward Apply UNDO steps to bring DB Back to beginning of micro update sequence
Logical Operations File has Structural Integrity	3rd: Continue Backward Traverse To Logically - Unmodify R_2 Insert R_1

Figure 10. New DiscProcess crash recovery using audit trail.

Takeover Recovery from Memory of Backup Disc Process



Backup's Buffer Pool Descriptors



Takeover Sequence

Physical Operations to restore structural Integrity	1st:Read in blocks from disc Corresponding to B.P. descriptors and apply REDO steps to bring BUFFER POOL up to date with received checkpoint data 2nd: Traverse OCB's UNDO chain backwards applying UNDO steps
Logical Operations File has integrity	3rd: DP' becomes DP : takeover complete 4th: TBO Process reads Audit trail from disc and sends requests to DP to logically Unmodify R_2 Insert R_1

Figure 11. New DiscProcess crash recovery using memory checkpoints.

CONCLUSIONS

The concepts of "crash" and "crash recovery" have been seen to require generalization in order to find applicability to a non shared-memory multi-processor architecture, in which some processors may survive the crash of other processors in the system. The architecture of the Tandem computer system was described as a case in point. A technique of logging to another processor's memory was described which tolerates single-processor failure and obviates the need to perform system restart. An analogy was drawn between the technique used in a Tandem system to recover from a single-processor failure and conventional crash recovery techniques which rely on a secondary-storage-resident log.

CONCLUSIONS

- In a Multi-computer system
  Single processor failure tolerance
  can be achieved by:
  - Sending log records from the memory of one processor to another
  - Using a takeover algorithm analogous to crash recovery

- Characterisctics of Takeover Recovery
  - No System Restart needed
  - Access to secondary storage log not needed
  - Virtually instantaneous
  - Transparent to application

Figure 12. Summary of conclusions.

ACKNOWLEGEMENTS

Many of the ideas incorporated into the new Discprocess design were originated by Franco Putzolu.

Thanks are due to Jim Gray, Chris Duke, and John Nauman for editorial suggestions whose implementation improved the presentation of this material.

REFERENCES

[1]. Borr, A. J., "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing", Seventh International Conference on Very Large Data Bases, September 1981.

[2]. Bartlett, J. F., "A 'NonStop' Operating System", Eleventh Hawaii International Conference on System Sciences, 1978.

[3]. Bartlett, J. F., "A NonStop Kernel", Proceedings of Eighth Symposium on Operating System Principles, ACM, 1981.

[4]. Gray, J. N., "Notes on Data Base Operating Systems", IBM Research Report: RJ 2188, February 1978.

[5]. Gray, J. N. et al., "The Recovery Manager of a Data Management System", IBM Research Report RJ 2623, August 1979.

[6]. Gray, J. N., "The Transaction Concept: Virtues and Limitations", Seventh International Conference on Very Large Data Bases, September 1981.

[7]. Katzman, J. A., "A Fault-Tolerant Computing System", Eleventh Hawaii International Conference on System Sciences, 1978.

[8]. Lampson, B. and Sturgis, H. E., "Crash Recovery in a Distributed Data Storage System", Xerox Palo Alto Research Center, 1976. Also appears as: Ch. I1, "Springer-Verlag Lecture Notes in Computer Science: Distributed Systems - Architecture and Implementation", Vol. 105, B. W. Lampson, Ed.,1981.

[9]. Menasce, D. A. and Landes, O. E., "On the Design of a Reliable Storage Component for Distributed Database Management Systems", Sixth International Conference on Very Large Data Bases", October 1980.

[10]. Verhofstad, J. S. M., "Recovery Techniques for Data Base Systems", Computer Surveys, Vol. 10, No. 2, June 1978

Distributed by
TANDEM COMPUTERS
Corporate Information Center
19333 Vallco Parkway MS3-07
Cupertino, CA 95014-2599