October 1987

UILU-ENG-87-2264
CSG-73

COORDINATED SCIENCE LABORATORY
College of Engineering

LOCAL CONCURRENT ERROR DETECTION AND CORRECTION IN DATA STRUCTURES USING VIRTUAL BACKPOINTERS

C. C. Li
P. P. Chen
W. K. Fuchs

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

LOCAL CONCURRENT ERROR DETECTION AND CORRECTION IN DATA STRUCTURES USING VIRTUAL BACKPOINTERS

C. C. Li, P. P. Chen, W. K. Fuchs

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Ave.
Urbana, IL 61801

ABSTRACT

A new technique, based on virtual backpointers, for local concurrent error detection and correction in linked data structures is presented in this paper. Two new data structures, the Virtual Double-Linked List, and the B-Tree with Virtual Backpointers, are described. For these structures, double errors can be detected in O(1) time and errors detected during forward moves can be corrected in O(1) time. The application of a concurrent auditor process to data structure error detection and correction is analyzed, and an implementation is described, to determine the effect on the mean time to failure of a multi-user shared-database system. The implementation utilizes a Sequent shared-memory multiprocessor system operating on a shared database of Virtual Double-Linked Lists.

Index Terms- concurrent error detection, data structures, concurrent structure checking

---

This research was supported in part by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-602, and in part by the SDIO/IST and managed by the Office of Naval Research under contract N00014-86-K-0519.

Address of Correspondent

W. Kent Fuchs
Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Ave.
Urbana, IL 61801

(217) 333-9731

I. INTRODUCTION

Linked data structures form an integral part of many software and database systems. Performing error detection and correction to preserve the correctness of data structures is important in achieving overall system reliability. To reduce the performance degradation incurred through their use, detection and correction should ideally be executed concurrently with normal processing, and every invocation of these procedures should be completed in O(1) time. If any global checking information (e.g., a global count) is used in detection or correction, then $O(n)$ nodes must be accessed, where n is the number of nodes in the structure, and those procedures cannot run in O(1) time. In addition, since node access time is the major contributing factor to the cost of error detection, the number of nodes accessed should be minimized. The Checking Window concept is introduced in this paper as a method of formalizing these ideas, and as a method of describing local concurrent error detectability as a function of the number of nodes to be checked. To preserve the structural integrity of linked data structures, a new approach to detecting and correcting structural errors, called the virtual backpointer, is also introduced in this paper. The technique is used to construct two new data structures: the Virtual Double-Linked List and the B-Tree with Virtual Backpointers. The Virtual Double-Linked List uses the same amount of storage as the double-linked list from which it is derived. The B-Tree with Virtual Backpointers, derived from the B-tree of order m, requires $m+4$ more fields in each node. It is shown that O(1) local concurrent error detection can be performed for both structures, and that O(1) correction is possible for those errors detected during forward moves through the structures. Correction for those errors detected during backward moves through the structures is in worst case $O(n)$.

The foundation work concerning robust data structures was performed by Taylor, Morgan, and Black [1]. Several techniques have since been presented to achieve robust data structures; however, most achieve error detection in $O(n)$ time. A global count, as used by Taylor, Morgan and Black in the modified(k) double-linked list, the chained and threaded binary tree, and the robust B-tree [1-3], by Munro and Poblete in their isomorphic binary tree [4], by Sampaio and Sauvé in

their robust binary tree [5], and by Seth and Muralidhar in their mod(2) chained and threaded binary tree [6], necessitates, for some errors, a traversal of all the nodes of the structure for error detection. The three pointer tree, as explained by Yoshihara *et al.* [7] requires $O(n)$ time to detect double errors, since a preorder traversal of all the nodes of the tree is performed. Though not indicated in their paper, error detection can be performed in O(1) time using the D-loops within the structure, but only single errors can be detected. Kuspert's work with the separately-chained hash table [8], which is an application of double-linked lists, achieves detection in O(1) time; however, five extra fields must be stored in each node.

A general theory of local detectability and local correctability has been introduced and formalized by Black and Taylor [9], and has been successfully applied to several different types of data structures, including: the spiral(k) list [9], the LB-tree [9-10], the mod(k) list [11], the helix(k) list [12], and the AVL tree [13]. The intention of their work is to be able to correct an arbitrary number of errors in a data structure, provided the errors are sufficiently separated from each other. However, the complexities of the correction algorithms (which include error detection) are typically not O(1).

The organization of this paper is as follows. Section II presents an analysis of local concurrent error detection, giving formal definitions for Checking Windows and local concurrent error detectability. In Section III, the virtual backpointer concept is described and is used to construct two new data structures: the Virtual Double-Linked List and the B-Tree with Virtual Backpointers. The local concurrent error detectability and correctability of each structure is analyzed. Section IV describes a concurrent auditor process as applied to data structure error detection, analyzes its effectiveness in increasing the mean time to failure of a system, and presents the results of an implementation. Finally, Section V summarizes the results.

II. LOCAL CONCURRENT ERROR DETECTION AND CORRECTION

Local concurrent error detection (LCED) is an on-line technique for detecting structural errors in a locality of a currently accessed node in a linked data structure. If the size of the locality is constant and the degree of each node is fixed, then an LCED procedure will run in O(1) time. *Local concurrent error correction* (LCEC) can correct errors detected by an LCED procedure, using another locality of the currently accessed node (not necessarily the same as that used by the LCED procedure). If the size of the locality is again constant, then an LCEC procedure will run in O(1) time. Error detection and correction typically degrade system performance. The degradation is a function of the number of nodes accessed, the number of nodes stored, and the computation required, for detection and correction. For the LCED procedures analyzed here, no extra node accesses are required (except in the initialization phase). Hence, the storage and computation requirements dominate the cost of error detection and correction.

Linked data structures may be modeled as directed graphs. A graph $G = (N, E)$ consists of a finite set of nodes $N = \{N_1, N_2, \cdots, N_n\}$ and a finite set of edges $E = \{E_1, E_2, \cdots, E_m\}$. Each edge $E_i = <N_j, N_k>$ links a pair of ordered nodes in this directed graph (digraph). In the digraph representation of a linked data structure, the nodes represent the data records, and the edges represent the pointers between the records. If all the nodes consist of the same fields, then the data structure is said to be *uniform*. A *move* from a node N_j to a node N_k is possible if there exists an edge E_i between them, and is represented as $N_j \rightarrow N_k$. Then N_k is *reached* from N_j by *following* E_i. A *traversal* is a series of moves starting at a root node or header of a structure that accesses part or all of the data structure.

An LCED procedure is invoked to detect structural errors whenever a move attempts to follow a pointer, which may be a forward pointer, a backward pointer, or a virtual backpointer (Section III). That is, the LCED procedure attempts to *verify* the move. Thus, it is on-line, or concurrent with normal structure access.

The errors considered in this paper are those that affect the structural information of the data structure (e.g., pointer values, structural checking information). The probability of an erroneous pointer to a random location remaining undetected by the techniques presented in this paper is proportional to 2^{-bd}, where b is the number of bits used to represent a pointer, and d is the number of erroneous pointers required for masking. Since this probability is very low, the error detection analysis concentrates on the case where erroneous pointers point to other nodes of the same type. This kind of error may occur in partially or incorrectly updated data structures, or as a result of software errors or hardware failures. These erroneous pointers may or may not coincide with logical pointer boundaries; however, the routine that accesses nodes from slow memory can detect these boundary errors and supply this information to the LCED procedure.

Memory subsystems are commonly configured hierarchically, and the ratio of the access time of slower memory (used to store the data structure, e.g., MOS RAM, disk) to that of faster memory (used to buffer the currently accessed nodes, e.g., cache, register file) is usually very large. Hence it is desirable to have all the nodes in the LCED or LCEC localities stored in the fastest memory. In the remainder of this paper, A_i will represent the address of a node N_i in a linked data structure. N_i may have many pointers to other nodes, and a desired move MV from N_i will be represented as $N_i \rightarrow N_{MV}$.

DEFINITION 1: R_c is a fast memory of capacity c nodes, which holds the c most recently accessed nodes, including the node reached by the current move MV. Since a move is performed between two nodes, c must be at least two to verify the move. That is, for a move MV $N_i \rightarrow N_{MV}$, R_c holds both N_i and N_{MV}. If $c = 1$ then only N_{MV} could be stored, and the information of the source node N_i (e.g., address, pointer value) would be lost. Thus, an erroneous move would be indistinguishable from a correct move. □

The LCED procedure requires a set of c nodes to verify the move MV. This set of nodes is called a *Checking Window*. The cost of a Checking Window is proportional to c, since it involves storing the required nodes in the fast memory (storage cost) and performing checks on those nodes

(computation cost). The nodes in the Checking Window need not be re-accessed from slow memory, since they are already stored in R_c.

DEFINITION 2: Let a set of Checking Windows of size c, W^c, be defined recursively as: $W^c = \{W_j^{c-1} \cup N_k\}$ where W_j^{c-1} is the j^{th} *Checking Window* of W^{c-1} $(1 \leq j \leq |W^{c-1}|)$ and $N_k \notin W_j^{c-1}$ is adjacent to one of the nodes in W_j^{c-1}. The base case is $W^2 = \{\{N_i, N_{MV}\}\}$. □

W_m^c, for some m, is constructed by adding one more node N_k to the smaller Checking Window W_j^{c-1}, such that N_k can be reached from W_j^{c-1} in one move. All such W_m^c form a set of sets, W^c. It will be shown that Checking Windows of the same size do not necessarily achieve the same detectability. When the context is clear, we may use W^c to represent one particular W_j^c.

EXAMPLE 1: Consider a forward move $N_i \rightarrow N_{i+1}$ in a normal double-linked list (Figure 1):

$$W_1^2 = \{N_i, N_{i+1}\}$$

$$W^2 = \{W_1^2\} = \{\{N_i, N_{i+1}\}\}$$

$$W_1^3 = \{N_i, N_{i+1}, N_{i+2}\}$$

$$W_2^3 = \{N_{i-1}, N_i, N_{i+1}\}$$

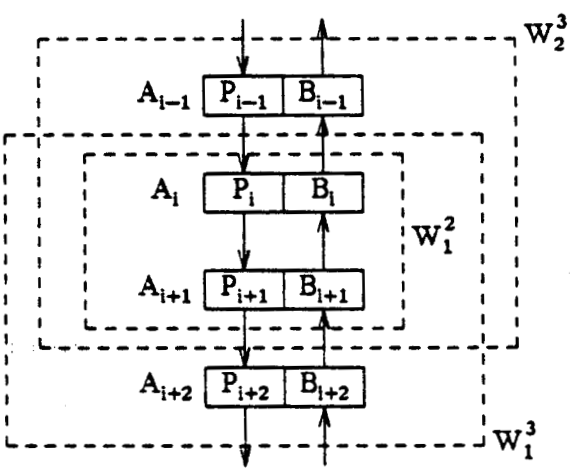

Figure 1. Checking Windows for a Double-Linked List.

$$W^3 = \{W_1^3, W_2^3\} = \{\{N_i, N_{i+1}, N_{i+2}\}, \{N_{i-1}, N_i, N_{i+1}\}\}$$

etc. □

The Lock and Key concept is now introduced as a generalization of structural checking information that is distributed throughout the nodes of linked data structures (distributed checks). In the simplest case, nodes in the structure will have associated with them a Key. When performing a move from a node to its child, the node's Key becomes an argument to the child's Lock function, which either returns "True," signaling a valid move, or "False," signaling error detection. In its most general form, the Lock and Key concept allows for multiple-Key Locks and Keys distributed over potentially many nodes.

DEFINITION 3: A *Key* is information associated with a node (e.g., its address, a pointer, or distributed check) that is used by a checking function to verify a move. □

DEFINITION 4: A *Lock*, $Lock_{MV}$, is a checking function that verifies a move, such that $Lock_{MV}(Key_1, \cdots, Key_k)$ = "True" if all its Key_i arguments are present and correct, "False" if all its Key_i arguments are present and not all are correct, or "X" (don't care) if not all its Key_i are present. A Lock whose Key arguments are all present is called a *checkable* Lock, otherwise the Lock is an *uncheckable* Lock. □

The computational overhead to evaluate the checkable Locks is O(1) if all $Lock_{MV}$ are defined on Keys that can be contained in a fixed-size Checking Window W_j^c. No storage overhead is necessary because Locks are functions and are not stored, and Keys can be information that is already present in the node, e.g., pointers.

DEFINITION 5: A *Circular Lock*, $CLock_{N_i \sim N_k}$, is a Lock function whose Keys are addresses of nodes:

$$Keys = \langle A_i, A_k \rangle$$

$$CLock_{N_i \sim N_k}(x, y) = (x \ ?= g(y))$$

where $\sim$ is a pointer (e.g., a forward pointer, a backward pointer, a virtual backpointer) of N_i to

N_k, g is a function that generates x using a series of pointers, and ?= represents a comparison that returns either "True" or "False" for a checkable CLock. □

Circular Locks possess the property that for all starting nodes N_i, any single pointer error encountered in the moves of g causes the Lock to evaluate to "False." The following two examples show that the double-linked list and a binary tree with signatured access paths employ Locks and Keys. The double-linked list uses a Circular Lock checking function, while the tree with signatured access paths uses a Lock defined on O(*height-of-tree*) Keys.

EXAMPLE 2: Let $N_0, N_1, \cdots, N_n$ be the nodes of a double-linked list. Let a node N_i have a forward pointer P_i and a backpointer B_i. For a forward move $N_i \rightarrow N_{i+1}$:

$$\text{Keys} = \langle A_i, A_{i+1} \rangle$$

$$\text{CLock}_{N_i \rightarrow N_{i+1}}(x, y) = (x \ ?= g(y)) = (x \ ?= y.B).$$

The backpointers are the distributed checks, and the g function in the Circular Lock retrieves the backpointer B from the node at y. This structure achieves O(1) single pointer error detection in Checking Window W_1^2 (*cf.* Example 1). □

EXAMPLE 3: In the signatured access path technique, signatures defined over the nodes of valid traversal paths are embedded at path termination points, where a traversal path starts at a header and ends at a leaf, for a binary tree [14]. Error detection is achieved by comparing signatures generated at traversal time with the embedded signatures. A simple signature is the logical exclusive-or function (⊕) of all the pointers in the valid traversal path.

$$\text{Keys} = \langle \text{ordered set of pointers in a valid traversal path, signature} \rangle$$

$$\text{Lock}_{\text{forward}}(p_1, \cdots, p_k, \text{signature}) = (p_1 \oplus \cdots \oplus p_k \oplus \text{signature} \ ?= 0).$$

The nodes' pointers are the distributed checks. This structure cannot guarantee O(1) detection time as O(*height-of-tree*) nodes may be accessed in the traversal path. □

We now determine the minimum number of errors that are required to cause the checkable Locks used by the LCED procedure to evaluate to "True" in a particular Checking Window. This is similar to the *changes* used by Taylor, Morgan and Black [15] to determine the distance between two data structure instances. The difference here is that the distance is measured within a Checking Window. Hence this new distance is termed *local distance*, from which the definition of local concurrent error detectability follows directly. Let $Lock_{MV}$ be defined, for every possible move MV in a specific data structure, over Keys distributed in nodes contained in a fixed-size Checking Window.

DEFINITION 6: The *local distance*, $d_j^c(MV)$, within a Checking Window of size c is defined as the minimum number of pointer errors in all W_j^c that can mask a move to an incorrect node, due to a pointer error, where MV is the move to the correct node. Errors are not detectable if all checkable $Lock_{MV}$ evaluate to "True." □

DEFINITION 7: The *local concurrent error detectability*, $D^c(MV)$, for a specified move MV and Checking Window of size c is given by:

$$D^c(MV) = \max(d_j^c(MV)) - 1,\ 1 \leqslant j \leqslant |W^c|.$$ □

The max function is used because, for a specified move, it is always possible to find a Checking Window W_j^c which can detect at least D^c simultaneous errors (including the pointer from N_i to N_{MV} that is erroneous). When the context is clear, we may omit the parameter MV in $d_j^c(MV)$ or $D^c(MV)$.

The following theorem will be used to prove that the local concurrent error detectability of data structures employing the virtual backpointer is the same for both forward and backward moves.

THEOREM 1: In a uniform data structure, if for every pointer of the form $N_i \rightarrow N_k$ there exists a $\sim$ pointer to reach N_i from N_k in one move, and the Lock functions are Circular Locks, then using an LCED procedure, $D^c(N_i \rightarrow N_k) = D^c(N_k \sim N_i) = D^c$.

PROOF: Since the data structure is uniform, $N_i \rightarrow N_k$ and $N_k \sim N_i$ represent all possible forward and backward moves, respectively. Notice that $W_1^2 = \{N_i, N_k\}$. Thus, all W_j^c are also the same for both moves as W_j^c is defined on W_1^2. If $N_i \rightarrow N_k$ is erroneously changed to $N_i \rightarrow N_{k'}$, it is isomorphic to the case $N_k \sim N_i$ being changed to $N_k \sim N_{i'}$, because the pointers used in the g function of the Circular Lock are not changed by the isomorphism. In both cases, the Locks evaluate to the same value because the accessible nodes in W_j^c are the same. By Definition 6, $d_j^c(N_i \rightarrow N_k) = d_j^c(N_k \sim N_i)$. Hence $D^c(N_i \rightarrow N_k) = D^c(N_k \sim N_i) = D^c$. □

Theorem 2 will be used in determining the upper bounds of local concurrent error detectability for the Virtual Double-Linked List and B-Tree with Virtual Backpointers.

THEOREM 2: Local concurrent error detectability is a monotonically increasing function of window size c. That is, $D^{c-1} \leqslant D^c \leqslant D^n$ for $3 \leqslant c \leqslant n$, where n is the total number of nodes in the data structure.

PROOF: Every W_m^c is constructed by adding one adjacent node N_k to a Checking Window of size c-1: $W_m^c = W_j^{c-1} \cup N_k$. If each checkable Lock in W_j^{c-1} evaluates to "True" in W_j^{c-1} then it will remain "True" in W_m^c because the Keys of the Lock are contained in both W_j^{c-1} and W_m^c. If the addition of N_k causes an uncheckable Lock in W_j^{c-1} to evaluate to "True" or "X" in W_m^c, this results in $d_m^c = d_j^{c-1}$. However, if the uncheckable Lock evaluates to "False," then $d_m^c > d_j^{c-1}$, since at least one other error would be required to mask the detected error. Hence, $d_m^c \geqslant d_j^{c-1}$. Then $\max(d_m^c) \geqslant \max(d_j^{c-1})$, and $D^c \geqslant D^{c-1}$ follows from Definition 7. The upper limit of detectability is trivially D^n, since the entire structure is then included in the Checking Window. □

If the Checking Window includes all the nodes of the structure, LCED procedure degenerates into a global error detection procedure, which requires $O(n)$ execution time. Therefore, to achieve maximum local concurrent error detectability, it is sufficient to use a W_j^c with minimum size c for which $D^c = D^n$.

The LCED procedures mentioned throughout this section were unspecified because the actual procedure used depends on the particular data structure to be checked. The general LCED

technique is as follows. First, determine the appropriate Checking Window W_j^c that achieves the desired local concurrent error detectability. For each possible move from each node, identify the Lock functions and associated Key arguments that are used to perform the checking. The LCED procedure can be constructed as follows: for each move made, access the nodes defined by the Checking Window, and evaluate all the checkable Lock functions. If all Locks return "True," then either no error has occurred or undetectable errors have occurred; if any Lock returns "False," then at least one error has been detected. Once an error has been detected by an LCED procedure, LCEC may be performed. The upper limit of correctability is $\left\lfloor \frac{D^c}{2} \right\rfloor$. However, the actual correctability depends upon the data structure.

Since errors are detected and corrected based only on information from nodes in the Checking Window, many other detectable errors may exist simultaneously throughout the data structure. Although the local concurrent error detectability and correctability may only be one or two in the window, the actual number of detectable and correctable errors may be much larger.

III. VIRTUAL BACKPOINTERS

The *virtual backpointer* is a distributed checking symbol that can be used to achieve O(1) LCED and O(1) LCEC during a forward move, and O(1) LCED and O(n) LCEC during a backward move in many linked data structures. In addition, it can be used to generate a backpointer from a node N_i to its parent N_{parent}. In the general case, a virtual backpointer may point to an ancestor $N_{ancestor}$ of a node N_i, where $N_{ancestor}$ is an *ancestor* of N_i if there exists a series of moves from $N_{ancestor}$ to N_i.

DEFINITION 8: In a linked data structure, let $N_{ancestor}$ be an ancestor of N_i, and Q_i be the set of all pointers in N_i. The *virtual backpointer* $V_i = f(Q_i, A_{ancestor})$, where f is a function such that $A_{ancestor} = f^*(Q_i, V_i) = f^*(Q_i, f(Q_i, A_{ancestor}))$, and f^* is a companion function determined by f. In

general, there may be vectors of virtual backpointers, $\vec{V}_i = \vec{f}(Q_i, \vec{A})$, which, after suitable transformation by $\vec{f}^*$, point to vectors of nodes $\vec{A}$. □

The virtual backpointer has the following properties. 1) For a forward move $N_i \rightarrow N_{i+1}$, V_{i+1} provides checking information. 2) For a backward move $N_{i+1} \sim N_i$, V_{i+1} provides the backpointer after transformation by f^*, and $Q_{ancestor}$ is used as checking information. Two example data structures employing the virtual backpointer are presented in the following subsections: the Virtual Double-Linked List, which is derived from the double-linked list, and the B-Tree with Virtual Backpointers, which is derived from the B-tree.

A. Virtual Double-Linked List

The *Virtual Double-Linked List* (VDLL) is a data structure that employs the virtual backpointer and possesses local concurrent error detectability and correctability. Errors are detected in O(1) time with an LCED procedure. For a forward move, detected errors may be corrected using LCEC in O(1) time; for a backward move, detected errors may be corrected using LCEC in $O(n)$ time. The VDLL requires no more storage space than the double-linked list (DLL), and retains the simplicity of the DLL, in that it is possible to move directly from a node to its parent, using the virtual backpointer. This is not possible, for example, in the modified(k) DLL [1], for $k \geq 2$, which must access other ancestors of a node in order to reach the node's parent.

DEFINITION 9: A *Virtual Double-Linked List* is described as follows (Figure 2). In a linked list data structure, let N_{i-1} be the parent of N_i, and P_i be the forward pointer of the N_i, therefore $Q_i = \{P_i\}$. Let $f(\{x\}, y) = f^*(\{x\}, y) = x \oplus y$, then $V_i = P_i \oplus A_{i-1} = A_{i+1} \oplus A_{i-1}$, and $A_{i-1} = P_i \oplus V_i$, where $\oplus$ denotes the logical exclusive-or function. Also, c header nodes $N_0, N_{-1}, \cdots, N_{-c+1}$ are added, where c is the size of the Checking Window. These header nodes are assumed to be always accessible by the LCED procedure. Note that $N_{-c+1} = N_\infty$. □

The VDLL is created from the DLL by replacing its backpointers with virtual backpointers. The same operation can be applied to the modified(k) DLL family [1], resulting in the modified(k)

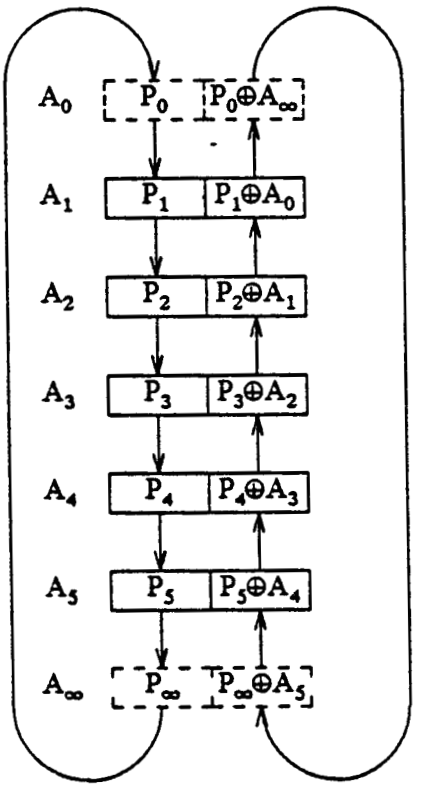

Figure 2. Virtual Double-Linked List (VDLL) of 5 nodes.

VDLL structures. It will be shown that each modified(k) VDLL achieves greater local concurrent error detectability than the corresponding modified(k) DLL.

DEFINITION 10: A *modified(k) Virtual Double-Linked List* is described as follows. In a linked list data structure, let N_{i-k} be the k^{th} ancestor of N_i, and P_i be the forward pointer of the N_i, therefore $Q_i = \{P_i\}$. Let $f(x, y) = f^*(\{x\}, y) = x \oplus y$, then $V_i = P_i \oplus A_{i-k} = A_{i+1} \oplus A_{i-k}$, and $A_{i-k} = P_i \oplus V_i$. Also, max(k+1, c) header nodes, are added. □

The possible Locks and Keys of the VDLL can be identified as follows (Figure 2). For a forward move $N_i \rightarrow N_{i+1}$ following P_i,

$$\text{Keys} = \langle A_i, P_{i+1} \oplus V_{i+1} \rangle$$

$$\text{CLock}_{N_i \rightarrow N_{i+1}}(x, y) = (x \stackrel{?}{=} g(y)) = (x \stackrel{?}{=} y).$$

where g is the identity function. For the backward move $N_{i+1} \sim N_i$ following $V_{i+1} \oplus P_{i+1}$.

$$\text{Keys} = \langle A_{i+1}, A_i \rangle$$

$$\text{CLock}_{N_{i+1} \sim N_i}(x, y) = (x \ ?= g(y)) = (x \ ?= y.P),$$

where g retrieves the pointer P from the node at y. Locks and Keys for the modified(k) VDLL can be identified similarly. Using the results of the analysis of LCED, we now determine the local concurrent error detectability of the VDLL.

THEOREM 3: Using an LCED procedure, the local concurrent error detectability of the VDLL is $D^2(\text{forward}) = D^2(\text{backward}) = D^2 = 1$, and $D^c(\text{forward}) = D^c(\text{backward}) = D^c = D^3 = 2$, $\forall\ c \geq 3$.

PROOF: Since the VDLL uses virtual backpointers and Circular Locks, by Theorem 1, $D^c(\text{forward}) = D^c(\text{backward})$. Consider a forward move MV, $N_i \rightarrow N_{i+1}$, following P_i. The LCED procedure attempts to verify this move. A pointer that does not point to a logical node boundary can easily be detected by the node access routine. Therefore consider only erroneous pointers that lead to valid logical node addresses. Suppose that P_i is erroneous and leads to N_{j+1} instead of N_{i+1}. In $W_1^2 = \{N_i, N_{j+1}\}$, $d_1^2 = 2$: either V_{j+1} or P_{j+1} must be erroneous to mask the error in P_i. Assume that V_{j+1} is erroneous (Figure 3a). In $W_1^3 = \{N_i, N_{j+1}, N_{j+2}\}$, $d_1^3 = 2$. However, in $W_2^3 = \{N_{i-1}, N_i, N_{j+1}\}$, V_i will lead to the detection of the error in P_i, because following the backpointer given by $V_i \oplus P_i$ will lead to a node N_{k-1} instead of N_{i-1}, and $P_{k-1} \neq N_i$. Therefore, V_i must be changed into the value $A_{j+1} \oplus A_{i-1}$ to mask the error in P_i. Thus $d_2^3 = 3$.

Assume now that V_{j+1} is not erroneous, so P_{j+1} must be erroneous (Figure 3b). Consider $W_1^3 = \{N_i, N_{j+1}, N_{k+2}\}$. The LCED procedure will not detect the error in P_i if P_{j+1} has been changed to $A_{k+2} = A_i \oplus V_{j+1}$, and $V_{k+2} \oplus P_{k+2}$ has been changed (via a change in either V_{k+2} or P_{k+2}) to A_{j+1}. The remainder of the analysis is similar to the case above, and gives $d_1^2 = 2$, $d_1^3 = 3$, and $d_2^3 = 3$. According to Definition 7, $D^2 = 1$ and $D^3 = 2$. Since the VDLL can be changed to another correct VDLL by three pointer errors (node deletion), $D^n = 2$, where n is the number of nodes in the struc-

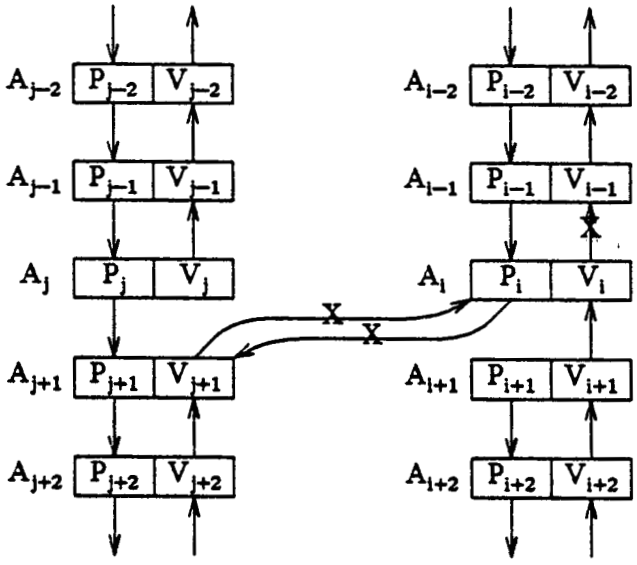

Figure 3a. Analysis of VDLL: Errors in P_i, V_i, and V_{j+1}.

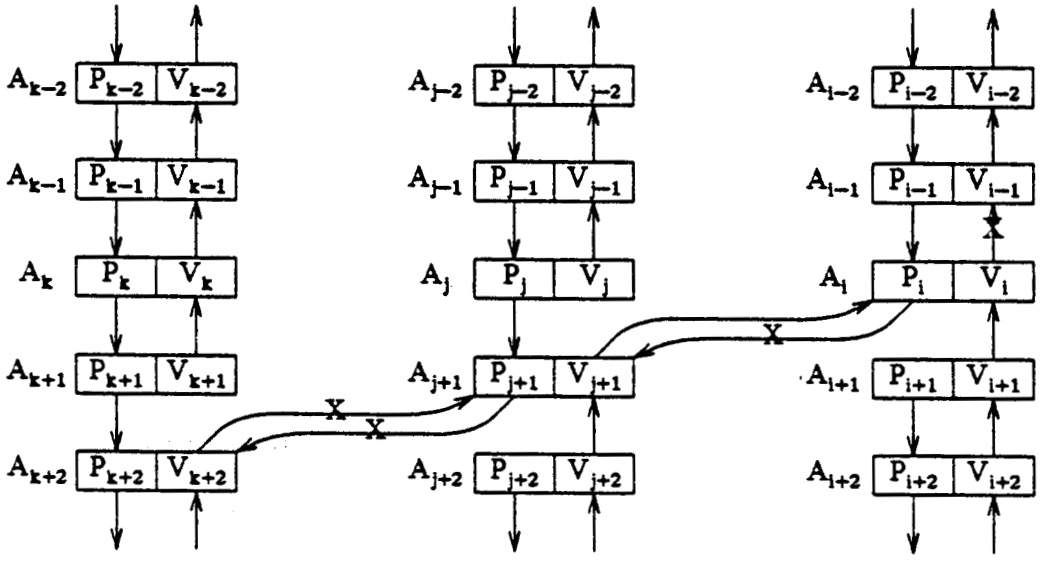

Figure 3b. Analysis of VDLL: Errors in P_i, V_i, P_{j+1}, and V_{k+2}.

ture. By Theorem 2, $D^c = 2, \forall\, c \geqslant 3$. □

The above proof suggests that when moving forward $N_i \rightarrow N_{MV}$ following P_i, use $W^3 = \{N_{prev}, N_i, N_{MV}\}$ as the Checking Window, where N_{prev} corresponds to N_{i-1} in the proof; and when moving backward $N_i \sim N_{MV}$ following $P_i \oplus V_i$, use $W^3 = \{N_i, N_{MV}, N_{next}\}$ as the Checking Window, where N_{next} is the node reached by following $P_{MV} \oplus V_{MV}$. By using these windows, double pointer errors can be detected, or single pointer errors corrected (described below). The LCED procedure using this Checking Window evaluates four locks when moving either forward or backward. For a forward move, the locks are: L1: $A_{prev} ?= P_i \oplus V_i$, L2: $A_i ?= P_{MV} \oplus V_{MV}$, L3: $A_i ?= P_{prev}$ and L4: $A_{MV} ?= P_i$. For a backward move, the locks are: L1: $A_{next} ?= P_{MV} \oplus V_{MV}$, L2: $A_{MV} ?= P_i \oplus V_i$, L3: $A_{MV} ?= P_{next}$ and L4: $A_i ?= P_{MV}$. (In the W^2 Checking Window, only two locks are evaluated, namely $A_i ?= P_{MV} \oplus V_{MV}$ and $A_{MV} ?= P_i$ for the forward move, and $A_{MV} ?= P_i \oplus V_i$ and $A_i ?= P_{MV}$ for the backward move.) A comparison of local concurrent error detectability is given in Table 1 for the VDLL, modified(2) VDLL, modified(3) VDLL, DLL without a global count, and modified(2) and modified(3) DLL without global counts [1], for various sized Checking Windows. The local detectability of the modified(2) and modified(3) VDLL can be obtained using

Table 1. Local Concurrent Error Detectability
of Several Linked List Data Structures.

Local Detectability	VDLL	mod(2) VDLL	mod(3) VDLL	DLL	mod(2) DLL	mod(3) DLL
D^2	1	0	0	1	0	0
D^3	2	1	0	1	1	0
D^4	2	2	1	1	2	1
D^5	2	3	2	1	2	2
D^6	2	3	3	1	2	3
D^7	2	3	4	1	2	3

the same analysis technique as that applied to VDLL. Any modified(k) VDLL achieves greater local concurrent error detectability than the corresponding modified(k) DLL. For $k > 3$, no further improvement in detectability can be made for either of the two families.

THEOREM 4: Any single pointer error detected by a forward move in $W^3 = \{N_{prev}, N_i, N_{MV}\}$ in a VDLL can be corrected with an O(1) LCEC procedure requiring at most one extra node access for both diagnosis and correction. Any single pointer error detected by a backward move in $W^3 = \{N_{next}, N_{MV}, N_i\}$ in a VDLL can be corrected with an O(n) LCEC procedure requiring at most one extra node access for diagnosis.

PROOF: Since the local concurrent error detectability for this structure using W^3 is $D^3 = 2$, the upper limit of correctability is 1. Assume that a single error has been detected during a forward move. The LCED procedure supplies the values of the four detection locks (Table 2a), and three error indication values generated by a node access routine, NA_{prev}, NA_i, NA_{MV}, that indicate out-of-bounds pointers or pointers that do not point to logical node boundaries, when used to access N_{prev}, N_i and N_{MV}, respectively. There are eight possible errors: 1) A_{prev} error, 2) P_{prev} error, 3) A_i error, 4) P_i error, 5) V_i error, 6) A_{MV} error, 7) P_{MV} error and 8) V_{MV} error. To distinguish the eight errors, the seven-tuple syndrome {L1, L2, L3, L4, NA_{prev}, NA_i, NA_{MV}} is constructed (Table 2b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case one extra node is accessed to completely diagnose the error. N_X is accessed by following P_{MV} to distinguish a P_{MV} error from a V_{MV} error. N_Y is accessed by following $P_i \oplus V_i$ to distinguish an A_{prev} error from a V_i error. Once the error has been diagnosed, correction proceeds as follows:

1) A_{prev} error: correct value is $P_i \oplus V_i$.

2) P_{prev} error: correct value is A_i.

3) A_i error: correct value is P_{prev}.

4) P_i error: correct value is A_{MV}.

Table 2a. Detection and Diagnosis Locks for Forward Moves in the VDLL using W^3.

Detection Locks		
L1	A_{prev} ?= $P_i \oplus V_i$	
L2	A_i ?= $P_{MV} \oplus V_{MV}$	
L3	A_i ?= P_{prev}	
L4	A_{MV} ?= P_i	
Diagnosis Locks		
L5	A_{MV} ?= $P_X \oplus V_X$	Access N_X via P_{MV}
L6	A_i ?= P_Y	Access N_Y via $P_i \oplus V_i$

Table 2b. Error Detection and Diagnosis Syndromes for Errors Detected by Forward Moves in the VDLL using W^3.

error	L1	L2	L3	L4	NA_{prev}	NA_i	NA_{MV}	L5	L6
A_{prev}	F	T	T	T	T	T	T	-	T
P_{prev}	T	T	F	T	T	T	T	-	-
A_i	T	F	F	T	T	T	T	-	-
P_i	F	T	T	T	T	T	F	-	-
	F	F	T	T	T	T	T	-	-
	F	F	T	T	T	T	F	-	-
V_i	F	T	T	T	T	T	T	-	F
A_{MV}	T	T	T	F	T	T	T	-	-
P_{MV}	T	F	T	T	T	T	T	F	-
V_{MV}	T	F	T	T	T	T	T	T	-

5) V_i error: correct value is $A_{prev} \oplus P_i$.

6) A_{MV} error: correct value is P_i.

7) P_{MV} error: correct value is $A_i \oplus V_{MV}$.

8) V_{MV} error: correct value is $A_i \oplus P_{MV}$.

Assume now that a single error has been detected during a backward move. The LCED procedure supplies the values of the four detection locks (Table 3a), and three error indication values

generated by a node access routine, NA_{next}, NA_{MV}, NA_i, that indicate out-of-bounds pointers or pointers that do not point to logical node boundaries, when used to access N_{next}, N_{MV} and N_i, respectively. There are eight possible errors: 1) A_{next} error, 2) P_{next} error, 3) A_{MV} error, 4) P_{MV} error, 5) V_{MV} error, 6) A_i error, 7) P_i error and 8) V_i error. To distinguish the eight errors, the seven-tuple syndrome {L1, L2, L3, L4, NA_{next}, NA_{MV}, NA_i} is constructed (Table 3b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case one extra node is accessed to

Table 3a. Detection and Diagnosis Locks for Backward Moves in the VDLL using W^3.

Detection Locks		
L1	A_{next} ?= $P_{MV} \oplus V_{MV}$	
L2	A_{MV} ?= $P_i \oplus V_i$	
L3	A_{MV} ?= P_{next}	
L4	A_i ?= P_{MV}	
Diagnosis Locks		
L5	A_{next} ?= $P_X \oplus V_X$	Access N_X via P_{next}
L6	A_i ?= $P_Y \oplus V_Y$	Access N_Y via P_i

Table 3b. Error Detection and Diagnosis Syndromes for Errors Detected by Backward Moves in the VDLL using W^3.

error	L1	L2	L3	L4	NA_{next}	NA_{MV}	NA_i	L5	L6
A_{next}	F	T	T	T	T	T	T	-	-
P_{next}	T	T	F	T	T	T	T	F	-
A_{MV}	T	F	F	T	T	T	T	-	-
P_{MV}	T	T	T	F	F	T	T	-	-
	T	T	F	F	T	T	T	-	-
	T	T	F	F	F	T	T	-	-
V_{MV}	T	T	T	T	F	T	T	T	-
	T	T	F	T	T	T	T	-	-
	T	T	F	T	F	T	T	-	-
A_i	T	T	T	F	T	T	T	-	-
P_i	T	F	T	T	T	T	T	-	F
V_i	T	F	T	T	T	T	T	-	T

completely diagnose the error. N_X is accessed by following P_{next} to distinguish a P_{next} error from a V_{MV} error. N_Y is accessed by following P_i to distinguish a P_i error from a V_i error. Once the error has been diagnosed, correction proceeds as follows:

1) A_{next} error: correct value is $P_{MV} \oplus V_{MV}$.

2) P_{next} error: correct value is A_{MV}.

3) A_{MV} error: correct value is P_{next}.

4) P_{MV} error: correct value is A_i.

5) V_{MV} error: To correct the error in V_{MV}, first access the headers of the structure. Next, move forward, accessing nodes $N_0, N_1, \cdots, N_k$, performing W^3 LCED and correcting single errors with O(1) LCEC, until $P_k = A_{MV}$. Then the correct value of $V_{MV} = A_k \oplus P_{MV}$.

6) A_i error: correct value is P_{MV}.

7) P_i error: correct value is $A_{MV} \oplus V_i$.

8) V_i error: correct value is $A_{MV} \oplus P_i$. □

Note that for a forward move, both diagnosis and correction are O(1) time, and require one extra node access. For a backward move, diagnosis is O(1) time (one extra node access) but correction requires $O(n)$ extra node accesses in the worst case. Thus, O(1) LCEC is possible for an error detected by a forward move, while $O(n)$ LCEC is possible for an error detected by a backward move. The proof assumed that W^3 LCED was used; if W^2 is used instead, then diagnosis for both the forward and backward moves is still O(1), but correction for both moves requires $O(n)$ LCEC.

B. B-Tree with Virtual Backpointers

The *B-Tree with Virtual Backpointers* (VBT) of order m is a data structure that possesses local concurrent error detectability and correctability. Errors are detected in O(1) time if the time complexity is measured as a function of the number of nodes in the tree, i.e., n. For a forward move,

detected errors can be corrected using O(1) LCEC; for a backward move, detected errors can be corrected using $O(\log_{2m} n)$ LCEC. The VBT requires m+4 extra fields in each node, and has the additional feature that backward traversal can be performed without a stack, using the virtual backpointer.

The underlying structure of the VBT is the B-tree of order m [16], which finds application in the construction and maintenance of large-scale search trees. The B-tree has the following characteristics:

1) Every node contains at most $2m$ keys, and every node except the root contains at least m keys. The root contains at least one key.

2) Every node is either a leaf node, with no pointers to other nodes, or an internal node, with pointers to other internal nodes or to leaf nodes.

3) All leaf nodes appear at the same level.

4) An internal node with k keys will have k+1 pointers to subtrees. The k keys will be arranged in strictly increasing order, and keys in the i^{th} subtree will be less than the i^{th} key, while keys in the $i+1^{th}$ subtree will be greater than the i^{th} key.

Let $P_{i,j}$ be the j^{th} pointer in node N_i. Assume that each pointer requires one word of memory. Therefore, each pointer is uniquely addressable by $A_{i,j}$ (Figure 4a). The VBT is modified from the B-tree in the following ways to achieve local concurrent error detectability.

1) A header node N_0 is created with $P_{0,j} = A_{1,j}$ for $0 \leqslant j \leqslant 2m$.

2) V_i, the virtual backpointer of N_i, is defined as $V_i = P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus A_{parent,j}$ where the j^{th} pointer in N_{parent} points to N_i. For the special case of the virtual backpointer from the root to the header, V_1 is defined on $A_{0,0}$, even though all $P_{0,j}$ point to N_1.

3) The keys of N_i (i.e., $K_{i,1}, K_{i,2}, \cdots, K_{i,2m}$) are arranged in a matrix (Figure 4b) and the *key check symbols* $X_{i,j}$ and $Y_{i,j}$ are generated using a product code [17] as follows:

$$X_{i,j} = K_{i,(j-1)m+1} \oplus K_{i,(j-1)m+2} \oplus \cdots \oplus K_{i,(j-1)m+m}, \ 1 \leqslant j \leqslant 2$$

$$Y_{i,j} = K_{i,j} \oplus K_{i,m+j}, \ 1 \leqslant j \leqslant m.$$

$K_{i,j}$ is used to determine $X_{i,\mathrm{int}((j-1)/m)+1}$ and $Y_{i,(j-1) \bmod m + 1}$, called its *corresponding* X and Y check symbols, respectively.

The number of key fields used in N_i is called $count_i$, which is added for performance enhancement. A VBT of order 2 is illustrated in Figure 4c. The possible Locks and Keys of the VBT can be identified as follows. Assuming the j^{th} pointer of N_i points to N_k, for a forward move $N_i \rightarrow N_k$ following $P_{i,j}$,

$$\text{Keys} = \langle A_{i,j}, (P_{k,0} \oplus P_{k,1} \oplus \cdots \oplus P_{k,2m} \oplus V_k) \rangle$$

$$\text{CLock}_{N_i \rightarrow N_k}(x, y) = (x \ ?= g(y)) = (x \ ?= y),$$

where g is the identity function. For the backward move $N_k \sim N_i$ following $(P_{k,0} \oplus P_{k,1} \oplus \cdots \oplus P_{k,2m} \oplus V_k)$,

$$\text{Keys} = \langle A_k, A_{i,j} \rangle$$

$$\text{CLock}_{N_k \sim N_i}(x, y) = (x \ ?= g(y)) = (x \ ?= y.P_j),$$

where g retrieves the j^{th} pointer $P_{i,j}$ from the node at y.

We now determine the local concurrent error detectability of the VBT, employing the results of the analysis of LCED. Using Theorem 2, Table 4 presents the possible key and pointer errors that can occur in the VBT (errors in the *count* field are covered by the fifth and sixth rows of the table), and the number of errors required to mask them, assuming an LCED procedure is used.

THEOREM 5: Using an LCED procedure, the local concurrent error detectability of the VBT is $D^2 = 1$ and $D^3 = D^c = 2, \forall c \geqslant 3$.

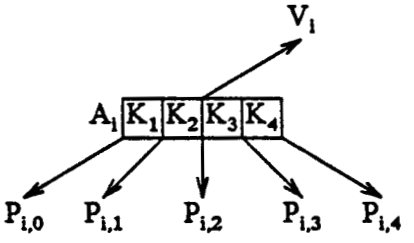

Figure 4a. Node Representation in Order-2 B-Tree with Virtual Backpointers.

$K_{i,1}$	$K_{i,2}$	...	$K_{i,m}$	$X_{i,1}$
$K_{i,m}+1$	$K_{i,m}+2$	...	$K_{i,2m}$	$X_{i,2}$
$Y_{i,1}$	$Y_{i,2}$	...	$Y_{i,m}$	

$V_i = P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus A_{parent,j}$
$count_i$ = number of key fields used in N_i

Figure 4b. Virtual Backpointer and Key Check Symbols in a VBT Node.

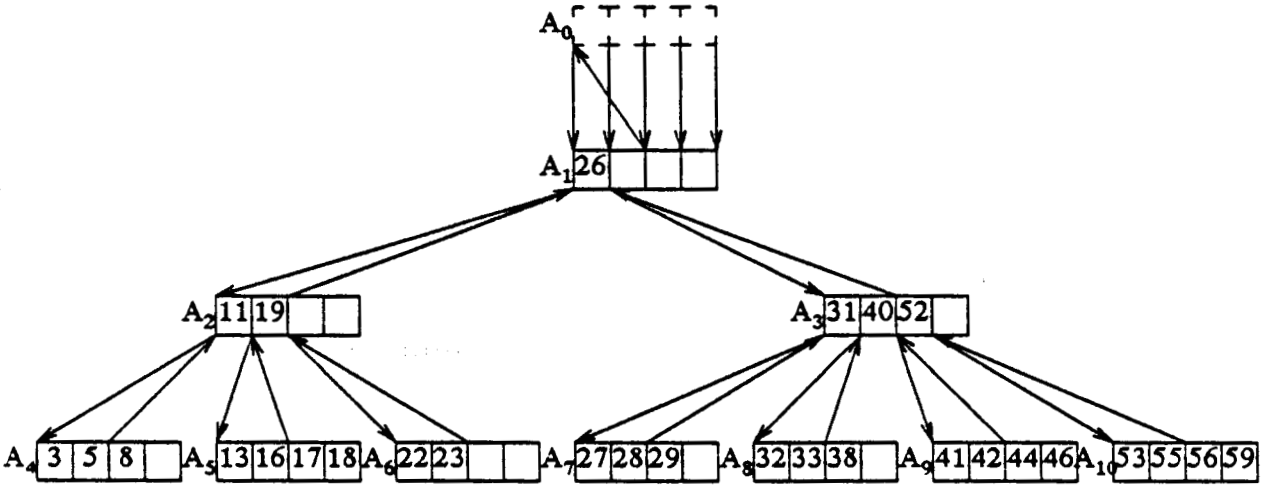

Figure 4c. Order-2 B-Tree with Virtual Backpointers (VBT).

PROOF: From Table 4, the minimum $d_j^2 = 2$ and the minimum $d_j^c = 3$, $\forall\ c \geqslant 3$. From Definition 7, it follows that $D^2 = 1$ and $D^c = 2$, $\forall\ c \geqslant 3$. □

From Table 4 it can be seen that no increase in the local concurrent error detectability can be gained by using W^c for $c \geqslant 3$. It can be shown that when moving forward $N_i \rightarrow N_{MV}$ following $P_{i,j}$, or when moving backward $N_i \sim N_{MV}$ following $(P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV})$, use $W^3 = \{N_{prev}, N_i, N_{MV}\}$ and $W^3 = \{N_i, N_{MV}, N_{next}\}$ respectively, to achieve detection of double pointer errors, or correction of single pointer errors (described below). In the window for the forward move, N_{prev} is the parent of N_i, and in that for the backward move, N_{next} is the parent of N_{MV}. The LCED procedure using this window evaluates four locks. For a forward move, the locks are: L1: $A_{prev,r} ?= P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$, L2: $A_{i,j} ?= P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$, L3: $A_i ?= P_{prev,r}$ and L4: $A_{MV} ?= P_{i,j}$. For a backward move, the locks are: L1: $A_{next,s} ?= P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$, L2: $A_{MV,t} ?= P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$, L3: $A_{MV} ?= P_{next,s}$ and L4: $A_i ?= P_{MV,t}$. (In the W^2 Checking Window, only two locks are evaluated, namely $A_{i,j} ?= P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$ and $A_{MV} ?= P_{i,j}$ for the forward move, and $A_{MV,t} ?= P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$ and $A_i ?= P_{MV,t}$ for the backward move).

Table 4. Analysis of Errors in the VBT.

Error Condition	$\max(d_i^2)$	$\max(d_i^3)$	$\max(d_j^c)$ $\forall\ c \geqslant 4$
Non-empty VBT becomes empty	$2m+1$	$2m+1$	$2m+1$
Empty VBT becomes non-empty	$2m+2$	$2m+2$	$2m+2$
Key, X or Y becomes erroneous	3	3	3
Internal node's non-null pointer points to incorrect node	2	3	3
Internal node's non-null pointer becomes null	6	6	6
Internal node's null pointer becomes non-null	6	7	7
Two of internal node's pointers exchanged	2	4	4
Internal node becomes a leaf node	3	3	3
Leaf node becomes an internal node	3	4	5

THEOREM 6: Any single pointer error detected by a forward move in $W^3 = \{N_{prev}, N_i, N_{MV}\}$ can be corrected with at most $2m+1$ extra node accesses in O(1) time. Any single pointer error detected by a backward move in $W^3 = \{N_i, N_{MV}, N_{next}\}$ can be corrected in $O(\log_{2m} n)$ time if it is detected during a backward move.

PROOF: Since the local concurrent error detectability of this structure in W^3 is $D^3 = 2$, the upper limit of correctability is 1. Assume that the error detected is a single error. The error may be a key, a key check symbol, a *count* or a pointer. For the key or key check symbol error, diagnosis and correction are performed using the procedures for product codes [17]. For a *count* error, all the keys and key check symbols will be correct, hence counting the non-null keys will regenerate the *count*.

For the pointer error, if the erroneous pointer is located at the header node, it can be corrected by simple comparison because there are $2m+1 \geqslant 3$ identical pointers in the header. Otherwise, there are two cases: detection by a forward move and detection by a backward move. Assume that the error has been detected during the forward move from N_i to N_{MV} following $P_{i,j}$. The LCED procedure supplies the values of the four detection locks (Table 5a), and three error indication values generated by a node access routine, NA_{prev}, NA_i, NA_{MV}, that indicate out-of-bounds pointers or pointers that do not point to logical pointer boundaries, when used to access N_{prev}, N_i and N_{MV}, respectively. There are nine possible errors: 1) A_{prev} error, 2) $P_{prev,r}$ error where $P_{prev,r}$ is the pointer from N_{prev} to N_i, 3) A_i error, 4) $P_{i,j}$ error, 5) $P_{i,s}$ error for $0 \leqslant s \leqslant 2m$ and $s \neq j$, 6) V_i error, 7) A_{MV} error, 8) $P_{MV,t}$ error for $0 \leqslant t \leqslant 2m$, and 9) V_{MV} error. To distinguish the nine errors, the seven-tuple syndrome {L1, L2, L3, L4, NA_{prev}, NA_i, NA_{MV}} is constructed (Table 5b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case extra nodes are accessed to completely diagnose the error. The nodes N_X^t are accessed by following all the pointers $P_{MV,t}$ from N_{MV} to distinguish a $P_{MV,t}$ error from a V_{MV} error. N_Y is accessed by following $P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$ to distinguish an A_{prev} error from a V_i error or a $P_{i,s}$ error. The latter two errors are distinguished

Table 5a. Detection and Diagnosis Locks for Forward Moves in the VBT using W^3.

Detection Locks		
L1	$A_{prev,r} ?= P_{i,0} \oplus \cdots \oplus P_{i,2m} \oplus V_i$	
L2	$A_{i,j} ?= P_{MV,0} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$	
L3	$A_i ?= P_{prev,r}$	
L4	$A_{MV} ?= P_{i,j}$	
Diagnosis Locks		
L5	$\prod_{t=0}^{count_{MV}} (A_{MV,t} ?= P^t_{X,0} \oplus \cdots \oplus P^t_{X,2m} \oplus V^t_X)$	Access N^t_X via $P_{MV,t}$ for $0 \leqslant t \leqslant 2m$
L6	$A_i ?= P_{Y,r}$	Access N_Y via $P_{i,0} \oplus \cdots \oplus P_{i,2m} \oplus V_i$
L7	$\prod_{s=0}^{count_i} (A_{i,j} ?= P^s_{Z,0} \oplus \cdots \oplus P^s_{Z,2m} \oplus V^s_Z)$	Access N^s_Z via $P_{i,s}$ for $0 \leqslant s \leqslant 2m$ and $s \neq j$

Table 5b. Error Detection and Diagnosis Syndromes for Errors Detected by Forward Moves in the VBT using W^3.

error	L1	L2	L3	L4	NA_{prev}	NA_i	NA_{MV}	L5	L6	L7
A_{prev}	F	T	T	T	T	T	T	-	T	-
$P_{prev,r}$	T	T	F	T	T	T	T	-	-	-
A_i	T	F	F	T	T	T	T	-	-	-
$P_{i,j}$	F	T	T	T	T	T	F	-	-	-
	F	F	T	T	T	T	T	-	-	-
	F	F	T	T	T	T	F	-	-	-
$P_{i,s}$ $(s \neq j)$	F	T	T	T	T	T	T	-	F	F
V_i	F	T	T	T	T	T	T	-	F	T
A_{MV}	T	T	T	F	T	T	T	-	-	-
$P_{MV,t}$	T	F	T	T	T	T	T	F	-	-
V_{MV}	T	F	T	T	T	T	T	T	-	-

by accessing the nodes N^s_Z by following all the pointers $P_{i,s}$ from N_i. Once the error has been diagnosed, correction proceeds as follows:

1) A_{prev} error: compute $A_{prev,r}$ from $P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$, from which A_{prev} can be calculated.

2) $P_{prev,r}$ error: correct value is A_i.

3) A_i error: correct value is $P_{prev,r}$.

4) $P_{i,j}$ error: correct value is A_{MV}.

5) $P_{i,s}$ error: correct value is $A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,s-1} \oplus P_{i,s+1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$.

6) V_i error: correct value is $A_{prev,r} \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$.

7) A_{MV} error: correct value is $P_{i,j}$.

8) $P_{MV,t}$ error: correct value is $A_{i,j} \oplus P_{MV,0} \oplus \cdots \oplus P_{MV,t-1} \oplus P_{MV,t+1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$.

9) V_{MV} error: correct value is $A_{i,j} \oplus P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m}$.

Assume now that the error has been detected during a backward move from N_i to N_{MV} following $P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$. The LCED procedure supplies the values of the four detection locks (Table 6a), and three error indication values generated by a node access routine, NA_{next}, NA_{MV}, NA_i, that indicate out-of-bounds pointers or pointers that do not point to logical pointer boundaries, when used to access N_{next}, N_{MV} and N_i, respectively. There are eight possible errors: 1) A_{next} error, 2) $P_{next,s}$ error where $P_{next,s}$ is the pointer from N_{next} to N_{MV}, 3) A_{MV} error, 4) $P_{MV,t}$ error for $0 \leq t \leq 2m$, 5) V_{MV} error, 6) A_i error, 7) $P_{i,j}$ error for $0 \leq j \leq 2m$, and 8) V_i error. To distinguish the eight errors, the seven-tuple syndrome {L1, L2, L3, L4, NA_{next}, NA_{MV}, NA_i} is constructed (Table 6b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case extra nodes are accessed to completely diagnose the error. The nodes N_X^j are accessed by following all the pointers $P_{i,j}$ from N_i to distinguish a $P_{i,j}$ error from a V_i error. N_Y is accessed by following $P_{next,0} \oplus P_{next,1} \oplus \cdots \oplus P_{next,2m} \oplus V_{next}$ to distinguish a $P_{next,s}$ error from a V_{MV} error or a $P_{MV,t}$ error. The latter two errors are distinguished by accessing the nodes N_Z^t by following all the pointers $P_{MV,t}$ from N_{MV}. Once the error has been diagnosed, correction proceeds as follows:

1) A_{next} error: compute $A_{next,s}$ from $P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$, from which A_{next} can be calculated.

2) $P_{next,s}$ error: correct value is A_{MV}.

Table 6a. Detection and Diagnosis Locks for Backward Moves in the VBT using W^3.

Detection Locks		
L1	$A_{next,s} ?= P_{MV,0} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$	
L2	$A_{MV,t} ?= P_{i,0} \oplus \cdots \oplus P_{i,2m} \oplus V_i$	
L3	$A_{MV} ?= P_{next,s}$	
L4	$A_i ?= P_{MV,t}$	
Diagnosis Locks		
L5	$\prod_{j=0}^{count_i} (A_{i,j} ?= P^j_{X,0} \oplus \cdots \oplus P^j_{X,2m} \oplus V^j_X)$	Access N^j_X via $P_{i,j}$ for $0 \leqslant j \leqslant 2m$
L6	$A_{next} ?= P_{Y,r}$	Access N_Y via $P_{next,0} \oplus \cdots \oplus P_{next,2m} \oplus V_{next} = A_{Y,r}$
L7	$\prod_{t=0}^{count_{MV}} (A_{MV,t} ?= P^t_{Z,0} \oplus \cdots \oplus P^t_{Z,2m} \oplus V^t_Z)$	Access N^t_Z via $P_{MV,t}$ for $0 \leqslant t \leqslant 2m$

Table 6b. Error Detection and Diagnosis Syndromes for Errors Detected by Backward Moves in the VBT using W^3.

error	L1	L2	L3	L4	NA_{next}	NA_{MV}	NA_i	L5	L6	L7
A_{next}	F	T	T	T	T	T	T	-	-	-
$P_{next,s}$	T	T	F	T	T	T	T	-	F	-
A_{MV}	T	F	F	T	T	T	T	-	-	-
$P_{MV,t}$	T	T	F	T	T	T	T	-	T	F
	T	T	F	T	F	T	T	-	T	F
V_{MV}	T	T	F	T	T	T	T	-	T	T
	T	T	F	T	F	T	T	-	T	T
A_i	T	T	T	F	T	T	T	-	-	-
$P_{i,j}$	T	F	T	T	T	T	T	F	-	-
V_i	T	F	T	T	T	T	T	T	-	-

3) A_{MV} error: correct value is $P_{next,s}$.

4) $P_{MV,t}$ error: To correct the error in $P_{MV,t}$, first access the headers of the structure. Next, move forward, accessing nodes $N_0, N_1, \cdots, N_k$, performing W^3 LCED and correcting single errors with O(1) LCEC, until $P_{k,s} = A_{MV}$. Then the correct value of $P_{MV,t}$ is $A_{k,s} \oplus P_{MV,0} \oplus \cdots \oplus P_{MV,t-1} \oplus P_{MV,t+1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$.

5) V_{MV} error: To correct the error in V_{MV}, first access the headers of the structure. Next, move forward, accessing nodes $N_0, N_1, \cdots, N_k$, performing W^3 LCED and correcting single errors with O(1) LCEC, until $P_{k,s} = A_{MV}$. Then the correct value of V_{MV} is $A_{k,s} \oplus P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m}$.

6) A_i error: correct value is $P_{MV,t}$.

7) $P_{i,j}$ error: correct value is $A_{MV,t} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$.

8) V_i error: correct value is $A_{MV,t} \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$. □

The robust B-tree [3] presented by Black, Taylor and Morgan performs double error detection or single error correction in $O(n)$ time, and requires $2m+3$ extra fields in each node of an order-m B-tree. Taylor and Black have also developed the LB-Tree [10] which is locally correctable, in that it can correct many single errors if they occur in separate substructures. However, in order to verify a pointer, one level of nodes must be traversed, and to correct a pointer, all the levels above the current level must be traversed. Hence, double error detection and single error correction require $O(n)$ time, and $2m+5$ extra fields in each node of an order-m B-tree are required. In comparison, the advantages of the VBT are as follows:

1) Double pointer errors can be detected in the VBT using an O(1) LCED procedure.

2) Single pointer errors can be corrected in the VBT using an O(1) LCEC procedure for an error detected during a forward move, or using an $O(\log_{2m} n)$

LCEC procedure for an error detected during a backward move.

3) The VBT requires only m+4 extra fields in each node.

4) The virtual backpointer facilitates backward traversals of the VBT, which can then be used to enhance performance.

IV. ANALYSIS AND IMPLEMENTATION OF A CONCURRENT AUDITOR PROCESS

The *Concurrent Auditor Process* (CAP) is an on-line process for error detection and correction that runs in parallel with user processes accessing a database. It is used, in this case, to perform data structure error detection and correction for the user processes, and allows concurrent access to structures being checked to reduce the system performance degradation due to error detection. Koved and Waldbaum have developed an auditor program that provides detection of computer subsystem failures [18], based on Waldbaum's concept of the auditor program [19]. Taylor, Morgan and Black have suggested the use of an audit program to periodically perform error detection and correction in data structures [1]. However, little analysis has been performed on the effectiveness of such an audit program. This section presents an analysis of the effectiveness of the CAP and presents measurements of the CAP's effectiveness in a Sequent Balance 8000 multiprocessor implementation using a database of VDLL.

The CAP described here accesses structures more frequently and uniformly than user processes to reduce the latency of error detection. Also, the CAP performs error detection in Checking Windows of higher cost than those used by user processes, to reduce their performance degradation. For example, if the database is composed of VDLL or VBT instances, user processes may perform single pointer error detection in W^2 with less computation cost, while relying on the CAP to detect the less-frequent double pointer errors in W^3 with more computation cost. The effectiveness of the CAP is determined by its increase of the mean time to failure (MTTF) of the

system. Ideally, a large increase is achieved with little degradation of user process performance. Hence, the CAP permits user processes to access structures being checked as long as they do not insert or delete nodes from the CAP's current Checking Window. Expressions are derived to determine the MTTF in a multi-user, n-process system with and without the use of the CAP. This is followed by the results of an implementation of the CAP using a VDLL database.

A. Analysis

In a multi-user, n-process shared-database environment, assume that the CAP performs error detection in W^3 and that user processes perform error detection in W^2. The pointer errors can then be divided into three classes: E_0, E_1 and E_2. E_0 errors are those which can be detected by a user process or by the CAP. E_1 errors can be detected by the CAP but not by a user process. E_2 errors can be detected by neither a user process nor the CAP. Suppose the time for an E_1 error to occur is $T_{E_1}^n$, the time for a user process to encounter that error is T_U, and the time for the CAP to detect an E_1 error is T_A. For the purposes of analysis assume, in a given time interval, both the number of errors that occur and the number of accesses to a particular node are random variables following a Poisson distribution. Then, random variables $T_{E_1}^n$, T_U and T_A follow an exponential distribution with mean time γ_1^n, β and α, respectively.

LEMMA 1: The probability of an E_1 error causing any of the n processes to fail in the presence of the CAP is $1-\left(\frac{\beta}{\alpha+\beta}\right)^n$.

PROOF: For a single process, the probability to fail can be derived using basic probability theory:

$$\text{Prob}(T_A>T_U) = \int_0^\infty \text{Prob}(T_U=x)\text{Prob}(T_A>x)dx = \int_0^\infty \frac{1}{\beta}e^{-x/\beta}e^{-x/\alpha}dx = \frac{\alpha}{\alpha+\beta}.$$

Therefore, the probability of any of the n processes failing is $1-\left(1-\frac{\alpha}{\alpha+\beta}\right)^n = 1-\left(\frac{\beta}{\alpha+\beta}\right)^n$. □

THEOREM 7: Without the use of the CAP, MTTF = $\gamma_1^n + \beta$, and with the use of the CAP,

$$\text{MTTF}_{\text{CAP}} = \min\left(\frac{\gamma_1^{n'}}{1-\left(\frac{\beta}{\alpha+\beta}\right)^n}, \gamma_2^{n'}\right) + \beta.$$

PROOF: If no CAP is used, MTTF = $\min(E(T_{E_1}^n), E(T_{E_2}^n)) + E(T_U) = \min(\gamma_1^n, \gamma_2^n) + \beta = \gamma_1^n + \beta$, where E(X) is the expected value of random variable X.

In the presence of the CAP, the determination of whether an E_1 error will cause a failure can be modeled as a Bernoulli trial with parameter $p = 1-\left(\frac{\beta}{\alpha+\beta}\right)^n$. Hence the MTTF_{CAP} follows a geometric distribution with mean $\frac{\gamma_1^{n'}}{p}$, where n' represents the effect of n user processes and the CAP. □

If E_1 and E_2 errors are formed by the accumulation of E_0 errors, then $T_{E_1}^n$ and $T_{E_2}^n$ are proportional to the access frequency. Thus $\gamma_1^n = n\gamma_1^1$, $\gamma_2^n = n\gamma_2^1$ and $\gamma_2^1 >> \gamma_1^1$. This gives, for the without-CAP case, MTTF = $\gamma_1^n + \beta = n\gamma_1^1 + \beta$. In the with-CAP case, since the CAP is $\frac{\beta}{\alpha}$ times faster in checking the data structure than a user process, $\gamma_1^{n'} = \left(n+\frac{\beta}{\alpha}\right)\gamma_1^1$. E_2 errors will retain an exponential distribution but with different mean $\gamma_2^{n'} = \left(n+\frac{\beta}{\alpha}\right)\gamma_2^1$. For this case the theorem gives

$$\text{MTTF}_{\text{CAP}} = \min\left(\frac{\left(n+\frac{\beta}{\alpha}\right)\gamma_1^1}{p}, \left(n+\frac{\beta}{\alpha}\right)\gamma_2^1\right) + \beta.$$

EXAMPLE 4: Suppose $\gamma_1^1 = 100$ hours, $\gamma_2^1 = 10{,}000$ hours, $\beta = 1$ minute, and 5 user processes are active on the system. Without the use of the CAP, MTTF $\simeq$ 500 hours. However, by using the CAP, and with $\alpha = 10$ seconds, MTTF is increased to $MTTF_{CAP}$ =2050 hours. □

If α is small enough (i.e., the CAP is fast enough), the $\frac{\left(n+\frac{\beta}{\alpha}\right)\gamma_1^1}{1-\left(\frac{\beta}{\alpha+\beta}\right)^n}$ term can exceed the $\left(n+\frac{\beta}{\alpha}\right)\gamma_2^1$ term. In this case, $MTTF_{CAP} = \left(n+\frac{\beta}{\alpha}\right)\gamma_2^1+\beta$. This effectively eliminates the chances of a user process failure due to E_1 errors, which occur more often than E_2 errors.

B. Implementation

A model database of VDLL was implemented in C and run on a Sequent Balance 8000 shared-memory multiprocessor system with six CPUs. Single random errors and worst-case double errors (called "double cooperative errors," where a second error masks a previous error) were injected into the database one at a time. Error detection was accomplished by one of four user processes, the database manager, or the CAP, each of which performed either W^2 or W^3 checking. The database manager serviced all update requests, and the CAP operated in the idle time of the database manager, to reduce performance degradation. Databases of 50, 100, 500 and 1000 nodes were used in the simulations. Each database consisted of eight VDLL instances: six non-empty instances, one empty instance, and a free list. To model the locality of user process database access, each user process performed approximately 80% of its operations (composed of 75% searches, 12.5% insertions and 12.5% deletions) within one VDLL, and the other 20% in a randomly selected VDLL.

For each single or double error injected, the detection latency and the number of operations completed in that time were measured, for five different combinations of user process LCED/CAP LCED (Table 7). The mean error detection latencies for the five combinations, applied to databases

of 50, 100, 500 and 1000 nodes, are shown in Table 8. Table 9 shows by what factor use of the CAP can decrease the error detection latency. The following observations can be made based on the results of the implementation:

1) Single and double LCED can be performed on the VDLL in O(1) time.

2) The use of the CAP significantly reduces the error detection latency of both single random errors and double cooperative errors.

3) The CAP is more effective in reducing the detection latency of single random errors as the size of the database increases.

Using the analysis results of the previous section, the first observation shows that $\gamma_1^{n'} \geq 5\gamma_1^n$. Thus from Theorem 5, the $MTTF_{CAP} > 5 \times MTTF$. This clearly shows the utility of the CAP in increasing the MTTF of the system.

V. SUMMARY

In this paper, we have presented a new technique for local concurrent error detection in linked data structures that can achieve O(1) error detection in a variety of data structures. This technique uses the concept of a Checking Window to define the locality in which local concurrent error detection is performed and also to determine the associated cost of the locality. The virtual backpointer was introduced and used to define two new data structures, the Virtual Double-Linked List, which incurs no storage overhead, and the B-Tree with Virtual Backpointers of order m, which requires $m+4$ extra fields per node. It was shown that double errors could be detected using a local concurrent error detection procedure in O(1) time for both structures. In addition, those errors detected during forward moves were shown to be correctable using a local concurrent error correction procedure in O(1) time. Correction of those errors detected during backward moves was shown to be, in worst case, $O(n)$. Finally, an analysis and implementation of a concurrent auditor

Table 7. Combinations of User Process LCED and CAP LCED.

Case	User Process LCED	CAP LCED
1	W^2	None
2	W^2	W^2
3	W^2	W^3
4	W^3	None
5	W^3	W^3

Table 8. Mean Error Detection Latencies.

Error Type	Database Size	Number of Samples	Case 1	2	3	4	5
Single Random Error	50	10000	77	8	7	64	7
	100	10000	144	11	10	127	10
	500	1800	4884	147	134	5052	140
	1000	200	29087	372	308	31033	312
Double Cooperative Error	50	10000	72	7	7	39	7
	100	10000	60	13	10	57	11
	500	1800	420	54	48	447	50

Table 9. Detection Latency Reduction Factor Through Use of the CAP.

Error Type	Database Size	Cases Compared 1:2	1:3	4:5
Single Random Error	50	10	11	9
	100	13	14	13
	500	33	37	36
	1000	78	94	99
Double Cooperative Error	50	10	10	6
	100	5	6	5
	500	8	9	9

process in a shared database using the virtual backpointer technique was shown to significantly reduce the error detection latency.

REFERENCES

[1] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 585-594, November 1980.

[2] D. J. Taylor, D. E. Morgan, and J. P. Black, "A Compendium of Robust Data Structures," *Proceedings of the 11th Annual International Symposium on Fault Tolerant Computing*, pp. 129-131, June 1981.

[3] J. P. Black, D. J. Taylor, and D. E. Morgan, "A Robust B-Tree Implementation," *Proceedings of the 5th International Conference on Software Engineering*, pp. 63-70, March 1981.

[4] J. I. Munro and P. V. Poblete, "Fault Tolerance and Storage Reduction in Binary Search Trees," *Information and Control*, vol. 62, pp. 210-218, 1984.

[5] M. C. Sampaio and J. P. Sauvé, "Robust Trees," *Proceedings of the 15th Annual International Symposium on Fault Tolerant Computing*, pp. 23-28, June 1985.

[6] S. C. Seth and R. Muralidhar, "Analysis and Design of Robust Data Structures," *Proceedings of the 15th Annual International Symposium on Fault Tolerant Computing*, pp. 14-19, June 1985.

[7] K. Yoshihara, Y. Koga, and T. Ishihara, "A Robust Data Structure Scheme with Checking Loops," *Proceedings of the 13th Annual International Symposium on Fault Tolerant Computing*, pp. 241-248, June 1983.

[8] K. Kuspert, "Efficient Error Detection Techniques for Hash Tables in Database Systems," *Proceedings of the 14th Annual International Symposium on Fault Tolerant Computing*, pp. 198-203, June 1984.

[9] J. P. Black and D. J. Taylor, "Local Correctability in Robust Storage Structures," to appear: *IEEE Transactions on Software Engineering*.

[10] D. J. Taylor and J. P. Black, "A Locally Correctable B-Tree Implementation," *The Computer Journal*, vol. 29, no. 3, pp. 269-276, 1986.

[11] I. J. Davis and D. J. Taylor, "Local Correction of Mod(k) Lists," CS-85-55, Dept. of Computer Science, University of Waterloo, December 1985.

[12] I. J. Davis, "Local Correction of Helix(k) Lists," CS-86-30, Dept. of Computer Science, University of Waterloo, August 1986.

[13] I. J. Davis, "A Locally Correctable AVL Tree," to appear: *17th Annual International Symposium on Fault-Tolerant Computing*, July 1987.

[14] W. K. Fuchs, "A Specification-Based Approach to Concurrent Structure Verification in Multiprocessor Systems," *IEEE International Conference on Computer Design*, pp. 375-378, October 1986.

[15] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Some Theoretical Results," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 595-602, November 1980.

[16] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173-189, 1972.

[17] P. Elias, "Error-Free Coding," *IRE Transactions on Information Theory*, vol. IT-4, pp. 29-37, 1954.

[18] L. Koved and G. Waldbaum, "Improving Availability of Software Subsystems through On-Line Error Detection," *IBM Systems Journal*, vol. 25, no. 1, pp. 105-115, 1986.

[19] G. Waldbaum, "Audit programs - A Proposal for Improving System Availability," Research Report RC-2811, IBM Thomas J. Watson Research Center, February 1970.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-87-2264 CSG-73	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA ONR
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center MS 130 Hampton, VA 23665 see back additional address
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA ONR 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG 1-602 N00014-86-K-0519
8c. ADDRESS (City, State, and ZIP Code) NASA Langley Research Ctr. MS 130 Hampton, VA 23665 see back additional address	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Local Concurrent Error Detection And Correction In Data Structures Using Virtual Backpointers

12. PERSONAL AUTHOR(S)
Li, C.C. , Chen, P.P. , Fuchs, W.K.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1987 October	39

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	concurrent error detection, data structures, concurrent structure checking

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
A new technique, based on virtual backpointers, for local concurrent error detection and correction in linked data structures is presented in this paper. Two new data structures, the Virtual Double-Linked List, and the B-Tree with Virtual Backpointers, are described. For these structures, double errors can be detected in O(1) time and errors detected during forward moves can be corrected in O(1) time. The application of a concurrent auditor process to data structure error detection and correction is analyzed, and an implementation is described, to determine the effect on the mean time to failure of a multi-user shared-database system. The implementation utilizes a Sequent shared-memory mulitprocessor system operating on a shared databased of Virtual Double-Linked Lists.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL