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

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		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
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 ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

7b. Department of the Navy
Office of Naval Research
Detachment Pasadena
1030 East Green Street
Pasadena, CA 91106-2485

8c. Department of the Navy
Office of Naval Research
Detachment Pasadena
1030 East Green Street
Pasadena, CA 91106-2485

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

fuchs@bach.csg.uiuc.edu

ABSTRACT

A new technique, based on virtual backpointers, is presented in this paper for local concurrent error detection and correction in linked data structures. Two new data structures utilizing virtual backpointers, the Virtual Double-Linked List and the B-Tree with Virtual Backpointers, are described. For these structures, double errors can be detected in constant time and single errors detected during forward moves can be corrected in constant time.

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

---

This research was supported in part by the SDIO/IST and managed by the Office of Naval Research under contract N00014-86-K-0519, in part by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-602, and in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Contract N00014-84-C-0149.

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 can be important in achieving overall system reliability. Detection and correction algorithms, when used concurrently with normal data structure access, typically degrade performance. To reduce this degradation, these algorithms should not access a large number of nodes that lie off the intended traversal path. If data structure checking operations are performed in a small locality around a currently accessed node, and execute in constant time, then error detection and correction can potentially be performed concurrently with normal data structure accesses without severely degrading the system performance. In addition, an arbitrary number of errors in the data structure may be detected and corrected assuming not too many exist within a given locality.

The foundation work concerning robust data structures was performed by Taylor, Morgan and Black [1,2]. Several techniques have since been developed to achieve robust data structures. Such structures include the modified(k) double-linked list, the chained and threaded binary tree, and the robust B-tree, by Taylor, Morgan and Black [1,3,4], the isomorphic binary tree by Munro and Poblete [5], the robust binary tree by Sampaio and Sauvé [6], and the mod(2) chained and threaded binary tree by Seth and Muralidhar [7]. In general, global detection techniques are used on these data structures. Similar techniques are also used on the three pointer tree, as explained by Yoshihara et al [8]. Though not indicated in their paper, single error detection can be performed in constant time within the D-loop localities in the structure. Kuspert's separately-chained hash table [9], which is an application of double-linked lists, guarantees single error detection in constant time through the use of five extra fields stored in each node. The concepts of local detectability and local correctability were introduced by Black and Taylor [10], and have been applied to several data structures, including the spiral(k) list [10], the LB-tree [10,11], the mod(k) list [12], the helix(k) list [13] and the AVL tree [14].

As a means of preserving the *structural* integrity of linked data structures, a new approach to detecting and correcting structural errors, called the *virtual backpointer*, is introduced in this paper. The virtual backpointer provides the capabilities of structural error detection and correction as well as the generation of backpointers. Two data structures are constructed in this paper using the virtual backpointer: the Virtual Double-Linked List and the B-Tree with Virtual Backpointers. The Virtual Double-Linked List requires the same amount of storage as the standard double-linked list from which it is derived, but possesses higher error detection and correction capabilities. The B-Tree with Virtual Backpointers, derived from the B-tree of order m, requires m+4 more fields in each node than the standard B-tree, with the benefit that it facilitates backward traversals as well as error detection and correction. The Virtual Double-Linked List and the B-Tree with Virtual Backpointers can be shown to possess the properties of local detectability and local correctability, according to the definitions of these properties introduced by Black and Taylor [10]. However, these definitions do not give a specific measure of the size of the locality within which local detection and local correction are performed. The concept of a checking window is introduced in this paper as a mechanism for specifically stating the size of this locality. Detectability and correctability within a window are presented as specific functions of the window size.

The organization of this paper is as follows. Section II presents the virtual backpointer and checking window concepts. In Section III, the Virtual Double-Linked List is described and its detection and correction capabilities are analyzed; similarly, Section IV describes and analyzes the B-Tree with Virtual Backpointers. Section V presents the results of experiments performed with the VDLL and the VBT to determine the effectiveness and the time overhead of local concurrent error detection and correction. Finally, Section VI provides a summary of this work.

II. VIRTUAL BACKPOINTERS

Linked data structures, as considered in this paper, are composed of nodes linked by pointers. Nodes consist of *components*, where a component has a type (e.g., pointer, checking symbol, backpointer) and a value. An instance mapping I accesses the components and interprets them to obtain their values. If all the nodes of a data structure contain the same number and types of components in the same order, then that structure is said to be *uniform*. A *move* from a node N_j to a node N_k is possible if there exists a pointer from N_j to N_k, in which case N_k is *reached* from N_j by *following* that pointer. Two moves are *similar* if the pointers for those moves are derived from the same types of components at identical positions within their respective nodes. A series of moves that begins at the header node(s) of a data structure and accesses part or all of the structure is called a *traversal*.

The errors considered in this paper are those that affect the structural information of the data structure (i.e., structural components). To simplify the error detection and correction analyses, it is assumed that instances of a uniform data structure reside in a node space in virtual memory, and that no node external to a correct instance contains information which could be interpreted as a pointer into that instance. This latter assumption is similar to the Valid State Hypothesis of Taylor, Morgan, and Black [2], but differs in that there is no assumption concerning identifier fields.

In this paper, A_i represents the memory address of a node N_i in a linked data structure. N_i may have many pointers to other nodes, with $N_i \rightarrow N_{MV}$ representing a desired move from N_i, following a pointer, to reach a node N_{MV}. The pointer value may be given directly by I from one of the components of N_i or may be a function of the values (also given by I) of several components.

The virtual backpointer formally defined below can provide the address of a parent N_{parent} of a node N_i. 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 possible moves from $N_{ancestor}$ to N_i.

DEFINITION 1: Let $N_{ancestor}$ be an ancestor of N_i, and Q_i be the set of pointers in N_i. A *virtual backpointer*, V_i, is: $V_i = f(Q_i, A_{ancestor})$, 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 ancestor nodes $\vec{A}$. □

The standard backpointer is a degenerate case of the virtual backpointer, with $f(Q_i, A_{ancestor}) = A_{ancestor}$ and $f^*(Q_i, V_i) = V_i$. The virtual backpointers developed in this paper have the following properties. 1) V_{MV} is used for structural error detection and correction for a forward move $N_i \rightarrow N_{MV}$. 2) V_{MV} provides the backpointer for a backward move $N_{MV} \rightarrow N_i$ after transformation by f^*, and Q_i is used for structural error detection and correction. Although not considered here, the virtual backpointer could be generalized to a *virtual pointer*. In that case, the values generated by f^* would be the addresses of nodes which are not constrained to be ancestors of the current node.

Virtual backpointers are used in this paper to perform error detection within a locality of a specific size in the data structure. This locality is formalized as a *checking window*. A checking window of size s is a set of s adjacent nodes in a data structure.

DEFINITION 2: For a given move $N_i \rightarrow N_{MV}$ in a uniform data structure, the set of *checking windows* of size s is: $W^s = \{W_j^{s-1} \cup N_k \mid W_j^{s-1} \in W^{s-1}, N_k \notin W_j^{s-1}$, and N_k is adjacent to some node in $W_j^{s-1}\}$. The base case, determined by the move, is: $W^2 = \{W_1^2\} = \{\{N_i, N_{MV}\}\}$. □

This definition says that a window W_m^s, for some m, is constructed by adding one more node N_k to the smaller checking window W_j^{s-1}, such that N_k can be reached from some node in W_j^{s-1} in one move. Each W_m^s is a set of nodes, and all W_m^s for a particular move together form a set, W^s. Since the data structure is uniform, W^s for a given move will be isomorphic to W'^s for any other similar move in that structure. In our discussion, two nodes of any W_j^s will be written with a node's son to the node's right.

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

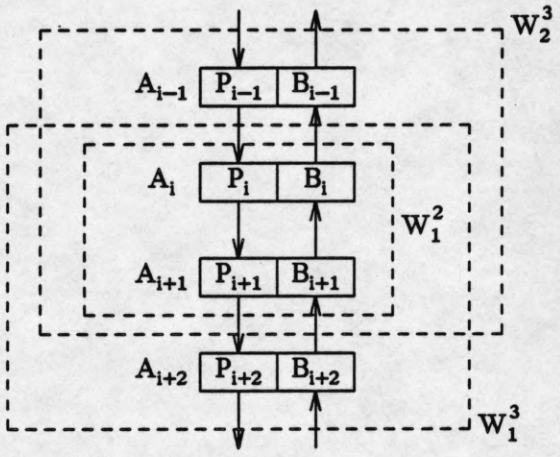

Figure 1. Checking Windows for a Double-Linked List.

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

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

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

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

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

etc. □

Structural error detection is accomplished in the two data structures of this paper by evaluating checking predicates within a checking window. A *checking predicate* performs one-error detection on the components on which it is defined. That is, it returns "True" for zero erroneous components, "False" for one erroneous component, and either "True" or "False" for multiple erroneous components. If all the components which a predicate uses are available in a particular checking window, then that predicate is *evaluable* in that window. Clearly, isomorphic checking windows will have identical sets of evaluable checking predicates. An example checking predicate is the *vote* of Black and Taylor [10]. *Constructive votes* generate possible values for a component under evaluation and, together with *diagnostic votes*, elect one value as the correct value of that

component. The component under evaluation is called the *principal component*, *c*.

The minimum number of errors within a checking window of size *s* that can mask an incorrect move due to an erroneous component will be used to evaluate the error detection capabilities of the data structures of this paper. This is similar to the concept of *changes* used by Taylor, Morgan, and Black [2] to determine the distance between two data structure instances, in a global context. The differences here are first, that the distance is measured within a fixed-size checking window, and second, that no global information is used in the determination of the distance.

DEFINITION 3: For a given move MV in a uniform data structure and a specific checking window W_j^s, the *local distance* of the window, $d_j^s(MV)$, is the minimum number of erroneous components in W_j^s required to mask an incorrect move due to an erroneous component, by causing all the checking predicates within W_j^s to return "True" as a false indication of a no-error condition. □

The error detectability of a specific move and a set of checking windows of size *s* can now be described, in terms of this local distance.

DEFINITION 4: For a given move MV in a uniform data structure and for the set of windows W^s, the *detectability* in a window, $D^s(MV)$, is $D^s(MV) = max(d_j^s(MV)) - 1, 1 \leq j \leq |W^s|$. □

For a particular move, different checking windows of the same size do not consist of the same nodes. Consequently, they do not necessarily achieve the same detectability. This is the reason that the *max* function is used: for a given move, it is always possible to use the particular W_j^s that gives the greatest detectability. That particular window is called W_*^s, and its corresponding local distance is d_*^s. Isomorphic instances of W_*^s are used for all similar moves. For a uniform data structure, the parameter MV may be omitted from $d_j^s(MV)$ or $D^s(MV)$, when the context is clear. A consequence of Definition 4 is that $D^s(MV) = D^s(MV')$, if MV and MV' are any two similar moves in the uniform data structure.

The following theorem shows that the detectability D^s is the same for both forward and backward moves, if forward and backward pointers (virtual backpointers) can be paired in one-to-one mapping. This assumes that no special knowledge (i.e., the known addresses of the header

nodes) is used if the move concerns the header nodes. This theorem will be used in the determination of $D^s(MV)$ for the Virtual Double-Linked List and the B-Tree with Virtual Backpointers.

THEOREM 1: In a uniform data structure, if there exists a one-to-one mapping between pointers allowing a move from $N_i \rightarrow N_k$ and pointers allowing a move from $N_k \rightarrow N_i$, then $D^s(N_i \rightarrow N_k) = D^s(N_k \rightarrow N_i) = D^s$.

PROOF: Let $N_i \rightarrow N_k$ represent any forward move, and $N_k \rightarrow N_i$ represent the backward move. Because the data structure is uniform, the case where $N_i \rightarrow N_k$ is changed by an error to $N_i \rightarrow N_{k'}$ is isomorphic to the case where $N_k \rightarrow N_i$ is changed by an error to $N_k \rightarrow N_{i'}$. Any W_j^s constructed for $N_i \rightarrow N_{k'}$ will be isomorphic to some W_m^s constructed for $N_k \rightarrow N_{i'}$. Thus, the W^s for $N_i \rightarrow N_{k'}$ will be isomorphic to the W^s for $N_k \rightarrow N_{i'}$, such that a one-to-one mapping of isomorphic windows will exist between the members of each W^s. Since the evaluable checking predicates are identical for each pair of isomorphic windows, then by Definition 3, each pair of isomorphic windows will have the same $d_j^s(MV)$. Thus, by Definition 4, $D^s(N_i \rightarrow N_k) = D^s(N_k \rightarrow N_i) = D^s$. □

Theorem 2 shows that the detectability $D^s(MV)$ in a uniform data structure is a monotonically non-decreasing function of the window size s. This result will be used to determine the upper bounds of $D^s(MV)$ for the Virtual Double-Linked List and the B-Tree with Virtual Backpointers.

THEOREM 2: For a given move MV in an n-node instance of a uniform data structure, and for all possible sets of erroneous components in that instance, if all evaluable checking predicates used in W_j^{s-1} are used in W_m^s where $W_j^{s-1} \subset W_m^s$, then $D^{s-1}(MV) \leqslant D^s(MV) \leqslant D^n(MV)$, $3 \leqslant s \leqslant n$.

PROOF: By Definition 2, every W_m^s is constructed by adding one adjacent node N_k to a checking window of size s-1: $W_m^s = W_j^{s-1} \cup N_k$. Any checking predicate in W_j^{s-1} that evaluates to "True" will remain "True" in W_m^s, because all the components used by that predicate in W_j^{s-1} retain their values in W_m^s. If the addition of N_k causes an unevaluable checking predicate in W_j^{s-1} to evaluate to "True" in W_m^s, this results in $d_m^s = d_j^{s-1}$. However, if the predicate evaluates to "False," then $d_m^s > d_j^{s-1}$, since at least one other error would be required to mask the detected error. Hence,

$d_m^s \geq d_j^{s-1}$. Then, $max(d_m^s) \geq max(d_j^{s-1})$, and from Definition 4, $D^s \geq D^{s-1}$. The upper limit of detectability is trivially D^n, since the checking window has reached its maximum size (it contains the entire structure). □

A general approach for performing error detection in a particular uniform data structure using checking windows is outlined in the following discussion. For the data structure chosen, each possible type of move in the structure (e.g., forward, backward) is identified. For each type, a representative move MV and the desired level of detectability $D^s(MV)$ are chosen. Checking windows of increasing size (starting with W_1^2, the smallest window) are analyzed using Definitions 3 and 4 to determine their detectabilities $D^s(MV)$ for each representative move. Once the desired level of detectability is achieved, the corresponding checking window is labeled W_*^s. It is only W_*^s that is constructed for every move made when accessing instances of this particular data structure. Finally, for each type of possible move, the checking predicates are identified that will detect erroneous components in W_*^s.

The preceding steps need to be performed only once for a given data structure. For every move made when accessing nodes of instances of the structure, W_*^s can be constructed and the checking predicates evaluated. If all evaluable predicates return "True," then either no error has occurred or undetectable errors have occurred; if any evaluable predicate returns "False," then at least one error has been detected.

Once an error has been detected, correction may be performed. The ability to perform correction relies on the existence of a correction procedure. Like detection, correction is performed within checking windows. The number of errors in the window that can be corrected is defined in the following.

DEFINITION 5: For a given move MV in a uniform data structure and for the set of windows W^s, the *correctability* in a window, $C^s(MV)$, is the maximum number of erroneous components that may exist in W_*^s such that for any set of erroneous components of cardinality $\leq C^s(MV)$, a detection procedure will detect the erroneous components, and when invoked after the application

of a correction procedure to W^s_*, the detection procedure will correctly indicate an error-free window, guaranteeing the correctness of the move. □

In order to obtain the desired levels of detectability and correctability, a large window may be necessary. Typically, the cost of performing error detection and correction will increase as the window size increases. By specifying the size of the locality within which local detection and correction are performed, a measure of the cost of those procedures may be made. A larger checking window will, in general, have a larger set of evaluable checking predicates and detection within that window will have a correspondingly greater computational cost. Similarly, the larger window will contain more nodes, so that the number of nodes accessed will be larger.

Since errors are detected and corrected based only on information from nodes in the checking window, many other detectable and correctable errors may exist simultaneously throughout the data structure. Although the detectability D^s and correctability C^s may only be one or two for single moves in the data structure, the total number of detectable and correctable errors in the entire structure may be much greater since D^s and C^s are measured relative to a checking window. Black and Taylor [10] were the first to introduce local detection and correction algorithms, which permit the correction of an arbitrary number of errors, provided the errors are in some sense sufficiently separated from each other. The checking window concept of this paper provides a measure of the "sufficient separation" of errors, which is the size of the window: for W^s_*, the sufficient separation is s nodes between every error.

III. VIRTUAL DOUBLE-LINKED LIST

The *Virtual Double-Linked List* (VDLL) is a uniform data structure that employs the virtual backpointer for local error detection and correction and for backward traversals. The VDLL requires the same storage space as the standard double-linked list (DLL), and retains the simplicity of the DLL, since it is possible to move directly from a node to its parent, using the virtual

backpointer.

DEFINITION 6: 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 $V_i = f(\{P_i\}, A_{i-1}) = P_i \oplus A_{i-1}$, thus $A_{i-1} = f^*(\{P_i\}, V_i) = P_i \oplus V_i$, where $\oplus$ denotes the logical exclusive-or function. The VDLL is created from the DLL by replacing the backpointers in the DLL with virtual backpointers. Also, s header nodes $N_0, N_{-1}, \cdots, N_{-s+1}$ are added, where s is the size of the checking window. These header nodes are assumed to be always accessible. Note that $N_{-s+1} \equiv N_\infty$. □

A similar construction can be applied to the modified(k) DLL family [1] resulting in the modified(k) VDLL structures.

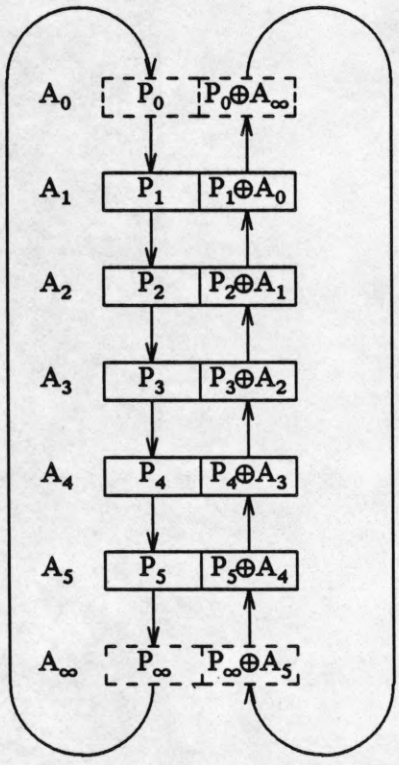

Figure 2. Virtual Double-Linked List (VDLL) of 5 nodes.

DEFINITION 7: 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 $V_i = f(\{P_i\}, A_{i-k}) = P_i \oplus A_{i-k}$, thus $A_{i-k} = f^*(\{P_i\}, V_i) = P_i \oplus V_i$. Also, *max*(k+1, *s*) header nodes are added. □

Local correction may be accomplished in the VDLL using the theoretical results of Black and Taylor [10]. The linearization function as described in their theory corresponds to the forward traversal of the VDLL. For each principal component $c = P_i$, there are two constructive votes (P_i, $V_i \oplus A_{prev}$) and one diagnostic vote ($P_x \oplus V_x \stackrel{?}{=} A_i$). ($N_{prev}$ is the parent of the current node N_i, and N_x is accessed using a candidate value.) These votes can be shown to possess the property of *distinctness*, which is a precondition for their results. By the r-local-detectable and r-local-correctable theorems of Black and Taylor [10], the VDLL can be shown to be 2-local-detectable and 1-local-correctable. However, local detectability and correctability in this case are not specified as a function of the size of the locality. The detectability D^s and correctability C^s, which specifically state the size of the locality as the window size *s*, are therefore analyzed for the VDLL in Theorems 3 and 4.

THEOREM 3: The detectability D^s(MV) of the VDLL is D^2(forward) = D^2(backward) = $D^2 = 1$, and D^s(forward) = D^s(backward) = $D^s = 2, \forall s \geq 3$.

PROOF: Since the VDLL is a uniform data structure and has a virtual backpointer for each forward pointer, D^s(forward) = D^s(backward) by Theorem 1. Consider a forward move $N_i \rightarrow N_{i+1}$ following P_i. 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 to $A_{j+1} \oplus A_{i-1}$ to mask the error in P_i: $d_2^3 = 3$.

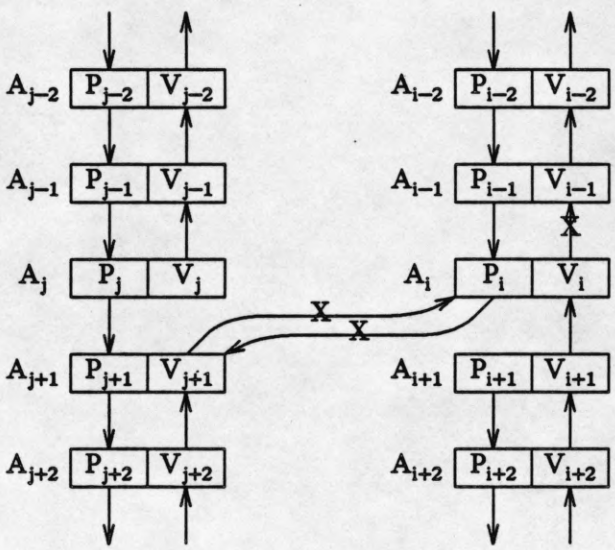

Figure 3a. Analysis of VDLL: Errors in P_i, V_i, and V_{j+1}.

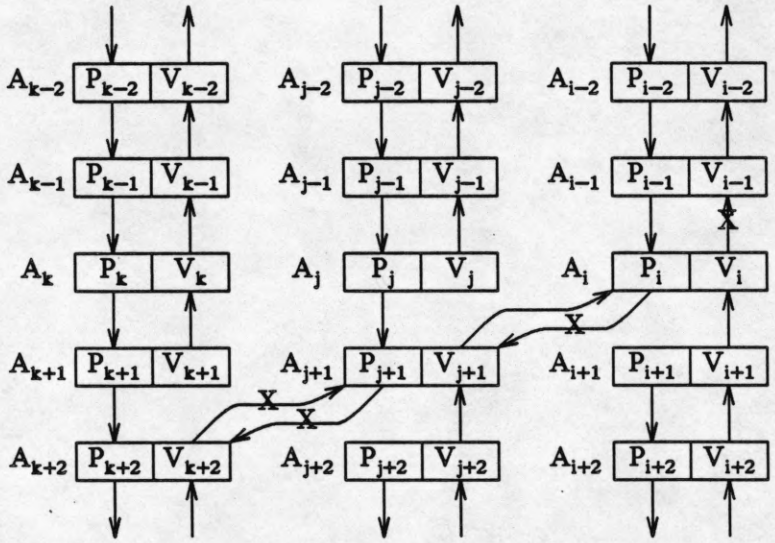

Figure 3b. Analysis of VDLL: Errors in P_i, V_i, P_{j+1}, and V_{k+2}.

Assume now that V_{j+1} is not erroneous, so P_{j+1} must be erroneous (Figure 3b) to mask the error in P_i. Consider $W_1^3 = \{N_i, N_{j+1}, N_{k+2}\}$. The P_i error will be undetectable 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 4, $D^2 = 1$ and $D^3 = 2$. Since the VDLL can be changed to another correct VDLL by three erroneous components (node deletion), then $D^n = 2$, where n is the number of nodes in the structure. By Theorem 2, $D^s = 2, \forall s \geqslant 3$. □

The above proof suggests that for a forward move, $W_*^2 = \{N_i, N_{MV}\}$ and $W_*^3 = \{N_{prev}, N_i, N_{MV}\}$, where N_{prev} corresponds to N_{i-1} in the proof. For a backward move, $W_*^2 = \{N_{MV}, N_i\}$ and $W_*^3 = \{N_{next}, N_{MV}, N_i\}$, where N_{next} is the node reached by following $P_{MV} \oplus V_{MV}$. By using these windows, double component errors can be detected, or single component errors corrected (Theorem 4 below). To verify a forward move, two constructive votes are evaluated (Figure 4a). The first vote (P_i) yields the first candidate, and the second vote ($A_{prev} \oplus V_i$) yields the second candidate. If the candidates are unequal, then an error has been detected, and the move may be incorrect. Otherwise, the move is correct.

To verify a backward move, one constructive vote ($P_i \oplus V_i$) and one diagnostic vote (P_{MV} ?= A_i) are evaluated (Figure 4b). If the diagnostic vote returns "False," then an error has been detected and the move may be incorrect. Otherwise, the move is correct.

```
(1)  candidate_1 = P_i
(2)  candidate_2 = A_prev ⊕ V_i
(3)  if (candidate_1 ≠ candidate_2) then
(4)     Error detected
(5)  else
(6)     Access N_MV via candidate_1
(7)     Set N_i = N_MV
```

Figure 4a. Detection Steps, VDLL Forward Move.

---

(1) *candidate* = $P_i \oplus V_i$
(2) Access N_{MV} via *candidate*
(3) **if** ($P_{MV} \neq A_i$) **then**
(4) Error detected
(5) **else**
(6) Set $N_i = N_{MV}$

Figure 4b. Detection Steps, VDLL Backward Move.

Figures 4a and 4b represent the steps used to detect errors between two adjacent VDLL nodes, accessing at most one node off the intended traversal path. For any W_j^2, these steps constitute the complete detection procedure. For larger windows, the steps may be repeated between every pair of consecutive nodes in the window, if desired, as the nodes are added to the window; this pessimistic procedure would evaluate every evaluable checking predicate in the window. An optimistic procedure would, for example, only verify the current move, letting later moves to check the other components in the window.

Table 1 gives the detectabilities D^s for the VDLL, modified(2) VDLL and modified(3) VDLL, compared to the DLL, modified(2) and modified(3) DLL [1], all without global counts or ID components, for various sized checking windows. The detectabilities D^s of the modified(2) and modified(3) VDLL can be obtained using a similar analysis as that applied to the VDLL. The VDLL and modified(2) VDLL achieve greater detectability than the DLL and modified(2) DLL, respectively. The modified(3) VDLL achieves no greater detectability than the modified(3) DLL, and because of certain pathological cases may exhibit less detection capability for small checking windows. It can be shown that the maximum detectability D^s for either the modified(k) DLL or the modified(k) VDLL is 3, $\forall k > 3$.

The following theorem provides the correctability C^s(MV) for the VDLL.

Table 1. Detectability D^s
of Several Linked List Data Structures.

D^s	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	2	2	1	2	2
D^6	2	3	2	1	2	3
D^7	2	3	2	1	2	3
D^8	2	3	3	1	2	3

THEOREM 4: In the VDLL, the correctability C^2(forward) = 0; C^s(forward) = 1 $\forall s > 2$, and at most one extra node is accessed for the correction of one error. Also, C^s(backward) = 0, $\forall s$.

PROOF: For the VDLL, the detectability D^2 is 1, the local distance is 2 and the correctability C^2 is 0. Similarly, the detectability D^s is 2 for $s > 2$, the local distance is 3 and therefore the upper limit on correctability within a window C^s is 1. Only single error cases in W_*^3 are considered.

If the detection steps of Figure 4a have detected an error during a forward move following P_i, then the correction steps given in Figure 5 can be invoked, where the principal component c is P_i. Once these steps have completed, the detection steps of Figure 4a can be reapplied to the original window. Assume an error is detected, i.e., the two constructive votes do not agree. This case arises if $P_i \neq A_{prev} \oplus V_i$; however, either Step 9 or 14 has guaranteed that this condition will be satisfied (single error assumed). This is a contradiction, so the two constructive votes must agree. Therefore, the assumption that an error has been detected is false, guaranteeing the correctness of the move. Then, by Definition 5, the correctability C^s(forward) = 1.

For a backward move following $P_i \oplus V_i$, the principal component is $c = V_i$, and there is only one constructive vote (V_i). To correct one error there must be at least two constructive votes.

```
(1)   candidate_1 = P_i
(2)   candidate_2 = V_i⊕A_prev
(3)   Access N_MV via candidate_1
(4)   val_1 = (P_MV⊕V_MV ?= A_i)
(5)   Access N_MV' via candidate_2
(6)   val_2 = (P_MV'⊕V_MV' ?= A_i)
(7)   if (val_1 = false) then
(8)      if (val_2 = true) then
(9)         Set P_i = V_i⊕A_prev
(10)     else
(11)        Uncorrectable: multiple errors
(12)  else
(13)     if (val_2 = false) then
(14)        Set V_i = P_i⊕A_prev
(15)     else
(16)        No error
```

Figure 5. Correction Steps, VDLL Forward Move.

Hence, the correctability C^s(backward) = 0. □

For a forward move, the correction procedure executes in constant time and requires one extra node access. For a backward move, the worst-case error (V_i or V_{MV}) cannot be corrected using only the information available in the window, and requires a global traversal for correction. It can be shown, however, that an error detected in P_i or P_{MV} during a backward move can be corrected in constant time using only information in the window.

IV. B-TREE WITH VIRTUAL BACKPOINTERS

The *B-Tree with Virtual Backpointers* (VBT) of order *m* is another uniform data structure which features the capabilities of local error detection and correction and performing backward traversals without a stack, using the virtual backpointer. The underlying structure of the VBT is the B-tree of order *m* [15], which finds frequent application in the construction and maintenance of

large-scale search trees. The standard 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. Keys in the i^{th} subtree will be less than the i^{th} key, and keys in the $i+1^{th}$ subtree will be greater than the i^{th} key.

DEFINITION 8: A *B-Tree with Virtual Backpointers* (VBT) of order m is described as follows (Figure 6). In a standard B-tree of order m, let $P_{i,j}$ be the j^{th} pointer in node N_i (Figure 6a). Recall that the address of N_i is A_i. Assume that each component requires one word of memory. Therefore, each pointer is uniquely addressable by $A_{i,j}$. The VBT is modified from the B-tree in the following ways.

1) A header node N_0 is created with $P_{0,j} = A_{1,0}$ for $0 \leqslant j \leqslant 2m$. The empty VBT consists of N_0 with all null pointers.
2) V_i, the virtual backpointer of N_i, is defined as $V_i = P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus A_{parent,r}$ where the r^{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, $K_{i,1}, K_{i,2}, \cdots, K_{i,2m}$ are arranged in a matrix (Figure 6b) and the *key check symbols* $X_{i,j}$ and $Y_{i,j}$ are generated using a product code [16]:

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

$K_{i,j}$ 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.

4) Each node will be aligned at an address in virtual memory that is a multiple of a constant, the *alignment constant*, which is greater than or equal to the size of a node. □

The number of key fields used in N_i is called $count_i$, which is added for performance enhancement. A leaf node may be distinguished from an internal node by all its pointers being null; an ID field may be used if quick identification is required. The forward move in the VBT simply follows a forward pointer. The backward move, $N_i \rightarrow N_{MV}$, is accomplished by the following steps: 1) $A_{MV,r}$ is generated via $V_i \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$. 2) $A_{MV} = \mathrm{int}(A_{MV,r}/\text{alignment constant}) \times$

alignment constant. 3) N_{MV} is accessed using A_{MV}. A VBT of order 2 is illustrated in Figure 6c.

Local correction may be accomplished in the VBT using the general theory of Black and Taylor [10]. The linearization function required in their approach corresponds to a forward traversal of the VBT from header N_0 to a leaf. For each principal component $c = P_{i,j}$ there are two constructive votes ($P_{i,j}$, $V_i \oplus A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m}$, where $A_{prev,r}$ is the address of the pointer $P_{prev,r}$) and one diagnostic vote ($V_x \oplus P_{x,0} \oplus P_{x,1} \oplus \cdots \oplus P_{x,2m} \oplus ?= A_{i,j}$). ($N_{prev}$ is the parent of the current node N_i, and N_x is accessed using a candidate value). These votes are distinct and by the r-local-detectable and r-local-correctable theorems of Black and Taylor [10], the VBT can be shown to be 2-local-detectable and 1-local-correctable.

As in the VDLL case, in order to specify the detectability and correctability as a function of the size of the locality, D^s and C^s are now analyzed for the VBT. Using Theorem 2, the possible component errors that can occur in the VBT are presented in Table 2 (errors in the count field are covered by the fifth and sixth rows of the table) along with the number of errors required to mask them. Each column of local distances in the table uses the same checking window, W_*^s, so that the maximum detectability D^s in each column is computed from the minimum local distance in that column.

THEOREM 5: The detectability $D^s(MV)$ of the VBT is $D^2(forward) = D^2(backward) = D^2 = 1$, and $D^s(forward) = D^s(backward) = D^s = 2, \forall s \geq 3$.

PROOF: From Table 2, the worst error has $d_*^2 = 2$ and $d_*^s = 3, \forall s \geq 3$. It follows from Definition 4 that $D^2 = 1$ and $D^s = 2, \forall s \geq 3$. □

Table 2 shows that no increase in the detectability D^s can be gained for $s > 3$. It can be shown that when moving forward $N_i \rightarrow N_{MV}$, $W_*^2 = \{N_i, N_{MV}\}$ and $W_*^3 = \{N_{prev}, N_i, N_{MV}\}$. When moving backward $N_i \rightarrow N_{MV}$, $W_*^2 = \{N_{MV}, N_i\}$ and $W_*^3 = \{N_{next}, N_{MV}, N_i\}$. Use of these windows allows detection of double component errors, or correction of single component errors (Theorem 6 below). To verify a forward move, two constructive votes are evaluated (Figure 7a). The first vote ($P_{i,j}$) yields the first candidate, and the second vote ($V_i \oplus A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m}$) yields

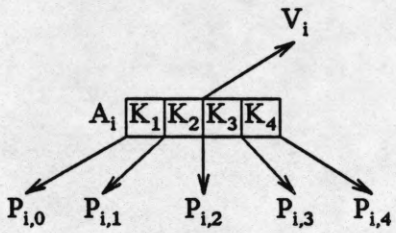

Figure 6a. Node Representation, Order-2 VBT.

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

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

Figure 6b. VBT Virtual Backpointer and Key Check Symbols.

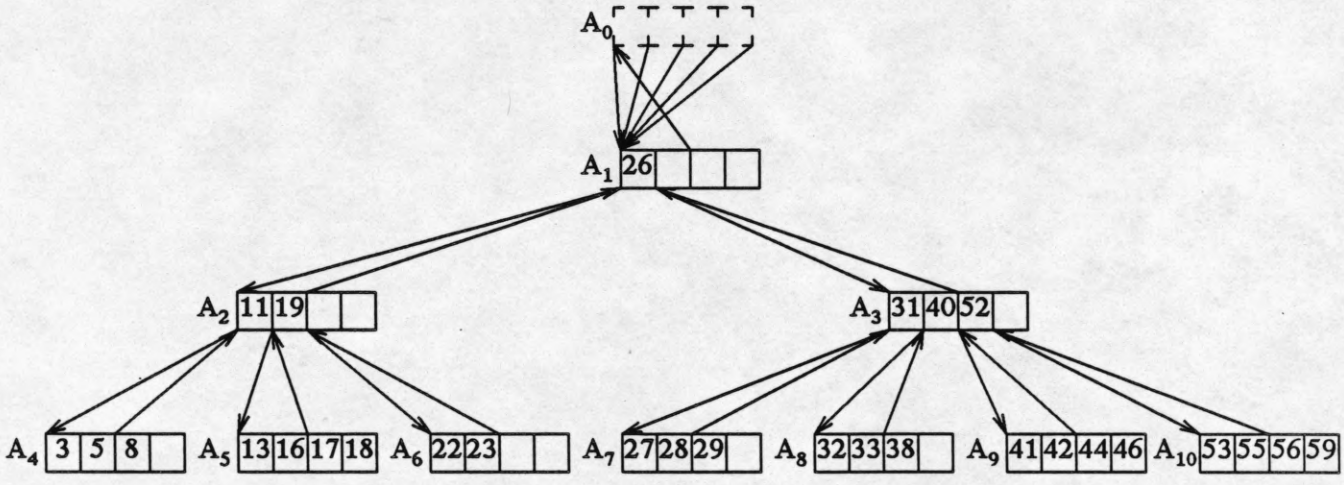

Figure 6c. Order-2 B-Tree with Virtual Backpointers (VBT).

Table 2. Analysis of Errors in the VBT.

Error Condition	d_*^2	d_*^3	d_*^s $\forall s \geq 4$
A non-empty VBT becomes empty	$2m+1$	$2m+1$	$2m+1$
An empty VBT becomes non-empty	$2m+2$	$2m+2$	$2m+2$
A key or key check symbol (X, Y) becomes erroneous	3	3	3
An internal node's non-null pointer becomes erroneous	2	3	3
An internal node's non-null pointer becomes null	6	6	6
An internal node's null pointer becomes non-null	6	7	7
Two of an internal node's pointers are exchanged	3	3	3
An internal node becomes a leaf node	3	3	3
A leaf node becomes an internal node	3	4	4

(1) $candidate_1 = P_{i,j}$
(2) $candidate_2 = V_i \oplus A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m}$
(3) **if** ($candidate_1 \neq candidate_2$) **then**
(4) Error detected
(5) **else**
(6) Access N_{MV} via $candidate_1$
(7) Set $N_i = N_{MV}$

Figure 7a. Detection Steps, VBT Forward Move.

the second candidate. If the candidates are unequal, then an error has been detected and the move may be incorrect. Otherwise, the move is correct.

To verify a backward move, one constructive vote ($V_i \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$) and one diagnostic vote ($P_{MV,r}$?= A_i) are evaluated (Figure 7b). If the diagnostic vote returns "False," then an error has been detected and the move may be incorrect. Otherwise, the move is correct. These detection steps, given in Figures 7a and 7b, access at most one node off the intended traversal path, and may be repeated for each pair of adjacent nodes in larger windows, just as for the VDLL.

The following theorem provides the correctability C^s(MV) for the VBT.

THEOREM 6: In the VBT, the correctability C^2(forward) = 0; C^s(forward) = 1 $\forall s > 2$ and at most $2m$ extra nodes are accessed. Also, C^s(backward) = 0, $\forall s > 2$.

(1) *candidate* = $V_i \oplus P_{i,0} \oplus P_{i,1} \cdots \oplus P_{i,2m}$
(2) Access N_{MV} via *candidate*
(3) **if** ($P_{MV,r} \neq A_i$) **then**
(4) Error detected
(5) **else**
(6) Set $N_i = N_{MV}$

Figure 7b. Detection Steps, VBT Backward Move.

PROOF: Since the detectability $D^2 = 1$, the correctability $C^2 = 0$. Similarly, the detectability $D^s = 2$ for $s > 2$, and the upper limit of correctability $C^s = 1$. Therefore, only single error cases in W_*^3 are considered. The error may be a key, a key check symbol, a count or a pointer. Key or key check symbol errors are diagnosed and corrected using the simple procedures for product codes [16]. Diagnosis and correction of an erroneous count is accomplished by counting the non-null keys. An erroneous pointer located in N_0 can be diagnosed and corrected by simple comparison, as there are $2m+1 \geq 3$ identical pointers in the header. For an erroneous pointer not in N_0, e.g., $P_{i,j}$ ($i \neq 0$), correction is performed after an error has been detected using the steps of Figure 7a, during a forward move $N_i \rightarrow N_{MV}$ following $P_{i,j}$. Then the principal component is $c = P_{i,j}$ and the correction steps in Figure 8 may be invoked.

Once these steps have completed, the detection steps given in Figure 7a can be reapplied to the original window. Assume an error is detected, i.e., the two constructive votes do not agree. This occurs if $P_{i,j} \neq V_i \oplus A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m}$; however, Step 9, 18 or 20 of the correction procedure guarantees that this condition will be satisfied (single error assumed). This is a contradiction, so the two constructive votes must agree. Therefore, the assumption that an error has been detected is false, and the correctness of the move is guaranteed. By Definition 5, the correctability C^s(forward) = 1.

For a backward move following $V_i \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$, the principal component is $c = V_i$, and there is only one constructive vote (V_i). However, to correct one error, there must be at least

(1) $candidate_1 = P_{i,j}$
(2) $candidate_2 = V_i \oplus A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m}$
(3) Access N_{MV} via $candidate_1$
(4) $val_1 = (V_{MV} \oplus P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \stackrel{?}{=} A_{i,j})$
(5) Access N_{MV}' via $candidate_2$
(6) $val_2 = (V_{MV}' \oplus P_{MV,0}' \oplus P_{MV,1}' \oplus \cdots \oplus P_{MV,2m}' \stackrel{?}{=} A_{i,j})$
(7) **if** (val_1 = **false**) **then**
(8) **if** (val_2 = **true**) **then**
(9) Set $P_{i,j} = V_i \oplus A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m}$
(10) **else**
(11) Uncorrectable: multiple errors
(12) **else**
(13) **if** (val_2 = **false**) **then**
(14) **for** k = 1 to $2m$, k $\neq$ j **do**
(15) Access N_x via $P_{i,k}$
(16) Perform detection on N_i and N_x
(17) **if** (error detected) **then**
(18) Set $P_{i,k} = V_i \oplus A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,k-1} \oplus \cdots \oplus P_{i,k+1} \oplus \cdots \oplus P_{i,2m}$
(19) **if** (no errors detected in **for** loop) **then**
(20) Set $V_i = A_{prev,r} \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$
(21) **else**
(22) No error

Figure 8. Correction Steps, VBT Forward Move.

two constructive votes. Hence, the correctability C^s(backward) = 0. □

The robust B-tree [4] presented by Black, Taylor and Morgan performs double error detection or single error correction using global techniques 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, 11] which is 2-local-detectable and 1-local-correctable. Each node of an order-m LB-tree is augmented by $2m$+5 extra fields. In comparison, the properties of the VBT are as follows:

1) Double component errors can be detected in the VBT using W_*^3.

2) Single component errors can be corrected in constant time in the VBT using W_*^3 for an error detected during a forward move.

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

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

V. EXPERIMENTAL RESULTS

This section describes a series of four experiments that provide example measurements of the capabilities (behavior under multiple error conditions) and performance (time overhead) of both local concurrent error detection and correction for example VDLL and VBT instances. A model virtual-memory database of VDLL and VBT instances was created and accessed by simulated database routines. Written in C, the implementations were run on Sun 3/50 and Sun 3/110 workstations.

For the capability experiments, VDLLs of 50, 100, 500 and 1000 nodes and VBTs of 100, 1000, 10,000 and 50,000 nodes were constructed. Varying numbers of errors were injected into the instance, ranging from single component errors to a maximum number of component errors equal to 30% of the nodes in the instance. Each error replaced a random component value in a random node with a random integer, chosen from the address space of the nodes of the instance. In the VDLL, values replaced were either pointers or virtual backpointers, and in the VBT, any one of: pointer, virtual backpointer, key, key check symbol or count value. Once the errors had been injected, a full traversal with local concurrent error detection (correction) was performed. The experiments determined the percentages of detectable and correctable errors and the number of fully recovered instances. It should be noted that a random distribution of errors may not accurately reflect the wide variety of structural errors possible, due to such problems as physical failures or corrupted software.

In the performance experiments, VDLL and VBT instances of 100, 1000, 10,000 and 50,000 nodes were constructed, to characterize the time overhead required for local concurrent error

detection and correction. The experiments assumed worst-case user behavior, to estimate the worst-case performance penalties. Only searches were performed, no data was read, no user computation was performed between database operations, and no disk accesses were required. In realistic applications of the VDLL and VBT database environments, the inclusion of delays due to disk accesses and user computations would dramatically reduce the performance overhead measured. The analysis of variance method [17] was used to determine that the performance was independent of the database size. Linear regressions were then used to estimate the time overhead of the algorithms. The results of the experiments are presented in the following four subsections.

A. Concurrent Error Detection Capability

VDLL Results

The following quantities were measured for each experiment: the total numbers of injected and corrected errors, whether the first erroneous node on the forward traversal path (FENF) had been detected, and whether the first erroneous node on the backward traversal path (FENB) had been detected. For all cases, 100% of the errors injected, 100% of the FENFs, and 100% of the FENBs were detected. The procedure performed perfect detection even under the multiple error conditions of this experiment.

VBT Results

The total numbers of injected, detected, and undetected errors were measured for each experiment. For all cases, 100% of the errors injected and 100% of the FENFs were detected. (There are no backward full traversals in the VBT, therefore no FENB measurement was done.)

B. Concurrent Error Correction Capability

VDLL Results

The results are shown in Figure 9. Each graph is associated with the VDLLs of a particular size. The x-axis in all graphs displays the number of errors injected, as a percentage of the number of nodes in the VDLL. The y-axis displays the average percentages of the errors that were corrected and the VDLLs that were recovered.

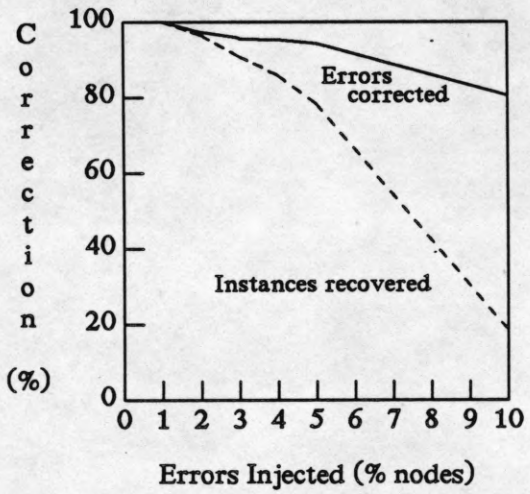

Figure 9a. 50-node VDLL.

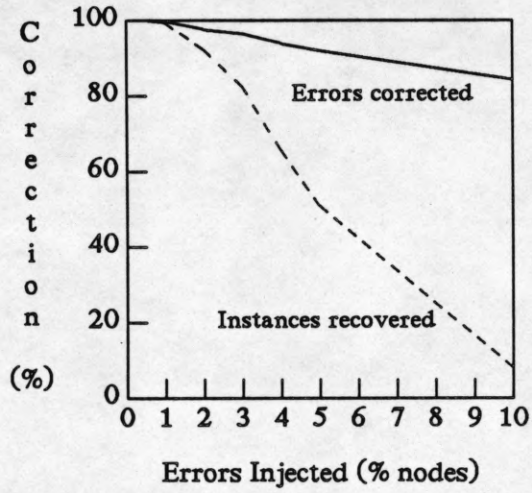

Figure 9b. 100-node VDLL.

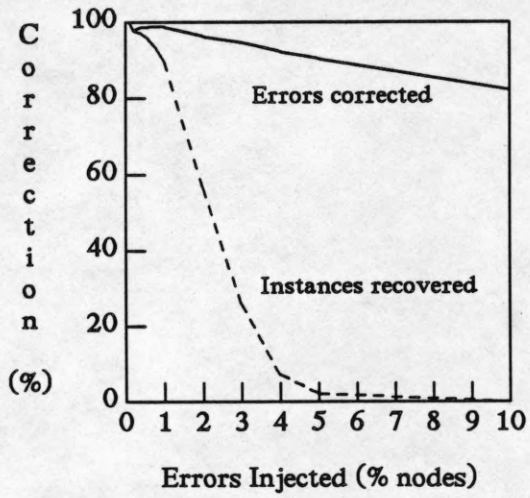

Figure 9c. 500-node VDLL.

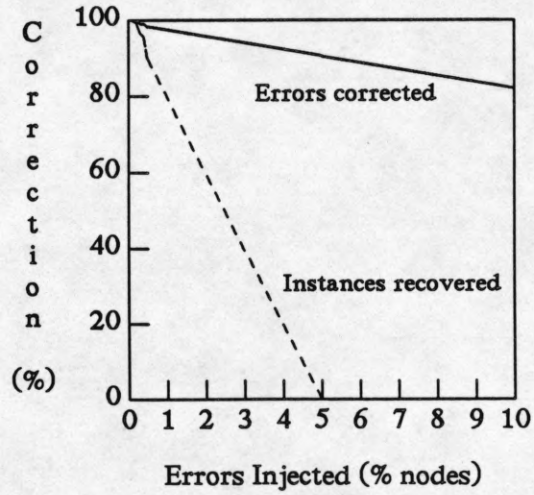

Figure 9d. 1000-node VDLL.

The graphs show that the procedure can consistently correct a significant percentage of the errors even when up to 10% of the instance's nodes may be corrupted. The instances-recovered curves drop off more sharply since even a single incorrect correction renders the instance unrecoverable. This also explains why the slope is steeper for the larger instances: even a small percentage of erroneous components translates into a large absolute number of errors, only one of which would need to be uncorrectable to render the structure unrecoverable.

VBT Results

As in the VDLL graphs, each graph of Figure 10 is associated with the VBTs of a particular size. The x-axis displays the number errors injected, as a percentage of the number of nodes in the VBT. The y-axis displays the average percentage of instances that were recovered.

The number of recoverable instances may be much greater than indicated in the graphs, depending on the timing and pattern of error formation. That is, an instance may still be recoverable if many errors arise but at such times that only one error is encountered by a checking window at any time. In practical applications, only a small number of errors would be expected in a data structure instance, so that the probability of recovery using this correction procedure would be very close to 100%.

C. Concurrent Error Detection Performance

VDLL Results

The time to perform searches without (TIME0) and with (TIME) local concurrent error detection were measured. An analysis of variance (ANOVA) procedure [18] first determined that OVERHEAD = TIME/TIME0 − 1 was independent of the database size, for our experiments. Next, a General Linear Models (GLM) procedure [18] determined the average value of OVERHEAD. It follows from the above definition of OVERHEAD that TIME = (1 + OVERHEAD) × TIME0. The following linear relationship between TIME and TIME0 was found to have $R^2 = 0.9998$, where $0 \leqslant R^2 \leqslant 1$, and $R^2 = 1$ indicates a perfect linear relationship between the quantities measured.

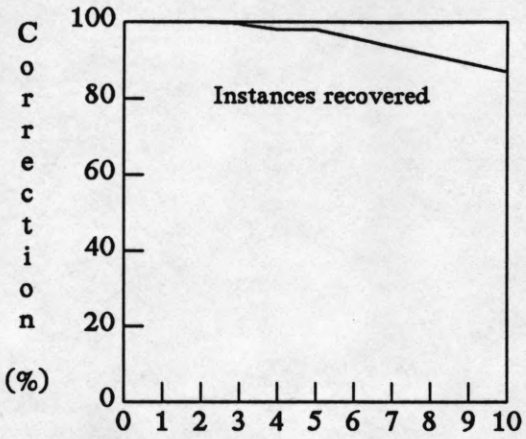

Figure 10a. 100-node VBT.

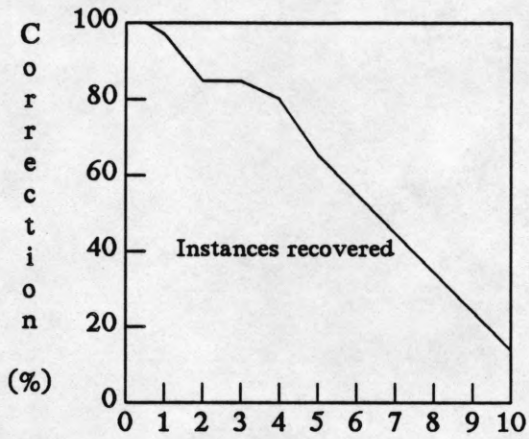

Figure 10b. 1000-node VBT.

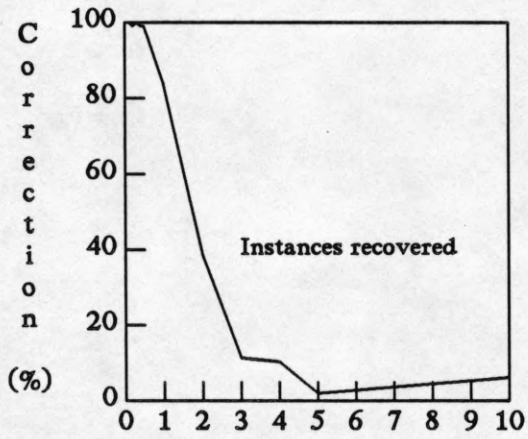

Figure 10c. 10000-node VBT.

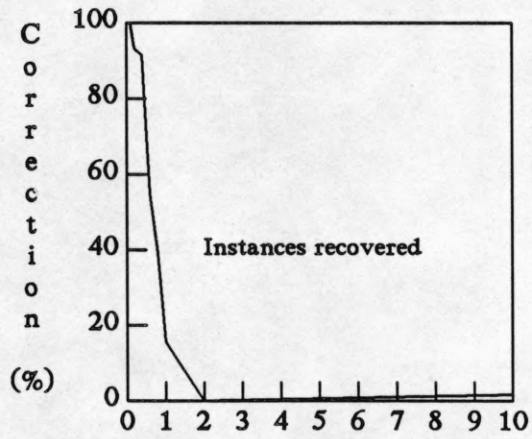

Figure 10d. 50000-node VBT.

$$\text{TIME (milliseconds)} = (1 + 1.01) \times \text{TIME0} + (-2.70).$$

The equation shows that the overhead of performing local concurrent error detection in the VDLL is approximately 100%. (The intercept value is at most 0.7% of the TIME0 value, and is insignificant when compared to the overhead of 100%.) This is an extremely conservative measure of the overhead, since it represents only the search computation overhead: actual database searches would involve disk activity, the magnitude of which would significantly reduce the detection over-

head. An experiment to support this was performed for the VBT (see below).

VBT Results

ANOVA determined that the variation in OVERHEAD = TIME/TIME0 − 1 was independent of the database size. GLM determined the following linear relationship between TIME and TIME0, with $R^2 = 0.9844$:

$$\text{TIME (milliseconds)} = (1 + 4.43) \times \text{TIME0} + (-181.58).$$

The equation shows that the overhead of performing local concurrent error detection in the VBT is less than 450%. (The intercept value represents at most 51% of the TIME0 value, and so is less significant than the overhead of 450%.) The overhead for the VBT is greater than that for the VDLL, since more checking is involved at each move. However, this represents only the search computation overhead. By comparison, the results of a VBT experiment which incorporated disk activity, but involved no user computations between database operations, gave the OVERHEAD as less than 10%.

D. Concurrent Error Correction Performance

VDLL Results

The time to perform 2000 traversals with local concurrent error detection (TIME0) and with both local concurrent error detection and correction (TIME) were measured. ANOVA showed that OVERHEAD = TIME/TIME0 − 1 was independent of the database size. GLM gave the following linear relationship between TIME and TIME0, where $R^2 = 0.9996$:

$$\text{TIME (milliseconds)} = (1 + 0.00) \times \text{TIME0} + (639.66).$$

The intercept represents the constant overhead incurred by the constant number of corrections performed (10 correction per traversal, for a total of 20,000 for each experiment). The time per correction is 0.8% of an average complete traversal in a 100-node data structure instance, showing

the low overhead involved in local concurrent error correction in the VDLL.

VBT Results

The time to perform 2000 traversals with local concurrent error detection (TIME0) and with both local concurrent error detection and correction (TIME) were measured. ANOVA showed that OVERHEAD = TIME/TIME0 − 1 was independent of the database size. GLM gave the following linear relationship between TIME and TIME0, where $R^2 = 0.9991$:

$$\text{TIME (milliseconds)} = (1 + 0.05) \times \text{TIME0} + (888.94).$$

The intercept represents the time required for 2000 corrections, and the value 0.05 represents the database-size dependent variable overhead, incurred by the slightly different detection procedures used with and without correction. The time penalty for using local concurrent error correction is about 5% of the time for a complete traversal of a data structure instance. This shows the low overhead for local correction in the VBT.

VI. SUMMARY

In this paper we have presented a new technique for local concurrent error detection and correction of structural errors in linked data structures that is applicable to a variety of data structures. The virtual backpointer uses the concept of a checking window to define the locality in which local concurrent error detection is performed and also to specifically state the size of that locality. The virtual backpointer was introduced and used to define and implement two new data structures, the Virtual Double-Linked List, which requires the same storage as the standard double-linked list, and the B-Tree with Virtual Backpointers of order m, which requires m+4 extra fields per node. It was shown that double errors are detectable using a local error detection procedure in constant time for both structures. In addition, single errors detected during forward

moves were shown to be correctable in constant time.

REFERENCES

[1] D. Taylor, D. Morgan, and J. Black, "Redundancy in data structures: Improving software fault tolerance," *IEEE Trans. Software Engineering*, vol. SE-6, no. 6, pp. 585-594, Nov. 1980.

[2] D. Taylor, D. Morgan, and J. Black, "Redundancy in data structures: Some theoretical results," *IEEE Trans. Software Engineering*, vol. SE-6, no. 6, pp. 595-602, Nov. 1980.

[3] D. Taylor, D. Morgan, and J. Black, "A compendium of robust data structures," *Proc. 11th Int. Symp. Fault-Tolerant Computing*, pp. 129-131, June 1981.

[4] J. Black, D. Taylor, and D. Morgan, "A robust B-tree implementation," *Proc. 5th Int'l Conf. Software Engineering*, pp. 63-70, Mar. 1981.

[5] J. Munro and P. Poblete, "Fault tolerance and storage reduction in binary search trees," *Information and Control*, vol. 62, pp. 210-218, 1984.

[6] M. Sampaio and J. Sauvé, "Robust trees," *Proc. 15th Int. Symp. Fault-Tolerant Computing*, pp. 23-28, June 1985.

[7] S. Seth and R. Muralidhar, "Analysis and design of robust data structures," *Proc. 15th Int. Symp. Fault-Tolerant Computing*, pp. 14-19, June 1985.

[8] K. Yoshihara, Y. Koga, and T. Ishihara, "A robust data structure scheme with checking loops," *Proc. 13th Int. Symp. Fault-Tolerant Computing*, pp. 241-248, June 1983.

[9] K. Kuspert, "Efficient error detection techniques for hash tables in database systems," *Proc. 14th Int. Symp. Fault-Tolerant Computing*, pp. 198-203, June 1984.

[10] J. Black and D. Taylor, "Local correctability in robust storage structures," to appear: *IEEE Trans. Software Engineering*.

[11] D. Taylor and J. Black, "A locally correctable B-tree implementation," *The Computer Journal*, vol. 29, no. 3, pp. 269-276, 1986.

[12] I. Davis and D. Taylor, "Local correction of mod(k) lists," CS-85-55, Dept. of Computer Science, University of Waterloo, Dec. 1985.

[13] I. Davis, "Local correction of helix(k) lists," CS-86-30, Dept. of Computer Science, University of Waterloo, Aug. 1986.

[14] I. Davis, "A locally correctable AVL tree," *Proc. 17th Int. Symp. Fault-Tolerant Computing*, pp. 85-88, July 1987.

[15] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173-189, 1972.

[16] P. Elias, "Error-free coding," *IRE Trans. Information Theory*, vol. IT-4, pp. 29-37, 1954.

[17] D. Ferrari, *Computer systems performance evaluation*. Englewood Cliffs: Prentice Hall, Inc., 1978.

[18] *SAS user's guide: Statistics*. Cary, NC: SAS Institute, Inc., 1985. pp. 113-138, 433-506.