A DISTRIBUTED, REPLICATED, DATA-BALANCED SEARCH STRUCTURE

THEODORE JOHNSON
University of Florida, Dept. of CIS
Gainesville, Fl 32611-2024

ADRIAN COLBROOK
MIT Laboratory for Computer Science
Cambridge, MA 02139

ABSTRACT

Many concurrent dictionary data structures have been proposed, but usually in the context of shared memory multiprocessors. In this paper, we present an algorithm for a concurrent distributed B-tree that can be implemented on message passing computer systems. Our distributed B-tree (the *dB-tree*) replicates the interior nodes in order to improve parallelism and reduce message passing. The dB-tree stores some redundant information in its nodes to permit the use of lazy updates to maintain replica coherency. We show how the dB-tree algorithm can be used to build an efficient implementation of a highly parallel, data-balanced distributed dictionary, the *dE-tree*.

Keywords: Concurrent dictionary data structures, Message passing multiprocessor systems, Balanced search trees, B-link trees, Replica coherency.

1. Introduction. We introduce a new balanced search tree algorithm for distributed memory architectures. The search tree uses the B-link tree [27] as a base, and distributes ownership of the nodes among the processors that maintain the tree. We also *replicate* the non-leaf nodes, to improve parallelism. If we apply a read-one, write-all consistency maintenance rule [44], then reading a node becomes cheaper and writing to a node becomes more expensive as we increase the degree of replication. We observe that a node

close to the root is often read, but rarely written to, but a node close to a leaf is rarely read, but is (relatively) often written to. Therefore, our ownership rule is that if a processor owns a leaf, it owns a copy of all nodes from the root to the leaf. The root is replicated at every node, so operations can be initiated in a fully parallel manner, while the often-modified nodes near the leaf are only moderately replicated, reducing the cost of their maintenance. Restructuring in a B-tree occurs on the path from the root to the leaf. A processor that restructures a node also has a local copy of the parent, so that restructuring decisions can be made locally, eliminating the need for centralized control and permitting further parallelism.

If the keys of a distributed data structure are arbitrarily or statically allocated to the processors, then some processors might be required to store a disproportionate share of the keys. Such an imbalance might cause a processor to exhaust its storage space even though other processors can store many more keys. A distributed search structure needs to be able to perform *data balancing* in order to efficiently use the storage that is available in the system. We show how data balancing can be implemented using the flexible distributed B-link tree algorithms.

1.1. Search Trees. Search trees are widely used for fast implementations of dictionary abstract data types. A dictionary is a partial mapping from keys to data that supports three operations: *insert*, *delete* and *search*. For simplicity we will assume that the dictionary stores no data with the keys and so may be viewed as a set of keys. A number of useful computations can be implemented in terms of dictionary abstract data types, including symbol tables, priority queues and pattern matching systems.

The B-tree was originally introduced by Bayer [1]. The B-tree algorithms for sequential applications were designed to minimize the response time for a single query and the sequential algorithm for a single search operation on a balanced B-tree has logarithmic complexity. The improvement in the response time that may be achieved by a parallel algorithm for a single search can at best be logarithmic in the number of processors used [37]. Therefore, for parallel systems a more important concern is increasing system throughput for a series of search, insertion and deletion operations executing in parallel.

The large number of concurrent search tree algorithms presented in the literature [27, 2, 31, 11, 24, 26, 41, 33, 39, 25, 42, 4, 32, 45, 49] prevents a complete description of each in this paper. Instead, we discuss the common issues that are addressed by all of the algorithms.

All concurrent search tree algorithms share the problem of managing contention. Concurrency control is required to ensure that two or more independent processes accessing a B-tree do not interfere with each other.

A common approach is to associate a read/write lock with every node in the search tree [42]. This causes data contention as writers block incoming writers and readers, and readers block incoming writers. The contention is severe when it occurs at higher levels in the search tree, particularly at the root (often termed a *root bottleneck* [22, 19]). Similar problems are caused by resource contention. In a shared-memory architecture all of the processes trying to access the same tree node will access the same memory module on the machine. Similarly, for message-passing architectures, the processor on which a node resides will receive messages from every processor trying to access the node. Resource contention is again most serious for the higher levels in the search tree. Node replication [45] reduces contention but requires a coherence protocol to maintain consistency. The redundant information stored in dB-tree nodes allows the use of lazy update algorithms to maintain consistency.

Associated with the contention issue is the problem of *process overtaking*. This may occur when a process that holds a lock selects the next node it wishes to access, releases its lock and attempts to acquire a lock for the next node. A second process may acquire a lock on the next node between the original process releasing the old lock and acquiring the lock for the next node. The second process can then update the node in such a fashion as to cause the first process to lock the wrong node when it eventually acquires the lock. To prevent this kind of process overtaking many algorithms have their operations use *lock coupling* to block independent operations [33, 17]. An operation traverses the tree by obtaining the appropriate lock on the child before releasing the lock it holds on the parent. B-link trees [27, 39, 25] eliminate the need for lock coupling. If the wrong node is reached at any stage the operation is able to find the correct node. This reduces the number of locks that must be held concurrently and increases throughput. We use the B-link tree as a base for the dB-tree.

1.2. Previous Work. Wang and Weihl [45, 49] have proposed that parallel B-trees be stored using *Multi-version Memory*, a special cache coherence algorithm for linked data structures. Multi-version Memory permits only a single update to occur on a replicated node at any point in time (analogous to *value logging* [47, 3] in transaction systems). Our algorithm permits concurrent updates on replicated nodes (analogous to *transition logging* [47, 3]).

Ellis [12] has proposed algorithms for a distributed hash table. The directories of the table are replicated among several sites, and the data buckets are distributed among several sites. The hash table is a shallow search structure, so every update to the index structure must be distributed to every copy of the index. The distribution of updates can be limited in a

distributed B-tree because it is a multilevel index structure. Also, the B-link tree is a very flexible data structure in which more sophisticated operations, such as range queries, can be easily implemented. Yen and Bastani [50] have developed algorithms for implementing a hash table on a SIMD parallel computer, such as a CM2. The authors examine the use of chaining, linear probing, and double hashing to handle bucket overflows.

Peleg [16, 36] has proposed several structures for implementing a distributed dictionary. The concern of these papers is the message complexity of access and data balancing. However, the issues of efficiency and concurrent access are not addressed. Matsliach and Shmueli [30] propose methods for distributing search structures in a way that has a high space utilization. The authors assume that the index is stored in shared memory, however, and don't address issues of concurrent restructuring.

Deitzfelbinger and Meyer auf der Hyde [9] give algorithms for implementing a hash table on a synchronous network. Ranade [38] gives algorithms and performance bounds for implementing a search tree in a synchronous butterfly or mesh network.

Colbrook et al. [8] have proposed a pipelined distributed B-tree, where each level of the tree is maintained by a different processor. The parallelism that can be obtained from this implementation is limited by the number of levels in the tree, and the distributed tree is not data-balanced.

The contribution of this paper is to present a highly parallel distributed B-tree (the *dB-tree*) that permits concurrent updates on a replicated tree node, and rarely blocks operations. We show how our distributed B-tree can be used to implement an efficient, highly parallel, data balanced distributed dictionary (the *dE-tree*). In Section 2 we describe the dB-tree. In Section 3 we show how the dE-tree can be built from the dB-tree. Finally, conclusions are drawn in Section 4.

2. The dB-tree, a Concurrent Distributed B-tree.

2.1. Concurrent B-link tree Algorithms. As a base for our distributed B-tree, we use the concurrent B-link tree [27, 39, 25]. The B-link tree algorithms have been found to have the highest performance among all existing concurrent B-tree algorithms [22, 19, 43]. Restructuring operations on B-link trees are performed one node at a time, so that the algorithms can be easily translated to a distributed environment. In a B-link tree, every node contains a pointer to its right neighbor. In the concurrent B-link tree algorithm, each node also contains a field, `highest`, which is the highest valued key that can be stored in the subtree rooted at that node. The B-link tree algorithms use the additional information stored in the nodes to let an operation recover if it misnavigates in the tree due to out-of-date

information.

The template for the concurrent B-link tree algorithm described by Sagiv [39] is shown as Code 1. Search operations start by placing an R (read) lock on the root and then searching the root to determine the next node to access. The search operation then unlocks the root and places an R lock on the next node. The search operation continues accessing the interior nodes in this manner until it reaches the leaf that could contain the key that it is searching for. If the operation reads a node and finds that the `highest` field is lower than the key it is searching for, it follows the right pointer of the node. When the search operation reaches the leaf that would contain the key that it is searching for, it searches the leaf for the key and returns success or failure depending on whether the key is or is not in the leaf.

```
link_locate_and_decisive(key:  x)
   var
      node:  n,next
   n := root
   loop  /* invariant:  x is in inreach(n) */
      lock(n)
      next := find(x,n)
      if next is nil,
         exit from loop
      unlock(n)
      n := next
   end loop
   /* x is in keyset(n) */
   dec(o)(x,n)
   restructure()
```

CODE 1. *Concurrent B-link tree algorithm.*

An insert operation works in two phases: A search phase and a restructuring phase. The search phase on the insert operation uses the same algorithm as the search operation, except that it places W (exclusive write) locks on the leaf nodes. When the insert operation reaches the appropriate leaf, it inserts its key if the key is not already in the leaf. If the leaf is now too full, the insert operation must split the leaf and restructure the tree, as in the usual B-tree algorithm. The operations hold at most one lock at a time, so they must break the restructuring into disjoint pieces. So, when an insert operation finds that it must restructure the tree it performs a *half-split* operation (see Figure 1). A half split operation consists of creating a new node (the sibling), transferring half of the node's keys to the sibling, and linking the sibling into the leaf list. In order to complete the split, the insert operation releases the lock on the node, locks the parent, and inserts a pointer to the sibling. If the parent becomes too full, the insert operation applies the same restructuring steps. Notice that for a period of time, there

is no pointer in the parent to the sibling. Operations can still reach the sibling because of the right pointers and `highest` field.

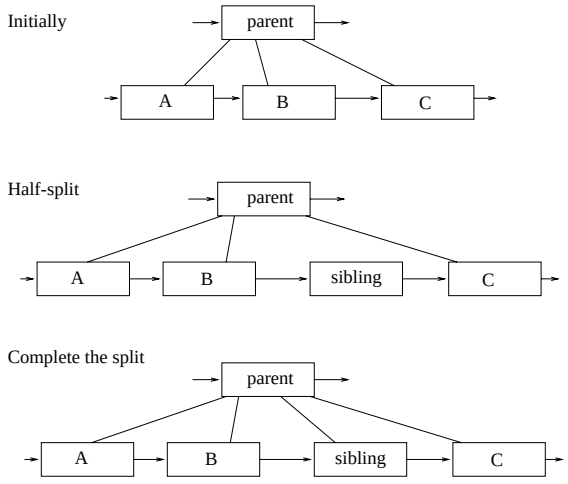

FIG. 1. *Half-split operation*

The B-link tree in a shared memory multiprocessor does not easily support the deletion of nodes. Lehman and Yao [27] recommend that nodes are never deleted. Sagiv [39] describes garbage collection algorithms for underfull nodes. Lanin and Shasha [25], and Wang [45] describe an algorithm that leaves stubs in place of deleted nodes. We show how nodes can safely be removed from the tree due to the explicit naming of shared nodes.

2.2. The dB-tree. Our distributed B-tree algorithm builds on the concurrent B-link algorithms. An example dB-tree (distributed B-tree) is shown in Figure 2. The leaves of the dB-tree are distributed among four processors. The interior nodes of the dB-tree are replicated among several processors. The rule for replicating the interior is that if a processor owns a leaf then it owns a copy of every node from the path from the root to the leaf, inclusive. Each processor on a level has links to both neighbors, not just the right neighbor. Keeping the nodes on a level in a double linked list simplifies replication control and the deletion of nodes (as will be discussed later), and also aids in downwards range queries. In addition, each node stores its distance from the leaves.

The nodes of the dB-tree are distributed among the processors, which do not share a common memory space. Instead of naming the nodes with address, we will name them with *tags*. Each node has a unique tag. These tags can be generated by requiring each processor to keep a count of the objects that it creates. When a processor creates a node, it names it with

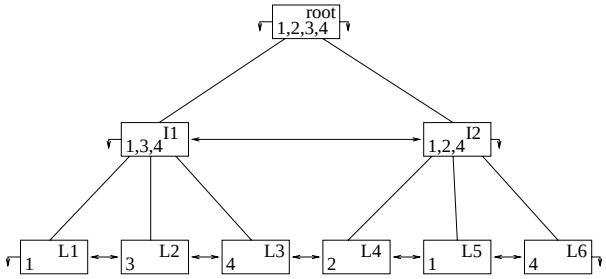

FIG. 2. *A dB-tree*

the counter concatenated with the processor id. A pointer in a dB-tree node is a node tag together with a list of the processors that own a copy of the node. In order to access a node, the processor translates the tag to a local address through a translation table (organized as a hash table, for example).

The dB-tree has three *operations* defined on it: search(v), insert(v), and delete(v). These operations have the usual semantics. An operation on the dB-tree performs *actions* on the tree nodes in order to perform its computation. The actions are performed atomically by a processor on the processor's local data. Each processor has a queue of pending actions that need to be performed on its local nodes. A processor that helps to maintain the dB-tree accepts operations from other processors and performs the operation's suboperations. Conversely, a processor can send an operation to a different processor. Thus, each processor executes Code 2.

```
while(1)
   get task=(node,action,parameters) from the work queue
   local_node=lookup(node)
   if(node not found)
      execute recovery actions
   else
      execute action on local_node
```

CODE 2. *Main loop of the node manager.*

If a processor does not store the node that the action is requesting, the action has misnavigated and must recover from its error. This case can occur if the processor once stored the node, but later deleted or transferred it. We later discuss the possibilities for this type of misnavigation.

A search operation that originates at one of the participating processors starts by performing a *node-search* action on the locally stored copy of the root. A search operation that originates at a processor that does not help

maintain the dB-tree must transmit its request to one of the participating processors. The node-search action is completely local: the node can be read in parallel at every processor that owns a copy of the node, and a node-search at one processor does not block an update of the same node at another processor. The node-search action determines the next node to access, and is essentially one step of the algorithm in Code 1. A copy of the next node on the search path may be stored locally, in which case the local copy should be used, or all copies may be stored remotely, in which case the processor must send the operation to a processor that stores a copy of the node.

```
node_search(local_node,v,action)
   if(v is in the range of local_node)
      if(local_node is a leaf or level(local_node)=i
            and the action is for level i)
         execute action on local_node
      else
         locate child
         perform local_search(child,v,action)
   else
      locate neighbor
      perform local_search(neighbor,v,action)
```

CODE 3. *Node-search action.*

A `node_search` action is used as the basic mechanism for ensuring that an action is performed on the correct node (as in the case of Sagiv's algorithm [39]). Thus, `node_search` is written so that it can deliver an action to a non-leaf node.

As an example, if a search operation for a key stored in $L3$ originates in processor 3, then the operation reads processor 3's copy of the root and $I1$, then transmits a request to read $L3$ at processor 4. If a processor has a choice of sites to send the search operation to, it can make a choice based any reasonable criteria, such as locality or estimated load. For example, if a search request for a key stored at $L5$ originates at processor 3, processor 3 chooses between processors 1, 2, and 4 to service the request to search node $I2$. When the search operation reaches the correct leaf, it returns a `success` message to the originating processor if the key is in the leaf, and a `failure` message otherwise.

If an insert or delete operation does not cause restructuring, then it is executed in the same way as a search operation, except that the action on the leaf is to insert or delete a key. The insert and delete actions can be applied to inserting keys into a leaf and to inserting pointers to new children into a parent. The insert and delete actions are specified Code 4 and 5.

```
insert(local_node,v,(pointer))
   if(no deletion is recorded for (v,pointer)
      insert key (and pointer) into local_node
      update replicated copies of local_node
      if(local_node is too full)
         perform half-split(local_node)
   else
      remove the record of the deletion.
```

CODE 4. *Insert algorithm.*

```
delete(local_node,v)
   if( key is in node)
      delete key from local_node
      update replicated copies of local_node
   else if ( local_node is not a leaf)
      record deletion on node
```

CODE 5. *Delete algorithm.*

Insert and delete actions are performed asynchronously. As a result, an action for the insertion of a child might arrive at a parent after the action that deletes the link arrives. If a delete action arrives and there is no corresponding link to delete, the delete is recorded and cancels the corresponding insert.

If an insert causes a node to become too full, it triggers a half-split action. We allow the implementation to choose the point at which node merges are triggered (for example, merge-at-empty causes little loss in space utilization [21], but greatly simplifies the implementation).

2.3. dB-tree Half-splits and Half-merges. The use of double links requires that splitting a node be carried out in three steps instead of two. The half-split action is shown in Code 6, and is illustrated in Figure 3. When node B becomes too full, it splits into B and *Sibling*, each taking half of the keys that were stored in B. The *Sibling* node can be stored locally or transferred to a different processor or set of processors. The next step is to make the leaf list consistent. Finally, a pointer to *Sibling* must be inserted into *parent*. The split procedure is carried out by performing actions on the nodes. First, a half-split action is performed on node B. Next, a link-change action is sent to node C, and insert action is sent to *parent*. Note that there might be several copies of B, *Sibling* and *parent*. The actions on the sibling and the parent can be performed concurrently, and the half-split action does not block waiting for their completion. Instead, an action is sent to the responsible processors. Also note that the parent might have been split or merged, so that several search actions might have to be performed before the insert or link-change actions can be performed. We will assume for now that

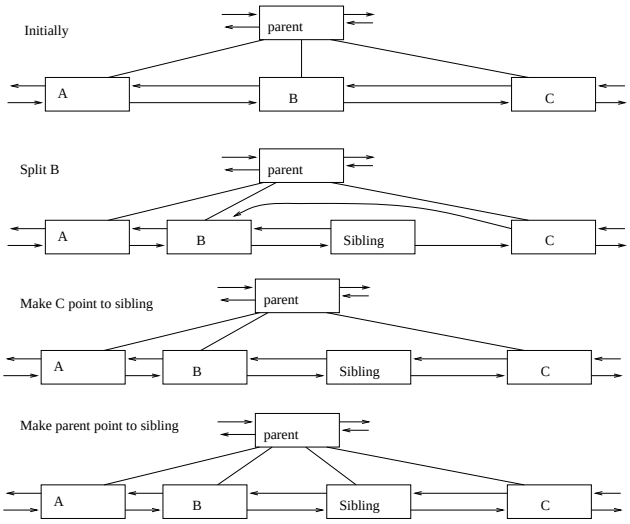

FIG. 3. *A dB-tree half-split*

modification suboperations are performed on all copies of a node atomically.

```
half_split(local_node)
   create sibling
   transfer half of the keys from local_node to sibling
   make link point to sibling
   add sibling to incoming_link list
   perform link_change(local_node → sibling) on neighbor
   perform insert(sibling) on parent
```

CODE 6. *Half-split action.*

After a node is merged into another, or becomes empty, it must be removed from the tree. The half-merge procedure is shown in Code 7, and is illustrated in Figure 4. First, the node that is to be deleted, B, must contact one of its neighbors (A) to transfer its key range, and any keys that it contains. When the transfer is completed, B marks itself as deleted, and indicates that node A contains its keyspace. In addition, node A performs a link change so that it points to B's other sibling. Node B then sends a link-change action to its other sibling, C, and a delete action to its parent.

While one node splits or merges, its neighbors might concurrently split or merge. Therefore, the node that should perform the link-change action must be identified by its upper (or lower) key space range rather than by name. Furthermore, the range boundary might be deleted due to a merge. In this case, the link-change is no longer applicable, so it should be ignored.

In order to ensure that every node on a level of the index can reach every

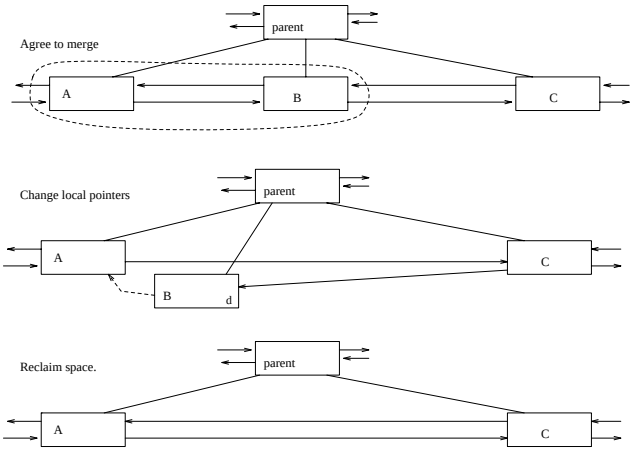

FIG. 4. *A dB-tree half-merge*

other node on the same level, we require that all links to a merged node from a node on the same level must be removed before the merged node can be deleted. Every node keeps a list of the nodes that might contain a left link to it. The left and right neighbor of a node n contain links to n, or will soon perform a link-change action to do so. In addition, nodes to the left of n might have not yet completed the link-change action due to a split, and previously deleted nodes might still remain in the structure (see Figure 5). When a node splits, it adds the name of the new sibling to the *incoming-link* list, and the sibling receives the name of the right neighbor. When a node agrees to a half-merge, it puts the deleted node on its incoming-link list. When a link-change action is performed, the node that performs the action acknowledges the link-change. A deleted node informs the node it merged with when its space is reclaimed.

If a node merges with its left neighbor, it must wait until all nodes on its incoming-link list have acknowledged the link change before reclaiming its space. If a node merges with its right neighbor, it must also wait for an acknowledgement from its left neighbor. If a node receives a right-link change action from a node on its incoming-link list, it deletes that node from the list (and possibly records the new right neighbor in the left link list).

```
half_merge(local_node)
   contact neighbor, agree to merge
   transfer keys and keyspace
   mark local_node as deleted, drain requests to neighbor
   perform link_change(local_node → neighbor) on other neighbor
   perform delete(local_node) on parent
   when all nodes on incoming-link list have acknowledged
```

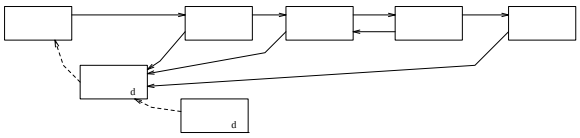

FIG. 5. *Old links to a deleted node.*

```
        a link-change or a deletion
    inform neighbor of deletion
    reclaim space
```

CODE 7. *Half-merge action.*

```
accept_half_merge(local_node, neighbor)
   if a half-merge is acceptable,
      agree to the half-merge
      accept keys and keyspace
      update link pointer
      add neighbor to incoming-link list
```

CODE 8. *Accepting a merge request.*

If the link-change actions are performed in the order they are generated, then the doubly-linked list of nodes on a level will be correctly maintained. All requests for a link-change on a node are causally related (since they are generated by actions at the neighbor), so a numbering scheme can be used to enforce the ordering [23].

```
link_change(local_node → neighbor, boundary_value)
   if the boundary_value matches local_node's boundary,
      change the link, obeying the causal ordering
   acknowledge the link-change
```

CODE 9. *Link-change action.*

When the parent of a deleted node receives a deletion from a boundary child, it might find that its keyspace has shrunk. For an example, see Figure 6. In this case, the boundary values of the parents must be modified through a link-change action. For a period of time, there will be an unclaimed key range in the parent's list. If messages are received in the order sent, however, this will cause an action to be misrouted only once. In addition, the grandparent must be informed of the new boundary.

2.4. Misrouted Actions. A node is deleted without guarantees that no action will attempt to access it in the future. In particular, actions might arrive from the parent or the children of the deleted node. As a result, a request to perform an action on a node might arrive at a processor that does

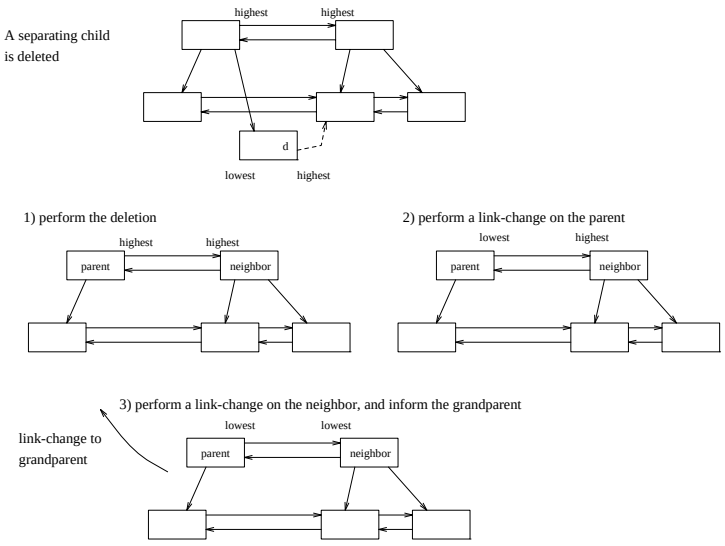

FIG. 6. *Shifting the key range in the parent.*

not store that node. The processor must find a node in the dB-tree to submit the action. The node to submit the action on can be the node that sent the action, another node in the same level, or the root. The node-search action will eventually deliver the misrouted action to the intended destination.

2.5. Integrating Concurrency Control with Replica Coherency. In the above discussion, we have assumed that modifications to nodes occur atomically. We can achieve the atomic updates by requiring that modifying actions lock every copy of the modified node before performing the update and block all reads and updates on the node, by using one of the well-known algorithms for managing replicated data [44, 15]. However, we can maintain our replicated nodes with far less synchronization and overhead. First, observe that it is not necessary to distribute the contents of the node on every modification, it is only necessary to distribute the modification itself. Second, the tree is never left in an incorrect state, so that search actions do not need to be blocked. Third, many of the modifying actions commute. Several authors have studied the issue of concurrency control on abstract data types when some operations may commute [18, 34, 48, 40, 46], but in the context of a transaction processing system. Our algorithms are similar to those described by Ellis [12] to maintain the replicated directories of the distributed hash tables in so that it is not always necessary that all actions are performed in the same order at all nodes. For example, insert actions may be performed out-of-order. In the example shown in Figure 7,

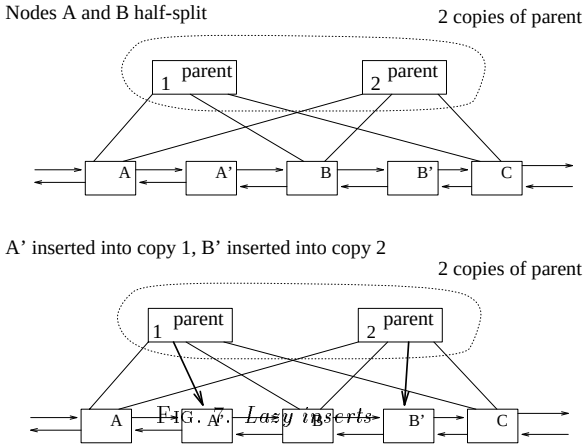

FIG. 7. Lazy inserts

nodes A and B have split, and pointers to the new nodes (A' and B') need to be inserted in the pointers to the parents. During the time from when a sibling is created to when a pointer to the sibling is inserted into the parent, the sibling can still be reached by the search, insert, and delete actions by the right and left pointers. So, if two inserts to the parent node are pending, it does not matter which is performed first, so that they can be performed in a different order in different copies of the parent node.

Not all actions on a node can be performed in an arbitrary order. Consider, as one example, two copies of a node that is full. There are two actions that are pending on the node, an insert and a delete. Suppose that the insert is performed before the delete at the first copy, and the delete is performed before the insert at the second copy. The first copy will decide to split the node, and the second copy will not. The problem is not that inserts and deletes must be ordered, rather the problem arises because the split action must be ordered with the insert and delete actions.

In order to argue for the correctness of the data structure, we need a correctness condition. The most natural correctness condition is that if no more operations are applied to the data structure, it will eventually evolve into a state such that all replicated copies of the nodes are the same, and their contents are the same as if the set of operations were applied to a single-copy data structure, sequentially in the equivalent serial order [41, 3].

Shasha and Goodman [41] describe proof schemas for concurrent search structures (as discussed later). For link-type data structures, if each action is atomic, maps a good state to good state (i.e, the structure can still be used after the action), and no action accesses a node after it is removed from

the search structure, then the algorithm is correct. To handle replication, we need two additional conditions: First, every copy of a node eventually receives the same set of actions. Second, non-commuting actions are performed in the same order at every copy. In addition, the replication algorithms should enforce any desired orderings on the actions, such as requiring that link changes be executed in the order requested.

We can categorize actions on nodes as being *lazy*, *semi-synchronous*, *synchronous* according to the amount of synchronization required to perform the action. A lazy action does not need to synchronize with other lazy actions. A semi-synchronous action must synchronize with some, but not all other actions. We classify a synchronous action as one that must be ordered with all other actions, or that requires communication with other nodes.

In [20], Johnson and Krishna present a framework for creating and analyzing lazy update algorithms. The framework is used to develop algorithms that can manage a dB-tree node. The algorithm uses lazy insert actions and semi-synchronous half-split actions. In addition, the algorithm framework accounts for *ordered actions*, to require that classes of actions are performed on a node in the order in which they are generated. For example, the link-change actions are ordered.

3. Correctness of the Operations. Shasha and Goodman [41] provide a framework for proving the correctness of non-replicated concurrent data structures. We make extensive use of their framework in order to discuss operation correctness.

An *abstract data type* consists of a *data structure*, D, and a set of *operations* on the data structure. In the case of a dictionary ADT, the operations are insert, delete, and search. An operation consists of a sequence of *actions* which carry out the operation.

An ADT operation takes the search structure from one state to another. For example, an insert operation adds its key to the search structure state. The first issue that we need to address is the meaning of the state of the search structure. The work that we base this section on [41] assumed that the search structure state is globally observable – an assumption that applies in the context of a multiprogrammed uniprocessor or a tightly coupled parallel processor. We are interested search structures in an asynchronous distributed system, in which a global time does not exist. As a result, we need to be careful in defining the state of the search structure at some particular point in the computation.

The nodes of the search structure are distributed across the processors in some manner while the operations are performing their actions. Each node will have contain some keys (perhaps none), and will contain pointers to other nodes, possibly stored on different processors. Intuitively, the col-

lection of these nodes and their contents are collectively the global state, but the question is, when do you look at the nodes in order to collect the global state? Due to restrictions on the speed of light, we cannot examine the state of all of the nodes simultaneously. While we are recording the nodes in one processor, the state of the nodes might be changing in another. We will use the concept of a consistent snapshot [7] to define our global state. The global state corresponds to what we might see if we were to run the snapshot algorithm. If all actions have been completed, and there is a simultaneous global state, then that is the snapshot that we will get.

The search structures that we are modeling are also replicated search structures, so that a node of the logical search structure might be stored at several different processors. We say that the physically stored replicas of the logical node are *copies* of the logical node. We will generally refer to the physically stored records as copies, and the logical records as nodes. Since we are using an asynchronous instead of a synchronous system, replicated copies of a node will not in general have the same value. To indicate that a set of copies are members of a set of copies of a node, we will mark all physical nodes in the system with a label. Two copies have the same label iff. they are replicated copies of the same logical node. We refer to the logical node by the label assigned to the copies.

The *state* S of a distributed search structure is

$$S = (N, C, copies, E, root, contents, edgeset)$$

where

- N is the set of nodes in the logical search structure.
- C is the set of physically stored copies.
- *copies* maps a logical node to its physical copies. We use the function *label* as the inverse mapping of *copies*. We note that there might be some copies that have no corresponding node.
- E is the set of directed edges in the data structure, $E \subseteq copies(N) \times copies(N)$.
- *root* is a distinguished node, $root \in N$. The copies of the root are *root copies*.
- *contents* is a function that returns the keys contained in a copy of a node. $contents : c \Rightarrow 2^{keyspace}$.
- *edgeset* is a function that returns the set of keys that an operation that traverses the edge would be searching for. $edgeset : E \Rightarrow 2^{keyspace}$, $edgeset(c, c') = \{k |$ an operation on key k starting at c will traverse E to $c'\}$.

If in state s, all copies with label l have the same value, then logical node l is *single copy equivalent*. If all logical nodes l are single copy equivalent

in state s, then the search structure is single copy equivalent. If for every $c \in C$, $contents(c) \neq \emptyset$ implies that $|copies(label(c))| = 1$, then the search structure *does not replicate data*. In this paper, we consider only search structures that do not replicate data.

Given a path $P = (e_1, e_2, \ldots, e_k)$, the *pathset* of P is the intersection of the edgesets of the edges in P. Suppose that there is only one copy of the root, $copy(root)$ (In this case we call the search structure *single rooted*). The *inset* of a copy c is the union of the pathsets of all paths from the $copy(root)$ to c. The *outset* of a node n is the union of the edgesets of all edges leaving n. The *keyset* of a node n is the difference between the inset and the outset of n. That is, $keyset(n)$ is the set of keys that could be stored in n.

A (single-rooted) search structure is in a *good state* if:

- $keyset(c) | c \in C$ partitions the keyspace.
- $\forall_{c \in C} contents(c) \subseteq keyset(c)$

A *multirooted* search structure has several root copies, and possible several roots. A multirooted search structure is in a good state if:

- The data structure is in a good state wrt. the edge set of every root copy.
- For every node, its keyset is the same wrt. every root copy.

The *global contents* GC of the search structure is the union of the contents of the nodes in the search structure $GC = \cup_{n \in N} contents(n)$. The *operations* on a data structure map the structure from one global contents to another. Each operation has a *specification*, which is a mapping between stated.

- $member(x)(ds) = (ds, bool)$ where $bool = true$ iff $x \in ds$
- $insert(x)(ds) = (ds', nil)$ where $ds' = ds \cup \{x\}$
- $delete(x)(ds) = (ds', nil)$ where $ds' = ds - \{x\}$

An operation executes by submitting an initial action, whose execution submits subsequent actions, and so forth until no further actions due to the operation are executing or pending at any processor. We define an execution of an operation O to be $(A, <, r)$, where A consists of O and the actions that the program for O issues, $<$ is a partial ordering on the actions in A, and r is the return values of the actions in A. The partial ordering $<$ is the causal happens-before ordering: $a_1 <^* a_2$ iff. a_1 and a_2 are executed on the same processor and a_1 is executed before a_2, or a_1 on processor p_1 sends message m to processor p_2 that executes a_2, and p_2 receives m before it executes a_2. Then, $<$ is the transitive closure of $<^*$. If two actions are related by $<$, then they are *ordered*, otherwise they are *concurrent*. An operation might contain concurrent actions if it performs an update on all copies of a replicated node, for example.

A *computation* is a quadruple $(s, s', E, <)$, where s is the search structure

state before the computation starts, s' is the search structure state after the computation completes, $E = \{(A_1, <_1, r_1), \ldots, (A_n, <_n, r_n)\}$ is the set of operations in the execution, and $<$ is a partial order that extends the operation partial orders $(\cup_{i=1}^{n} <_i) \subseteq <$. The partial order $<$ means that if $a < a'$, then a completes before a' begins (so actions that occur simultaneously aren't ordered).

PROPOSITION 1. *If $c = (s, s', E, <)$ is a distributed search structure computation, then there is a total order $<'$ on the actions of c such that $< \subseteq <'$ and $<'$ does not change the return values or the final state of any of the actions.* *Proof:* Since there is no communication between processors that execute concurrent actions, we can order two concurrent actions arbitrarily. Thus, we can place a total order on the execution of the actions of an operation. Similarly, we can place a total order $<_c$ on the actions in the distributed computation $(S, S', E, <)$ (By using Lamport's timestamps, for example) •

The actions A_c of the search structure computation c can thus be serialized as $A = (a_1, a_2, \ldots, a_m)$. The sequence of states $s_i = a_i(s_{i-1})$, $s_0 = s$ corresponds to a sequence of snapshots of the global state that we could take during the computation. A different set of snapshots would order concurrent actions differently, and we would see a different set of global states, but the operation return values and the final search structure state would be the same. The serialized order is thus one possible serialized order, but all serialized orders are equivalent in all important aspects.

Let $(s, s', E, <)$ be a computation, and let O_i be the parent of $(A_i, <_i, r_i)$ in E, $i = 1 \ldots, n$. Let $P = \{O_1, \ldots, O_m\}$. A computation c is (semantically) *serializable* if

1. P has a total order $<_P$. The execution of the sequence $(P, <_P)$ on GC_s yields $GC_{s'}$ for a final state, and returns $r(O_i)$ as the return value for each operation O_i.

A computation c is *strict serializable* is

1. It is serializable.
2. $\forall_{j,k \in 1,\ldots,m}, (\forall_{x \in A_j} \forall_{y \in O_k} x < y) \Rightarrow O_j <_P O_k$.

If one of the actions $a_{dec} = dec(O)$ of operation O can be distinguished as the *decisive action*, then we can define another type of serializability. A computation c is *decisive action serializable* if

1. c is serializable
2. $\forall_{j,k \in 1,\ldots,m}(dec(O_j) < dec(O_k)) \Rightarrow O_j <_p O_k$.

The decisive action of an operation is typically the one that changes the global state of the data structure, or which decides the return value. Decisive operation serializability is a stronger condition than strict serializability, (decisive operation serializable implies strict serializable) but is somewhat

easier to prove.

The correctness condition for a computation is some type of serializability – that it is equivalent to a serial execution. We generally start by assuming that a base serial algorithm is correct, and building on the serial algorithm. That is, we assume that the analogue of the serial algorithm, which issues actions to all copies simultaneously and executes in isolation, is correct.

As mentioned previously, the decisive action of an operation is meant to be the action that "performs" the operation. We say that action a of operation o is *proper decisive* if action a performs o's specification. A total well-formed search structure computation c is a *keyset computation* if

1. c begins in a good state and each action in c maps a good state to a good state.
2. Each dictionary operation execution has exactly one decisive action, which is proper decisive.
3. Nondecisive actions do not change the global contents.

THEOREM 1. *If $c = (s, s', E, <)$ is a keyset computation, then c is strict serializable.* *Proof:* A keyset computation is a total well-formed computation, so all of the actions can be ordered. In particular, the decisive actions of the operations can be ordered. Let $DO = (o_1, \ldots, o_n)$ be the ordering of the dictionary operations of c according to the ordering of their decisive actions. We let DO be the serial execution, whose specification is $((ds_1, v_1), \ldots, (ds_n, v_n))$. Consider the actions of c. Since the non-decisive operations don't change the global state, the global state remains the same from one decisive action to another. Since actions map good states to good states, when a decisive action is performed, it maps a representative of ds_{i-1} to ds_i, and returns the value v_i. The global contents don't change after the last decisive operation, so the final state of the computation is ds_n. Therefore, c is decisive operation serializable and thus strict serializable□

We actually want a stronger condition than serializability, since we want the search structure to look like a single-copy search structure. In particular, we want the the final state of the search structure to be single copy equivalent.

Definition: a distributed search structure computation is a *distributed keyset computation* if it is a keyset computation, and s' is single copy equivalent.

A distributed keyset computation is a keyset computation, so the operations are serializable. If the final state is single copy equivalent, then the search structure will have the appearance of a single copy search structure. We note that a distributed keyset computation is only required to be single copy equivalent in the final state, not in any intermediate state. The issue

of single copy equivalence is addressed in [20].

A *link algorithm* is a concurrent data structure that uses linked nodes to allow concurrent actions without locks. If an operation misnavigates due to the action of some other operation, the misnavigated node can *recover* from the misnavigation by following the link pointers. Code 1 shows the execution of a typical link algorithm. In the case of the dB-tree, the search, insert, and delete actions on the leaves are the decisive actions.

Shasha and Goodman show a correctness criteria for link-type algorithms: Let $P(n, m)$ be the set of all paths from n to m. Then we define the *inreach* of a node to be:

$$\cup_{m \in V, p \in P(n,m)} pathset(p) \cap inset(m)$$

Intuitively, the inreach of a node n is the set of keys that 'should be stored' in n and the keys that 'should be stored' in other nodes for which there is a legal path to that node.

The restructuring phase of the operation might do nothing but release the lock on `n`. A link algorithm will work if the inreach of `next` does not decrease - so that an operation can recover from an undetected decrease in the inset of `next`. Shasha and Goodman present the following guidelines for link-type algorithms

1. Every non-decisive action maps a good state to a good state.
2. If $x \in keyset(n)$ in the state preceding $dec(o)(x, n)$, and $dec(o)(x, n)$ occurs, then $dec(o)(x, n)$ is a proper decisive action for o and maps a good state to a good state.
3. Nondecisive action do not change the global contents.
4. If an action a removes x from $inreach(n)$, then no action a' with argument x accesses n after a.

THEOREM 2 (SHASHA AND GOODMAN). *If a computation C begins in a good state, and the dictionary actions in C use the link technique and satisfy the guidelines, then C is decisive-action serializable.*

COROLLARY 1. *The dB-tree is decisive-action serializable. Proof:* The proof of the corollary consists of applying Theorem 2 to the dB-tree, regarding it as a multi-rooted search structure. We note first that the dB-tree uses the link technique. Thus, we need to show that the guidelines are followed:

1. Keys are stored only in leaf nodes, and the leaf nodes partition the keyspace. Therefore, every non-decisive action maps a good state to a good state.
2. A search insert or a delete operation uses the node-search action to locate the key with the proper key range, then atomically perform the decisive action. Therefore, the decisive action is proper decisive

and maps a good state to a good state.

3. The global contents of the dB-tree are stored in the leaves. If the replica coherency algorithm ensures that link-change actions are executed in the order they are issued, then every leaf is reachable from every node. For a period of time, a parent may point to a deleted node whose space is reclaimed, and hence a portion of the key space $\mathcal{K}$ may not be directly reachable from the parent. However, the algorithm detects this condition, and the link will eventually be changed, and $\mathcal{K}$ will again be reachable. Hence, operations on $\mathcal{K}$ are blocked for a period of time, but keys in $\mathcal{K}$ are not removed from the global contents.
4. After the space for a node is reclaimed, the algorithm does not permit access to the node, and instead executes a recovery action.

□

4. Distributing the Data : The dE-tree. A simple implementation of a distributed dictionary is to statically divide the key range among the processors and direct operations to the processor that manages the operation's key. The problem with the simple approach is that the processors will not be data-balanced. Even if each insert is equally likely to be directed to each processor, the processor with the most keys will hold considerably more keys than the average [10]. Data-balancing is necessary in order to distribute the request load, prevent processors from being required to devote a disproportionate share of their memory resources to the dictionary, and prevent out-of-storage errors when storage is available on other processors.

Ellis' algorithm [12] performs data-balancing whenever a processor runs out of storage. Peleg [16, 36] has studied the issue of data-balancing in distributed dictionaries from a complexity point of view, requiring that no processor store more than $O(M/N)$ keys, where M is the number of keys and N is the number of processors. In practice, this definition is simultaneously too strong and too weak, because it ignores constants and node capacities.

Our dB-tree provides us with the flexibility to perform data balancing among the processors, for the leaves of the tree can be distributed among the processors in an arbitrary fashion. If a processor splits a node, it can assign the newly created node to the processor that owns the fewest leaves.

Data balancing using the above mechanism is expensive, because every restructuring requires communication among the processors, and causes the internal nodes to be widely replicated. One way to reduce the communication costs is to try to have neighboring leaves stored in the same processor. We define an *extent* to be a maximal length sequence of neighboring leaves that are owned by the same processor. When a processor decides that it owns too many leaves, it first looks at the processors who own neighboring extents. If

the neighbor will accept the leaves, the processor transfers some of its leaves to the neighbor. If no neighboring processor is lightly loaded, the heavily loaded processor searches for a lightly loaded processor and creates a new extent.

Figure 8 shows a four processor dB-tree that is data balanced using the extents. Notice that the extents have the characteristics of a leaf in the dB-tree: they have an upper and lower range, are doubly linked, accept the dictionary operations, and are occasionally split or merged. We can transform the extent-balanced dB-tree into the *dE-tree*: the distributed extent tree. Each processor manages a number of extents. The keys stored in the extent are kept in some convenient data structure. Each extent is linked with its neighboring extent.

The extents are managed as the leaves in a dB-tree. When a processor decides that it is too heavily loaded, it first looks at the neighboring extents to take some of its keys. If all neighboring processors are heavily loaded, a new extent is created for a lightly loaded processor. The processor that manages the new extent can chosen according to a number of heuristics that examine factors such as data load, processor capacity, communication locality, and degree of replication. The creation and deletion of extents, and the shifting of keys between extents in the dE-tree correspond to splitting and merging leaves in the dB-tree, and the index can be updated by using dB-tree algorithms.

The primary advantage of the dE-tree over the dB-tree is a great reduction in the amount of index restructuring that is required, since many more keys must be inserted or deleted before a restructuring is required. In addition, the size of the distributed index is reduced, because the size becomes proportional to the number of processors instead of the number of keys, and we assume that each processor can store many keys.

4.1. Propagating Load Information. Information about the load on processors must be distributed throughout the system so that informed decisions for creating and placing new extents can be made. The design of a load propagation mechanism must address the tradeoff between propagating information quickly to all processors, which can require significant communication bandwidth to distribute the load information, and propagating information either too slowly or to too few processors, which can result in load imbalance and processors sitting idle because the system is slow to react to changes in the load on individual processors.

In addition, many load-balancing techniques become unstable in large systems. Instability can arise because information is propagated too slowly, so that many processors may think another is lightly loaded long after it becomes heavily loaded and continue to create tasks there. It can also arise

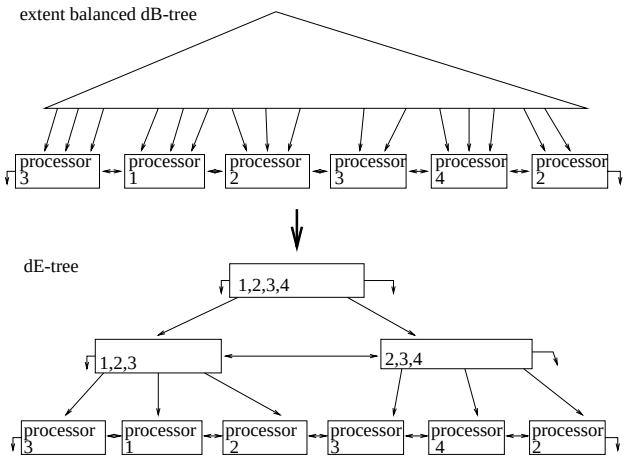

FIG. 8. *The dE-tree*

because decisions to create new extents are made quickly when the load on another processor changes, so that many new extents are moved to a processor that suddenly becomes lightly loaded. To avoid these problems, information must be propagated reasonably rapidly through the system, but the number of processors that will react quickly to a change in load on a particular processor must be limited.

Suitable load propagation mechanisms include probing [6], gradient methods [28], processor pairing [14, 5], drafting algorithms [35] and bidding algorithms [13]. The ultradiffusive algorithms of Lumer and Huberman [29], which uses a hierarchical control structure, are particularly well suited when the number of processors is large.

5. Analysis. The dB-tree algorithm is designed to avoid blocking and permit highly concurrent access. The only occasion when an action is blocked occurs when a parent points to a deleted child whose space is reclaimed. If the action is resubmitted to the parent, the parent can submit the action to one of the node's neighbors, or it can block the action. If the action is routed to a node on the same level as the deleted node, then the action will not be blocked. Further, updates on a node can use non-blocking lazy updates [20], which permit concurrent updates of a node.

The performance of the dB-tree depends on many factors. For example, the message passing overhead depends on the network topology, the distribution of nodes to processors (i.e, the load balancing algorithm), and the implementation of replica coherency.

Let us consider the space overhead required by the dE-tree. In the best case, each processor stores $O(1)$ extents, and in the worst case each processor stores $O(P)$ extents, where P is the number of processors. Since a processor stores the nodes on the path from the root to a leaf, each processor stores between $O(\log(P))$ and $O(P\log(P))$ internal nodes. Ranade [38] gives algorithms for storing a search tree on a synchronous mesh or butterfly network that require $O(1)$ storage overhead per key. While our suggested replication strategy does not make this storage complexity guarantee, we note that a mass storage system will typically store $M \gg P\log(P)$ bytes per processor.

If P processors store a dE-tree, then at most $2\log(P)$ messages must be passed to perform the desired operation. Index restructuring is only required when data balancing is performed, which is relatively rare [23]. Thus, the dE-tree is an efficient distributed index in practice.

6. Conclusions. We present a distributed dictionary based on the B-link tree, the dB-tree. The interior nodes of the tree are replicated in order to allow many processors to read the same node and thus increase parallelism. The degree of replication decreases as one descends from the root, in order to control the cost of maintaining replica coherency. Update operations on the interior nodes rarely block operations in their search phases and can proceed concurrently on the same replicated node in some cases, further increasing the parallelism of the tree.

We use the flexible dB-tree to construct an efficient data-balanced dictionary, the dE-tree. The dE-tree assigns key ranges to processors, and a processor may maintain several key ranges. The dB-tree is used to allow the key ranges to be created, deleted, or modified without centralized control.

7. Acknowledgments. We thank William Weihl for his comments and suggestions on this work.

REFERENCES

[1] R. Bayer, *Symmetric Binary B-trees: Data Structure and Maintenance Algorithms*, Acta Informatica, 1 (1972), pp. 290–306.

[2] R. Bayer and M. Schkolnick, *Concurrency of operations on B-trees*, Acta Informatica, 9 (1977), pp. 1–21.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[4] A. Biliris, *Operation specific locking in B-trees*, in Symposiun on the Principles of Database Systems, ACM SIGACT-SIGART-SIGMOD, 1987, pp. 159–169.

[5] R. Bryant and R. Finkel, *A stable distributed scheduling algorithm*, in Proceedings of the International Conference on Distributed Computing Systems, 1981, pp. 314–323.

[6] C. G. Cassandras, J. F. Kurose, and D. Towsley, *Resource contention management in parallel systems*, radc-tr-89-48, Rome Air Development Center (RADC), April 1989.

[7] K. Chandy and L. Lamport, *Distributed snapshots: Determining global states of distributed systems*, ACM Transactions on Computer Systems, 3 (1985), pp. 63–75.

[8] A. Colbrook, E. Brewer, C. Dellarocas, and W. Weihl, *An algorithm for concurrent search trees*, in Proceedings of the 20th International Conference on Parallel Processing, 1991, pp. III138–III141.

[9] M. Dietzfelbinger and F. Meyer auf der Hyde, *An optimal parallel dictionary*, in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1989, pp. 360–368.

[10] K. Donovan, *Performance of shared memory in a parallel computer*, IEEE Transactions on Parallel and Distributed Systems, 2 (1991), pp. 253–256.

[11] C. Ellis, *Concurrent search and inserts in 2-3 trees*, Acta Informatica, 14 (1980), pp. 63–86.

[12] ———, *Distributed data structures: A case study*, IEEE Transactions on Computing, C-34 (1985), pp. 1178–1185.

[13] K. H. et al, *A unix-based local computer network with load balancing*, IEEE Computer, (1982), pp. 55–64.

[14] R. Finkel, M. Solomon, and M. Horowitz, *Distributed algorithms for global structuring*, in Proceedings of the National Computer Conference, 1979, pp. 455–460.

[15] D. Gifford, *Weighted voting for replicated data*, in Proceedings of the Seventh Annual ACM Symposium on Operating System Principles, ACM, 1979, pp. 150–150.

[16] K. Gilon and D. Peleg, *Compact deterministic distributed dictionaries*, in Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, ACM, 1991, pp. 81–94.

[17] L. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, in Proc. 19th Annual Symposium of Foundations of Computer Science, ACM, 1978, pp. 8–21.

[18] M. Herlihy, *A quorum-consensus replication method for abstract data types*, ACM Transactions on Computer Systems, 4 (1986), pp. 32–53.

[19] T. Johnson, *The Performance of Concurrent Data Structure Algorithms*, PhD thesis, NYU Dept. of Computer Science, 1990.

[20] T. Johnson and P. Krishna, *Lazy updates in distributed search structures*, in SIGMOD '93, 1993.

[21] T. Johnson and D. Shasha, *Utilization of B-trees with inserts, deletes and modifies*, in ACM SIGACT/SIGMOD/SIGART Symposium on Principles of Database Systems, 1989, pp. 235–246.

[22] ———, *A framework for the performance analysis of concurrent B-tree algorithms*, in ACM Symp. on Principles of Database Systems, 1990, pp. 273–287.

[23] P. Krishna and T. Johnson, *Implementing distributed search structures*, Tech. Report UF CIS TR92-032, Availiable at anonymous ftp site cis.ufl.edu, University of Florida, Dept. of CIS, 1992.

[24] Y. Kwong and D. Wood, *A new method for concurrency in B-trees*, IEEE Transactions on Software Engineering, SE-8 (1982), pp. 211–222.

[25] V. Lanin and D. Shasha, *A symmetric concurrent B-tree algorithm*, in 1986 Fall Joint Computer Conference, 1986, pp. 380–389.

[26] G. Lausen, *Integrated concurrency control in shared B-trees*, Computing, 33 (1984), pp. 13–26.

[27] P. Lehman and S. Yao, *Efficient locking for concurrent operations on B-trees*, ACM Transactions on Database Systems, 6 (1981), pp. 650–670.

[28] F. Lin and R. Keller, *The gradient model load balancing method*, IEEE Transactions on Software Engineering, SE-13 (1987), pp. 32–38.

[29] E. Lumer and B. Huberman, *Dynamics of resource allocation in distributed systems*, preprint (submitted to ieee transactions on smc), Xerox Palo Alto Research Center, March 1990.

[30] G. Matsliach and O. Shmueli, *An efficient method for distributing search structures*, in Symposium on Parallel and Distributed Information Systems, 1991, pp. 159–166.

[31] R. Miller, *Multiple access to B-trees*, in Proceedings of the 1978 Conference on Information Sciences and Systems, Johns Hopkins University, Baltimore, MD, 1978, pp. 400–408.

[32] C. Mohan and F. Levine, *ARIES/IM: An efficient and high concurrency index management method using write-ahead logging*, Research Report RJ 6864, IBM, 1989.

[33] Y. Mond and Y. Raz, *Concurrency control in B^+-trees databases using preparatory operations*, in 11th International Conference on Very Large Databases, Stockholm, Aug. 1985, pp. 331–334.

[34] T. Ng, *Using histories to implement atomic objects*, ACM Transactions of Computer Systems, 7 (1989), pp. 360–393.

[35] L. Ni, C. Xu, and T. Genfreau, *A distributed drafting algorithm for load balancing*, IEEE Transactions on Software Engineering, SE-11 (1985), pp. 32–38.

[36] D. Peleg, *Distributed data structures: A complexity oriented view*, in Fourth Int'l Workshop on Distributed Algorithms, Bari, Italy, 1990, pp. 71–89.

[37] M. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 19787.

[38] A. Ranade, *Maintaining dynamic ordered sets on processor networks*, in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1992, pp. 127–137.

[39] Y. Sagiv, *Concurrent Operations on B-Trees with Overtaking*, Journal of Computer and System Sciences, 33 (1986), pp. 275–296.

[40] P. Schwartz and A. Spector, *Synchronizing abstract data types*, ACM Transactions on Computer Systems, 2 (1894), pp. 223–250.

[41] D. Shasha and N. Goodman, *Concurrent search structure algorithms*, ACM Transactions on Database Systems, 13 (1988), pp. 53–90.

[42] D. Shasha, V. Lanin, and J. Schmidt, *An analytical model for the performance of concurrent B-tree algorithms*, NYU Ultracomputer Note 311, NYU Ultracomputer lab, 1987.

[43] V. Srinivasan and M. Carey, *Performance of B-tree concurrency control algorithms*, Tech. Report Computer Sciences Technical Report 999, University of Wisconson-Madison, 1991.

[44] R. H. Thomas, *A majority consensus approach to concurrency control for multiple copy databases*, ACM Transactions on Database Systems, 4 (1979), pp. 180–209.

[45] P. Wang, *An in-depth analysis of concurrent b-tree algorithms*, Tech. Report MIT/LCS/TR-496, MIT Laboratory for Computer Science, 1991.

[46] W. Weihl, *Commutivity-based concurrency control for abstract data types*, IEEE Trans. Computers, 37 (1988), pp. 1488–1505.

[47] ———, *The impact or recovery on concurrency control*, Tech. Report MIT/LCS/TM-382b, MIT Laboratory for Computer Science, 1989.

[48] W. Weihl and B. Liskov, *Implementation of resilient, atomic data objects*, ACM Transactions on Programming Languages and Systems, 7 (1985), pp. 244–269.

[49] W. Weihl and P. Wang, *Multi-version memory: Software cache management for concurrent B-trees*, in Proc. 2nd IEEE Symp. Parallel and Distributed Processing, 1990, pp. 650–655.

[50] I. Yen and F. Bastani, *Hash tables in massively parallel systems*, in Int'l Parallel Processing Symposium, 1992, pp. 660–664.