JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **9**, 69–76 (1990)

On Performing Robust Order Statistics in Tree-Structured Dictionary Machines

MICHAEL T. GOODRICH*

Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218

AND

MIKHAIL J. ATALLAH†

Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907

We show how to extend any tree-structured dictionary machine so that it can perform order statistics robustly. In particular, we consider how to allow for redundant insertions, deletions, and updates, as well as operations based on the ranks of data items, such as **Extract**(j), which simultaneously selects and deletes the jth smallest data item. All these operations can be performed without ever interrupting the pipelining of responses coming from the machine, and the resulting machine has the same interval time and latency time performance as the original design, to within constant factors. © 1990 Academic Press, Inc.

1. INTRODUCTION

An application of parallel computing which seems to offer many rewards is that of manufacturing powerful, special-purpose computing devices which allow general-purpose (host) computers to off-load some of their computational burden. Specifically, one such application is the design of dictionary machines (sometimes referred to as database machines) [1–4, 6–9]. A dictionary machine is a computing device which is to be used to allow a host computer to off-load certain operations for maintaining and querying data files.

For a dictionary machine to be attractive for use with a general-purpose computer, it should obviously perform the required operations better than the general-purpose computer can. One way such a special-purpose machine can excel over a general-purpose machine is in the use of parallelism. The possible speedups are even more dramatic if the special-purpose machine has the ability to pipeline operations. That is, the host computer using such a machine can send it a stream of operations, and can get back a stream of responses. In this context there are two useful measures for evaluating a machine's performance: its *latency* time and its *interval* time (or *period*). The latency time of a machine is the time it takes one operation to flow through the machine, from the input of the operation to the output of its result. The interval time of a machine is the time that elapses between consecutive responses. In designing an efficient special-purpose machine one wishes to minimize both of these quantities. Typically, one wishes to have a constant interval time and a latency time which is logarithmic in either the size of the machine or the number of elements it stores.

One design which makes considerable use of this type of parallelism is the tree-structured dictionary machine [1, 2, 3, 6, 8, 9], depicted in Fig. 1. Such a machine is structured as a binary tree of processing elements (PEs), and stores data items internally, each PE having $O(1)$ storage registers. Operations are sent to the dictionary machine in a pipelined fashion, the execution of an operation beginning when it is input as a single instruction to the tree's root processor. (These high-level instructions should not be confused with the primitive instructions which compose the instruction set for the CPU in a PE.) Instructions flow through the machine like waves, flowing down the tree in a broadcast mode until they reach the last level of the tree storing data items, at which point the instructions "bounce" and start flowing up the tree in a combining mode, eventually resulting in an answer being output at the root. For example, the response from a **Delete**(k) instruction could be that each PE indicates by responding with a "yes" or "no" whether it deleted

* This author's research was supported by a grant from the Sperry Corp., by a David Ross Grant from the Purdue Research Foundation, and by the National Science Foundation under Grant CCR-8810568.

† This author's research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and by the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.

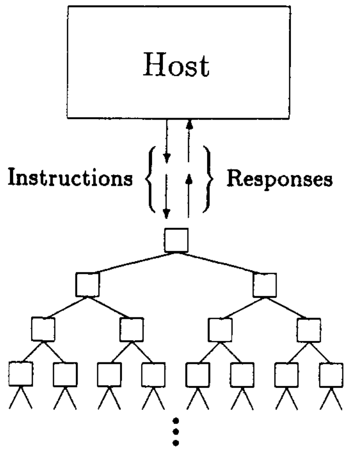

FIG. 1. Instruction pipelining in the dictionary machine.

the specified item or not. For the host to know if the item was deleted all of the responses would need to be combined into just one answer: "yes" or "no." Or, alternatively, the operation could be a query to count how many elements fall within a certain range, in which case each PE with an element in the range would respond with 1 and all others would respond with 0. The combining operation would simply be to sum up all the responses.

Some of the previous designs (e.g., [1, 6]) describe dictionary machines which are robust in the sense that they allow for graceful recovery from redundant **Insert** and **Delete** operations. That is, the machine can recover gracefully if an **Insert** operation tries to insert something that is already in the dictionary machine or if a **Delete** tries to remove something that is not in the machine. Clearly, one desires that a special-purpose machine be robust if its task is to off-load as much of the computational burden from the host computer as possible.

Much of the previous work on tree-structured dictionary machines [1–3, 6, 8, 9] was dealt with efficient ways of implementing simple operations, such as inserting a data item, deleting an item, finding an item, etc. Some of these papers describe machine designs in which data items are stored close to the top of the tree, thus optimizing the latency time of the machine. Others describe ways of optimizing the interval time of the machine. Because so much work has been spent optimizing the simple operations, one might be led to the misconception that a tree-structured dictionary machine cannot perform more complex operations, such as rank-based queries. A notable exception to this restriction to simple instructions is the design of Song [9] which can perform complex operations, such as multiselection, sorting, and join. This design has some unfortunate drawbacks, however, in that it does not support rank-based queries, it requires that data items be stored only at the leaves of the tree, and operations cannot be redundant.

In this paper we show how to extend any tree-structured dictionary machine design so that it can perform order statistics and other data operations in a robust fashion, even if the previous design was not robust. It is this robustness that makes our design interesting, for if we were to disallow redundant **Insert**'s and **Delete**'s, then rank-based queries would be straightforward to implement by simply having each data item in the dictionary machine keep track of its rank. This becomes nontrivial, however, in the presence of redundant operations, especially if we allow for an operation **Extract**(j), which simultaneously selects and deletes the jth smallest data item. This **Extract**(j) operation is a generalization of the **ExtractMin** and **ExtractMax** operations that some of the previous designs support [1, 6, 8].

Our scheme does not depend on any single way of implementing the low-level instructions, such as insertion and deletion of data items. Instead, it is a design built "on top" of any existing method for implementing these instructions. Our methods guarantee that the generalized dictionary machine will achieve the same latency and interval times as the original machine, to within constant factors. In addition, our scheme is general enough that it can be implemented in a synchronous VLSI environment or in a SIMD parallel environment which allows for on-line definable "masking" vectors (which identify which PEs perform an instruction and which ones simply perform a no-operation).

The remainder of this paper is organized as follows. In the next section we review the tree-structured dictionary machine model, and in Section 3 we describe our protocol for instruction flow and show how to implement order statistics robustly, even in the presence of **Extract**(j) instructions.

2. THE TREE-STRUCTURED DICTIONARY MACHINE

The machine model we will be using incorporates the common properties of previous tree-structured dictionary machine designs [1–3, 6, 8, 9]. The general framework is that the machine's topology is a complete binary tree. (Some designs have additional connections, but we will only use edges of the binary tree.) The data elements are stored in the nodes of this tree, $O(1)$ elements per node. Some of the previously proposed machines have data items stored only at the leaves of the tree, but for the purpose of more generality we will assume that a data item can be stored at any node in the tree. Also, without loss of generality, we assume that each PE can store at most one data element. This allows us to refer to *the* element stored in a PE.

The flow of an instruction through the machine is as follows. The execution of the instruction begins when it is sent to the root of the dictionary machine from the host computer. We assume that each PE executes synchronously using a global clock, or, equivalently, that the dictionary machine is implemented to support SIMD operation with on-line definable masking vectors for enabling and disabling PEs. A PE receives an instruction from its parent, executes the instruction on its contents, then divides the instruction into two instances, called *bubbles,* and sends them to its children. It includes its response (if there was one) with one of the bubbles, and its parent's response with the other. The instruction continues to be broadcast down the tree in this manner, PEs broadcasting bubbles to their children. As the bubbles from this instruction move down the tree, like soldiers marching in formation, they carry along the responses that have been generated so far. We define the last level of the tree still containing data items to be the *effective last level* of the tree. When bubbles from an instruction reach the effective last level of the tree they stop being broadcast down the tree, "bounce," and start moving back up the tree to be output, combining with each other as they move up. When a (response) bubble arrives at the root, the root simply sends it back to the host. An exception to this generic execution pattern is the design of Somani and Aggarwal [8], where instructions can be executed as they are going up the tree. The algorithms we describe in this paper assume that instructions execute as they travel down the tree. Modifying our machine design so that instructions execute on the way up is straightforward.

A data item is represented as a key–record pair (k, r), where k is a key uniquely identifying the item, and r is a data record. In a database it is often useful for k to be a tuple $(k_1, k_2, \ldots, k_d)$. Without loss of generality, we consider each k to be single integer. An instruction bubble consists of an instruction–response pair (i, b), where i is the instruction being executed and b is a response (i.e., an answer) from some PE which has executed i on its contents. We place no restrictions on the response b, but typically it will be $\varnothing$ (the null response), "yes," "no," or some (k, r) pair.

There are six primitive operations which previously proposed machines can perform:

1. **Insert**(k, r): insert a (k, r) pair;
2. **Delete**(k): delete the (k', r) pair for which $k' = k$;
3. **Update**(k, r): replace (k, r') by (k, r), if there is no (k, r') pair then this operation has no effect;
4. **Member**(k): return the (k', r) pair for which $k' = k$ if there is such a pair, otherwise, return "no";
5. **Min**( ): return the (k, r) pair which has the minimum k value among all those stored in the machine;
6. **ExtractMin**( ), **ExtractMax**( ): simultaneously select and delete the (k, r) pair with smallest (largest) k value [1, 6, 8].

Since the implementation details for operations 1, 2, and 3 above differ from machine design to machine design, we will not describe their implementation in any detail. For the details of various implementations see the references at the end of this paper. The only thing we assume is that, like all operations, operations that can modify the contents of the dictionary machine execute while traveling down the tree. We will describe how to perform operations 4, 5, and 6 in this paper, and in fact will describe a generalization of operation 6.

2.1. *Robustness*

One of the important aspects of our design is that it is robust. That is, the machine can gracefully recover from redundant instructions. An **Insert**(k, r) is redundant if there is already a pair (k, r') in the tree, and a **Delete**(k) or **Update**(k, r) is redundant if there is no pair (k, r') in the tree. Not all machine designs allow for redundant operations, and some of the ones that do allow for them require that all redundant **Insert**'s be treated like **Update**'s. Our solution allows for any kind of error recovery the user wants. The main idea of our method is to change the way instructions are pipelined through the machine similar to the method used by Atallah and Kosaraju in [1]. Instead of only going up and down the tree once, instructions in our machine go up and down the tree twice. In the first round-trip instructions are tagged as being either "redundant" or "nonredundant." Then, in the second round-trip, the nonredundant instructions will perform, more or less, just as in the original design [1–3, 6, 8, 9]. The redundant instructions will perform some error recovery procedure (usually a no-operation) in their second round-trip. For example, a redundant **Insert**(k, r) could act as a nonredundant **Update**(k, r) or it could simply do nothing, allowing the (k, r') pair already in the machine to remain unchanged.

If we restrict the operations which can modify the contents of the machine to **Insert**(k, r), **Delete**(k), and **Update**(k, r), then the original scheme of Atallah and Kosaraju [1], as just outlined, will correctly tag an instruction as "redundant" or "nonredundant" by the end of its first round-trip through the machine. Their scheme is not sufficient, however, for maintaining robustness in the presence of content-modifying instructions that depend on the *rank* of data elements.

Since our method builds on that of Atallah and Kosaraju, we review their scheme here, and then in the next section show how to extend it for our design. Their scheme called for instructions traveling down the tree in their first round-trip to examine the (k, r) pairs in the tree, and the instruction traveling up the tree in their first round-trip to examine the instructions traveling down the tree in their second round-trip (see Fig. 2). The redundancy tags of the instructions in their first round-trip are updated depending on the results of these comparisons. Operations flowing through the tree in their second round-trip operate exactly as before.

We refer to the updating done while instructions travel down in their first round-trip as *phase 1 updating* and up-

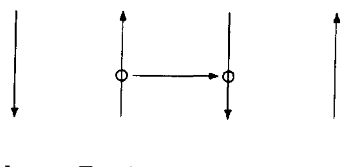

FIG. 2. The phases of instructions flowing through the machine. The first round-trip consists of a broadcast phase and a tagging phase; the second round-trip consists of an execute phase and an output phase.

dating while instructions travel up in their first round-trip as *phase 2 updating.* One updates the redundancy tags as follows:

Phase 1 Redundancy Tag Updating:

A nonredundant **Insert**(k, r) becomes redundant by finding a pair (k, r') in the tree. Similarly, a redundant **Delete**(k) or **Update**(k, r), becomes nonredundant by finding a pair (k, r') in the tree.

Phase 2 Redundancy Tag Updating:

A nonredundant **Insert**(k, r) becomes redundant, or a redundant **Delete**(k) or **Update**(k, r) becomes nonredundant, by finding a pair (k, r') in the tree or by encountering a nonredundant **Insert**(k, r') instruction making its way down the tree in its second round-trip.

A redundant **Insert**(k, r) becomes nonredundant, or a nonredundant **Delete**(k) or **Update**(k, r) becomes redundant, by encountering a nonredundant **Delete**(k) in its second round-trip down the tree.

Initially, all **Insert**(k, r) instructions are tagged nonredundant, and all **Delete**(k) and **Update**(k, r) instructions are tagged redundant. On the way up the tree in the first round-trip the tags are combined so that if the tags of two bubbles to be combined into one differ, then one always prefers the tag which is different from the tag the instruction was given initially when it was input to the machine [1]. Thus, if differences occur, the tag always represents a value it was *changed* to.

As mentioned before, this scheme does not work in the presence of rank-based content-modifying instructions. In the next section we show how to eliminate this drawback.

3. ROBUST ORDER STATISTICS

Since our methods are based on how instructions flow through the machine, we begin by describing the instruction-flow protocol.

3.1. *A Simple Instruction-Flow Protocol*

In this section we describe our protocol for how instructions should flow through the machine. An instruction flowing through the machine passes through four phases before its answer is output at the root. An instruction is input to the machine at the root PE and is in phase 1 while its bubbles are flowing down the tree. During this phase the instruction bubbles are being compared to data elements in the tree in order to initialize each bubble's redundancy tag. The instruction makes the transition to phase 2 at the effective last level, where it bounces, and its bubbles proceed back up to the root. During this phase the PEs are updating the instruction bubbles' redundancy tags by comparing each instruction bubble going up the tree in its phase 2 to the instructions coming down the tree in their phase 3. An instruction begins phase 3 when it reaches the root, completing its first round-trip in the tree. In phase 3 it flows back down the tree and executes exactly as in the original scheme. The response from a PE is always sent to its right child, and its parent's response is sent to the PE's left child (even if that response is null). When the instruction reaches the effective last level in the tree it makes the transition to phase 4, where the bubbles again proceed up the tree to the root; this time all the answers to the instruction are combined to be output to the host computer as a single answer.

Each PE has communication ports which allows it to deal with the four different types of instruction bubbles implied by this scheme. We describe our protocol assuming communication ports are directed, and we denote each port relative to a particular PE p by $\alpha\beta$, where α is P (resp., L or R) if the port is connected to p's parent (resp., left child or right child), and β is the phase number of instructions that will come from or go to that port. For example, port L2 in some PE p will be connected to port P2 of p's left child. Because of the semantics of the phase numbers, it should be clear that, relative to a particular PE, P1, L2, R2, P3, L4, and R4 are input ports and L1, R1, P2, L3, R3, and P4 are output ports.

We denote the operation of a PE writing a value to a port (resp., reading a value from a port) by $P \leftarrow V$ (resp., $V \leftarrow P$), where V is the value and P is the port. If a PE reads from a port which has not been written upon since the last time it was read, then it receives a special value **nil** as the result of the read. We assume that the operations of reading and writing to a port are mutually exclusive and atomic. The PE execution-cycle procedure (which we describe below), and the fact that the dictionary machine operates synchronously, guarantees that no port is ever written to unless its previous value has been read by the PE at the other end.

In its execution loop the PE p processes the instruction bubbles in phase 4 first, then those in phase 3, and so on, to those in phase 1. In addition to these ports, a PE has four registers storing the instruction bubbles in each phase currently passing through p, which we denote as $b1$, $b2$, $b3$, and $b4$. Finally, the data record stored at p is denoted D. The details of the execution loop can be found in Fig. 3.

Although we have not shown it in the pseudocode, we assume that if a PE does not execute the body of an **if** statement, then it performs a number of no-operations so that it is at the same place in the code as the other PEs. In other words, our methods can be implemented in a SIMD environment using on-line definable masking vectors.

A PE on the effective last level performs the computation

```
Psuedo Code:                                  Comments:

while (TRUE) do
    /* phase 4 */
                                              Combine4 returns nil if L4 and R4 are nil,
        b4 $\leftarrow$ Combine4(L4,R4);      and combines L4 and R4 as in the original
                                              design otherwise.
        if (b4 $\neq$ nil) then
            P4 $\leftarrow$ b4;
        end if
    /* phase 3 */
        b3 $\leftarrow$ P3;
        if (b3 $\neq$ nil) then
            L3 $\leftarrow$ b3;
                                              Execute performs the operation b3 on the data
            R3 $\leftarrow$ Execute(b3,D);    record D just as in the original design. Note
                                              that the response is sent to the right child.
        end if
    /* phase 2 */
                                              Combine2 returns nil if L2 and R2 are nil,
        b2 $\leftarrow$ Combine2(L2,R2);      and combines the redundancy tags for L4 and
                                              R4 as described earlier.
        if (b2 $\neq$ nil) then
            if (b3 $\neq$ nil) then
                                              Tag2 compares the bubbles b2 and b3 and sets
                Tag2(b2,b3);                  the redundancy tag of b2 as was described
                                              earlier.
            end if
            P2 $\leftarrow$ b2;
        end if
    /* phase 1 */
        b1 $\leftarrow$ P1;
        if (b1 $\neq$ nil) then
                                              Tag1 compares the bubble b2 and the data
            Tag1(b1,D);                       record D and sets the redundancy tag of b2
                                              as was described earlier.
            L1 $\leftarrow$ b1;
            R1 $\leftarrow$ b1;
        end if
end while.
```

FIG. 3. The PE execution loop.

of Fig. 3 as if L1 (resp., R1, L3, R3) is connected to L2 (resp., R2, L4, R4). That is, it reads an instruction–response pair (i, b) from its parent, computes its response b' to i, and then performs the combining process as if it were an interior node PE which had just read b and b' from its children.

We assume that every other instruction being pipelined through the machine is either a no-operation or at least is not a content-modifying instruction. This assures us that we will never have a content-modifying instruction in phase 2 "pass" an instruction in phase 3 without the two instructions being compared in the *Tag2* procedure. The root processor can easily maintain this invariant. This idea, in fact, is already present in some of the previous designs (e.g., [1, 6]).

Note that with every other instruction cycle there is always some response being output at the root, so the machine is pipelining responses as fast as is possible to within a constant factor. In other words, the interval time for the machine is on the order of that of the original design. In addition, since we have instruction only flow down to the effective last level in the tree, the latency time is on the order of that of the original design.

In the next subsection we show how to maintain robustness while adding order-statistic capability to the machine.

3.2. *Implementing Order Statistics*

There are a number of useful data operations which use the ranks of data items as operands: selecting the median element, extracting the jth smallest element, or selecting the jth largest element, to name a few. In this section we show how to implement these types of rank-based queries. The first operation we describe is selecting the jth smallest data item in the dictionary tree. If we add a rank field to the [k, r] register of each PE, to store the rank of k, then we can easily add a new instruction **Select**(j), which returns the (k, r) pair with jth smallest k value.

A PE storing (k, r) can easily update the rank field for k by incrementing its rank for every nonredundant **Insert**(k', r') with $k' < k$ and decrementing it for every nonredundant **Delete**(k') with $k' < k$ that passes through the PE in its phase 3 (as a part of the *Execute* procedure). If the key k is really a tuple (k_1, k_2, . . . , k_d), then the rank of k can also be a tuple (r_1, r_2, . . . , r_d). The algorithms we present below consider the rank to be a single integer, but changing them for the case of a tuple of ranks should be obvious.

Adding the operation **Select**(j) does not corrupt the redundancy tagging mechanism presented in the previous section, because it does not alter the contents of the dictionary machine. However, if we include the instruction **Extract**(j), a generalization of **ExtractMin**( ) and **ExtractMax**( ) which simultaneously selects and deletes the (k, r) pair with the jth smallest k value, then the previous scheme for maintaining robustness is not enough. This is because an **Extract**(j) instruction can make redundant insertions nonredundant and can make nonredundant deletions redundant.

We can maintain the rank of the (k, r) pairs in the machine even with this new instruction by decrementing the rank of k by 1 for each nonredundant **Extract**(j) which is executed at the PE storing (k, r) such that the rank of k is more than j. Of course, if the rank of k equals j, then we delete (k, r) and (k, r) becomes the response of this instruction. Also, if the rank of k is less than j then the instruction has no effect on the rank of k.

It is not so easy, however, to determine which instructions are affected by **Extract**(j) instructions. For determining that in a robust environment we must be more clever in how we process instructions through the machine. Let i be some content-modifying instruction, using the key k, which is in its first round-trip moving up the machine (phase 2). We have already noted that as i is moving through the machine it exists as a number of instances (or "bubbles"). We define the *relative rank* of k for a bubble of i which is in its first round-trip moving up the tree and is currently at some PE p to be the rank of k if we were to complete executing all the instructions which have already passed through p in their second round-trip moving down the machine (phase 3). (See Fig. 4.) To rectify the problems introduced by **Extract**(j) we add a rank field to each bubble of every content-modifying instruction i. When i is in its phase 1 we will use this rank field to compute the relative rank of k for at least one of the bubbles of i, and when i is in its phase 2 we will update the rank fields so that at the end of i's first round-trip through the machine this field will store the actual rank of k (for i). The most important application of the rank field used in the instruction's first round-trip through the machine is that an instruction in its phase 2 can use this field to tell if it is affected by an **Extract**(j) moving down the machine in its phase 3. This rank field is also used

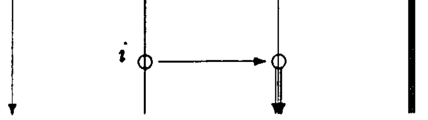

FIG. 4. The relative rank of i, currently in some PE p, is the rank it would have after executing all the instructions (indicated by a heavy line) which are past p in their second round-trip.

in the second round-trip to help compute the ranks of bubbles of instructions in their first round-trip.

When an instruction i is moving up the tree in its phase 2 it will be compared against those instructions coming down the tree in their phase 3 by examining their rank fields as well as their redundancy tags (in the procedure *Tag2*). We denote the rank field of a bubble $B = (i, b)$ by *B.rank*.

Initially, the rank of an **Insert**(k, r) instruction will be 1, **Delete**(k) and **Update**(k, r) instructions have an undefined rank (since they start out redundant), and **Extract**(j) has rank j. To maintain the value of *B.rank* we must modify the procedures *Tag1, Tag2,* and *Combine2* to maintain rank information as well as performing their original duties. We begin with the modifications for *Tag1*. Recall that this is the procedure for tagging instruction bubbles as they are moving down the tree in their phase 1, being compared to the data records in the tree. We add the following check to *Tag1:*

Let (i, b) be the bubble stored in $b1$, let k be the key value for this instruction, and let $D = (k', r')$ be the data record stored at the PE. If $k = k'$, then set the rank of $b1$ to the rank of D (note that this case should change the redundancy tag of $b1$ as well). Otherwise, if the bubble $b1$ is for a nonredundant **Insert**(k, r) instruction and $k' < k$, then set $b1.rank := \max(b1.rank, \text{rank}(k') + 1)$.

This rank is then sent down to the children PEs of this PE as a part of the message that is sent. We make an important observation about this rank-updating procedure in the following lemma:

LEMMA 3.1. *Let i be some content-modifying instruction with key value k. At least one bubble for i will contain the correct relative rank of k by the time the bubble reaches the effective last level of the tree, unless i is* **Delete**(k) *or* **Update**(k, r) *and all its bubbles are redundant when they reach the effective last level.*

Proof. Suppose i is a **Delete**(k) or **Update**(k, r) instruction. If there is a nonredundant bubble for i on the effective last level of the tree, then there was some (k, r') pair in the tree when this bubble passed by. In this case the correct relative rank for i is the rank of (k, r') when i "passed through" the PE p storing this value, since no instruction in its phase 3 could pass through p and then pass by i (because of the synchronous nature of the machine). That is, if there is an instruction i' which modifies the rank of k after the bubble for i passes through p, then i' has yet to encounter i (while the bubbles for i are in their phase 2). Thus, there is a bubble on the effective last level which stores the correct relative rank of i. This completes the proof for deletions and updates.

Suppose i is an **Insert**(k, r) instruction. If there are no redundant bubbles for i on the effective last level, then the correct relative rank of k at effective last level of the tree is one greater than the maximum rank of all (k', r') pairs, $k' < k$, which were in the tree when the bubbles of i flowed past them (by an argument similar to that of the previous paragraph). Since bubbles for the instruction i pass by every PE in the tree storing a (k', r') pair, then there is certainly some bubble for i which will pass through the PE storing the immediate predecessor of (k, r) (if there was one). Thus, if there are no redundant bubbles for i on the effective last level, then this bubble will store the correct relative rank of k for i. On the other hand, if there is a redundant bubble for i on the effective last level of the tree, then the relative rank for i is the rank of the (k, r') pair which caused that bubble to become redundant (again, by an argument similar to that of the previous paragraph). Since this is exactly the rank that is given to the redundant bubble in this case, there is a bubble which has the correct relative rank of k for i on the effective last level of the tree. ■

In combining instructions as they proceed up the tree in their phase 2 we take the rank of the combined bubble to be the maximum of the ranks of the bubbles coming from the children PEs. This, of course, amounts to a slight addition to the procedure *Combine2*.

We update the relative rank fields of the bubbles for an instruction i as it proceeds up the tree in its phase 2 so that the rank field of i will store the correct rank of the key value for i by the time it completes is phase 2. We must add some steps to the *Tag2* procedure to accomplish this. Let i_2 be the instruction for $b2$ and let i_3 be the instruction for $b3$. Let us assume that i_2 is an insertion, a deletion, or an update, since we need not update *b2.rank* if this is not the case. The first step we add to *Tag2* is to check if $b3$ changes $b2$ from a redundant operation to a nonredundant operation. If this happens, then we set *b2.rank* to *b3.rank*. For this reason, and reasons which will become apparent shortly, we do not maintain *b2.rank* if i_2 is a redundant **Delete**(k) or **Update**(k, r). We let k_2 denote the key value of i_2. Let us also assume that i_3 is a nonredundant rank-modifying instruction (i.e., **Insert**(k, r), **Delete**(k), or **Extract**(j)), since we need not update *b2.rank* if this is not the case. We have yet to specify how to update *b2.rank* if i_2 is an **Insert**(k, r) operation or i_2 is a nonredundant **Delete**(k) or **Update**(k, r) operation:

1. Suppose i_3 = **Insert**(k_3, r). If *b3.rank* < *b2.rank*, then increment *b2.rank* by 1. If *b3.rank* = *b2.rank*, then increment *b2.rank* by 1 only if $k_3 < k_2$. Otherwise, if *b3.rank* > *b2.rank*, then *b2.rank* is unchanged.

2. Suppose i_3 = **Delete**(k_3). If *b3.rank* < *b2.rank*, then decrement *b2.rank* by 1. Otherwise, *b2.rank* remains unchanged.

3. Suppose i_3 = **Extract**(j) (in which case *b3.rank* = j by definition). If *b3.rank* < *b2.rank*, then decrement *b2.rank* by 1. Otherwise, *b2.rank* remains unchanged.

We also add the following rule to the redundancy tagging rules of the *Tag2* procedure:

A redundant **Insert**(k, r) becomes nonredundant, or a nonredundant **Delete**(k) or **Update**(k, r) becomes redundant, if it encounters an **Extract**(j) instruction traveling down the machine in its second round-trip with j equal to the rank field for that instruction.

It should now be clear that the other reasons we do not care about the ranks of redundant deletions and updates are that (i) the **Extract**(j) only affects nonredundant deletions and updates, and (ii) redundant operations turn into no-operations in their phase 3.

The following lemma shows that the above scheme will maintain the correct relative rank of an instruction as it flows up the tree in its phase 2.

LEMMA 3.2. *Let i_2, k_2, i_3, b2, and b3 be as in the previous discussion. If b2.rank stores the correct relative rank of k_2 before b2 is compared to b3 (in the Tag2 procedure), then it will store the correct relative rank of k_2 after this comparison.*

Proof. As might be expected, the proof is by a case analysis. Suppose *b2.rank* stores the correct relative rank of k_2. By an argument similar to that of Lemma 3.1, if $b3$ changes $b2$ from being redundant to nonredundant, then the correct relative rank of $b2$ is the rank of $b3$. Thus, let us assume that $b3$ does not make $b2$ nonredundant.

Case 1. i_3 = **Insert**(k_3, r). If *b3.rank* < *b2.rank*, then, by the induction hypothesis, $k_3 < k_2$. Thus, it is correct that we increment *b2.rank* by 1. Suppose *b3.rank* = *b2.rank*; i.e., the relative rank of k_2 is the same as the actual rank of k_3. If $k_3 < k_2$, then the relative rank of k_2 is 1 more than that of k_3. So we are correct in incrementing *b2.rank* in this case. Suppose $k_3 = k_2$. If i_2 = **Insert**(k_2, r_2), then $b3$ makes $b2$ redundant and the relative rank of k_2 is unchanged. Otherwise, $b3$ forces $b2$ to become nonredundant. In either case, *b2.rank* is updated correctly. If $k_3 > k_2$, then the relative rank of k_2 is unchanged. In addition, if *b3.rank* > *b2.rank*, then *b2.rank* is unaffected by $b3$.

Case 2. i_3 = **Delete**(k_3). If *b3.rank* < *b2.rank*, then, by the induction hypothesis, $k_3 < k_2$. Thus, it is correct that we decrement *b2.rank* by 1. Suppose *b3.rank* = *b2.rank*. If i_2 = **Insert**(k_2, r_2), then by the induction hypothesis $k_2 = k_3$ and i_3 forces i_2 to become nonredundant; hence, *b2.rank* is updated correctly. If i_2 = **Delete**(k_2) or i_2 = **Update**(k_2, r_2), then by the induction hypothesis i_3 makes i_2 redundant and we do not care about *b2.rank* (the induction hypothesis is vacuously true in this case). Finally, if *b3.rank* > *b2.rank*, then i_3 does not change the relative rank of i_2.

Case 3. i_3 = **Extract**(j). If j < *b2.rank*, then it is clearly correct that we decrement *b2.rank* by 1. Suppose j = *b2.rank*. Then i_3 changes the redundancy tag of i_2, and either we do not care about *b2.rank* or (if i_2 = **Insert**(k_2, r_2)) we correctly set *b2.rank* to j. Finally, if j > *b2.rank*, then the relative rank of i_2 is unaffected by i_3. This completes the proof. ■

The following immediately follows from this lemma:

COROLLARY 3.3. *If a content-modifying instruction completes its phase 2 and is nonredundant, then its rank field stores the actual rank of the* (k, r) *pair it is to affect.*

Note that the only time the **Extract**(j) operation can be redundant is if there are less than j elements being stored in the machine when the instruction flows through. But, as should be apparent by the way **Extract**(j) interacts with other instructions and the (k, r) pairs in tree, the fact that $n < j$ essentially means the instruction is a no-operation. Thus, we conclude this subsection with the following theorem:

THEOREM 3.4. *An instruction is correctly tagged as redundant or nonredundant by the time it is finished traveling through the tree for the first time, even in the presence of* **Extract**(j) *instructions.*

Proof. We have already established the theorem for the basic three content-modifying instructions. So we need to prove that an instruction is correctly tagged in the presence of **Extract**(j) instructions. From the previous lemma, for an instruction i moving up the tree in its first round-trip, where i is a nonredundant **Insert**(k, r) or a redundant **Delete**(k) or **Update**(k, r), there is at least one bubble of i with the correct relative rank of k. If i on its way up encounters an **Extract**(j) on its way down with j equal to its relative rank, then it is affected by this **Extract**(j). Note that the redundancy rule added above to *Tag2* for this instruction correctly handles this case. Since we have already established the theorem for the other content-modifying instructions, this completes the proof. ■

3.3. *Using the Value of n*

Another important piece of information that should be allowed to be included in machine instructions is n, the number of (k, r) pairs in the machine. This makes operations such as extracting the maximum element, selecting the median, or selecting the jth largest element possible. One might be tempted to require that every PE store the value of n to be able to implement these operations, but the only PE which needs to store the value of n is the root processor. Any instruction that uses the value of n as an argument can have the actual value substituted as soon as the instruction reaches the root after its first round-trip through the machine. This is because the root processor will store the correct value of n for that instruction, having processed all content-modifying instructions which precede it. For example, when an instruction such as **SelectMedian**( ) or **SelectMax**( ) reaches the root PE the root simply changes the instruction into the appropriate **Select**(j) instruction. Similarly, the root can convert an **ExtractMax**( ) or **ExtractMedian**( ) operation to an **Extract**(j) in this way, as well. After this change, the instruction proceeds in its second round-trip as before.

Received September 1, 1987; revised February 17, 1988

ACKNOWLEDGMENTS

We thank the referees for their helpful suggestions. In particular, we acknowledge their suggestions on ways of improving the rigor and correctness of the presentation.

REFERENCES

1. Atallah, M. J., and Kosaraju, S. R. A generalized dictionary machine for VLSI. *IEEE Trans. Comput.* **C-34,** 2 (1985), 151–155.
2. Bentley, J. L., and Kung, H. T. A tree machine for searching problems. *Proc. 1979 IEEE International Conference on Parallel Processing,* pp. 257–266.
3. Bonuccelli, M. A., Lodi, E., Luccio, F., Maestrini, P., and Pagli, L. A VLSI tree machine for relational data bases. *Proc. 10th ACM/IEEE International Symposium on Computer Architecture,* 1983, pp. 67–73.
4. Fisher, A. L. Dictionary machines with a small number of processors. *Proc. 11th ACM/IEEE International Symposium on Computer Architecture,* 1984, pp. 151–156.
5. Fredman, M. L. A lower bound on the complexity of orthogonal range queries. *J. ACM* **28,** (1981), 696–706.
6. Ottmann, T. A., Rosenberg, A. L., and Stockmeyer, L. J. A dictionary machine (for VLSI). *IEEE Trans. Comput.* **C-31,** 9 (1982), 892–897.
7. Shemer, J., and Neches, P. The genesis of a database computer. *Computer* **17,** 11 (1984), 42–56.
8. Somani, A. K., and Agarwal, V. K. An efficient unsorted VLSI dictionary machine. *IEEE Trans. Comput.* **C-34,** 9 (1985), 841–852.
9. Song, S. W. A highly concurrent tree machine for database applications. *Proc. 1980 IEEE International Conference on Parallel Processing,* pp. 259–268.
10. Willard, D. E., and Lueker, G. S. Adding range restriction capability to dynamic data structures. *J. ACM* **32,** 3 (1985), 597–617.

MICHAEL T. GOODRICH received the B.A. in mathematics and computer science from Calvin College, Grand Rapids, Michigan, in 1983 and the M.S. and Ph.D. in computer science from Purdue University, West Lafayette, Indiana, in 1985 and 1987, respectively. In July 1987 he joined the Department of Computer Science at Johns Hopkins University, Baltimore, Maryland, where he is currently an assistant professor. In 1988 he received a Research Initiation Award from the National Science Foundation. His research interests include computational geometry and the design and analysis of sequential and parallel algorithms. Dr. Goodrich is a member of the Association for Computing Machinery, including the Special Interest Groups on Automata and Computability Theory (SIGACT) and Computer Graphics (SIGGRAPH).

MIKHAIL J. ATALLAH received the B.E. in electrical engineering from the American University, Beirut, Lebanon, in 1975 and the M.S.E. and Ph.D. in electrical engineering and computer science from Johns Hopkins University, Baltimore, Maryland, in 1980 and 1982, respectively. In August 1982 he joined the Department of Computer Science at Purdue University, West Lafayette, Indiana, where he is currently a professor. In 1985 he received a Presidential Young Investigator award from the National Science Foundation. His research interests include the design and analysis of algorithms, parallel computation, and computational geometry. Dr. Atallah is a member of the Association for Computing Machinery, the Institute of Electrical and Electronics Engineers, and the Society for Industrial and Applied Mathematics (SIAM). He serves on the Editorial Board of the *SIAM Journal on Computing.*