estimates of the values for P_a, BW, and U_p, if C_v is small. The second of these, the MC model, is the more complex model but, according to comparisons to simulations, provides accurate estimates of the values for P_a, BW, and U_p, for a wide range of C_v. The ER model requires the values of M, N, r, and $\overline{X}$ as inputs. The MC model requires, in addition, the value of $\overline{X^2}$. The explicit dependence of the MC model on $\overline{X^2}$ (and hence C_v) can be observed in (12). This was confirmed empirically; specifically, it was shown that BW decreases with increase in C_v. The fact that the second moment is an important feature of memory interference should not be completely unexpected as the behavior of similar systems, e.g., networks of queues, also depend on the variance of underlying stochastic processes (in the case of queues, it is the variances of the interarrival time and service time).

ACKNOWLEDGMENT

The authors gratefully acknowledge comments and suggestions made by B. A. Makrucki.

REFERENCES

[1] C. E. Skinner and J. R. Asher, "Effects of storage contention on system performance," *IBM Syst. J.*, vol. 8, no. 4, pp. 319–333, 1969.
[2] W. D. Strecker, "Analysis of the instruction execution rate in certain computer structures," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1970.
[3] D. P. Bhandarkar, "Analysis of memory interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-24, pp. 897–908, Sept. 1975.
[4] F. Baskett and A. J. Smith, "Interference in multiprocessor computer systems with interleaved memory," *Commun. ACM*, vol. 19, pp. 327–334, June 1976.
[5] C. H. Hoogendoorn, "A general model for memory interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-26, pp. 998–1005, Oct. 1977.
[6] J. S. Emer and E. S. Davidson, "Control store organization for multiple stream pipelined processors," in *Proc. 1978 Int. Conf. on Parallel Processing*, Aug. 1978, pp. 43–48.
[7] B. R. Rau, "Interleaved memory bandwidth in a model of a multiprocessors," *IEEE Trans. Comput.*, vol. C-28, pp. 678–681, Sept. 1979.
[8] J. H. Patel, "Processor–memory interconnections for multiprocessors," *IEEE Trans. Comput.*, vol. C-30, pp. 771–780, Oct. 1981.
[9] T. N. Mudge and B. A. Makrucki, "Probabilistic analysis of a crossbar switch," in *Proc. IEEE 9th Annu. Symp. Comput. Arch.*, Apr. 1982, pp. 311–319.
[10] D. W. L. Yen, J. H. Patel, and E. S. Davidson, "Memory interference in synchronous multiprocessor systems," *IEEE Trans. Comput.*, vol. C-31, pp. 1116–1121, Nov. 1982.
[11] L. N. Bhuyan and C. W. Lee, "An interference analysis of interconnection networks," in *Proc. 1983 Int. Conf. on Parallel Processing*, Aug. 1983, pp. 2–9.
[12] F. A. Briggs and E. S. Davidson, "Organization of semiconductor memories for parallel-pipelined processors," *IEEE Trans. Comput.*, vol. C-26, pp. 162–169, Feb. 1977.
[13] M. A. Marsan and M. Gerla, "Markov models for multiple bus multiprocessor systems," *IEEE Trans. Comput.*, vol. C-31, pp. 239–248, Mar. 1982.
[14] I. H. Onyuksel and K. B. Irani, "A Markov queueing network model for performance evaluation of bus-deficient multiprocessor systems," in *Proc. 1983 Int. Conf. on Parallel Processing*, Aug. 1983, pp. 437–439.
[15] T. N. Mudge, J. P. Hayes, G. D. Buzzard, and D. C. Winsor, "Analysis of multiple-bus interconnection networks," in *Proc. 1984 Int. Conf. on Parallel Processing*, Aug. 1984, pp. 228–232.
[16] S. H. Fuller, "Performance evaluation," in *Introduction to Computer Architecture*, H. S. Stone, Ed. Chicago, IL: Science Research, 1975, pp. 474–546.
[17] D. P. Bhandarkar and S. H. Fuller, "Markov chain models for analyzing memory interference in multiprocessor computer systems," in *Proc. 1st Annu. Symp. Comput. Arch.*, Dec. 1973, pp. 1–6.
[18] J. H. Patel, "Analysis of multiprocessors with private cache memories," *IEEE Trans. Comput.*, vol. C-31, pp. 296–304, Apr. 1982.
[19] T. N. Mudge and H. B. Al-Sadoun, "Memory interference models with variable connection time," Comput. Res. Lab., Dep. Elec. Eng. Comput. Sci., Univ. Michigan, Ann Arbor, MI, Rep. CRL-TR-16-84, Mar. 1984.

An Efficient Implementation of Search Trees on $\lceil lg\ N + 1 \rceil$ Processors

MICHAEL J. CAREY AND CLARK D. THOMPSON

Abstract **— A scheme for maintaining a balanced search tree on $\lceil lg\ N + 1 \rceil$ parallel processors is described. The scheme is almost fully pipelined: $\lceil lg\ N + 1 \rceil/2$ search, insert, and delete operations may run concurrently. Each processor executes $O(1)$ instructions of a top-down 2–3–4 tree manipulation algorithm before passing the operation along to the next processor in the pipeline. Thus, the total delay per tree operation is $O(lg\ N)$, and one tree operation completes every $O(1)$ time units.**

Index Terms **— Algorithms for VLSI, dictionary search, pipelining, search trees, special-purpose architectures.**

I. INTRODUCTION

The problem of implementing a search tree on a digital computer has received intense study. If all the data in the tree will fit into main memory, the conventional approach is to maintain the data in the form of a balanced binary tree [1]. Using the balanced tree format, a uniprocessor can perform search tree operations such as *insert*, *delete*, *exact-match search*, and *range search* by executing just $O(lg\ N)$ instructions where N is the current number of entries in the tree.[1] In this correspondence, we show how to design a multiprocessor system that is $O(lg\ N)$ times as fast as the conventional approach. Our system can accept a new search tree operation once every $O(1)$ instruction times. This increased throughput makes our scheme attractive for the implementation of centralized databases handling a large volume of queries. A more complete presentation of our scheme is available as [8].

In brief, the idea is to use a linear array of $\lceil lg\ N + 1 \rceil$ processors to maintain a balanced tree structure for up to N items. Each item is assumed to consist of a primary key K and an uninterpreted data field $I(K)$. Our scheme handles the following operations on the data in the tree: insertions, deletions, exact-match searches, and range queries. Each operation completes after $O(lg\ N)$ delay where the unit of time is taken to be the instruction cycle time of an individual processor. The scheme is almost fully pipelined, allowing as many as $\lceil lg\ N + 1 \rceil/2$ operations to be at varying stages of execution at any point in time, so one operation can complete every $O(1)$ time units. The processors in the system operate independently, each executing its own instruction stream, so the system is an MIMD architecture.

Each of the processors in our linear array is furnished with a private memory unit. Processor P_1 has memory capable of storing a single tree node. Processor P_i, $2 \leq i \leq \lceil lg\ N + 1 \rceil$, has twice the amount of memory of its predecessor P_{i-1}. The last processor $P_{\lceil lg\ N+1 \rceil}$ must have memory sufficient to hold all of the data which are to be stored in the machine. Adding bidirectional communication paths between the processors, the resulting machine architecture is shown for $N = 16$ in Fig. 1. (Also shown is an example of tree storage layout in our scheme.) This is essentially the same architecture used by Armstrong [3] and by Tanaka, Nozaka, and Masuyama [19]. In distinction to these earlier machines, however, our processors execute a top-down version of a 2–3–4 tree manipulation algorithm [13], in which each processor takes care of one level of the tree. By using this algorithm, our machine provides a richer set of database operations than either of its predecessors.

Manuscript received March 5, 1984; revised July 16, 1984. This work was supported by the National Science Foundation under Grant ECS-8110684, the Air Force Office of Scientific Research under Grant AFOSR-78-3596, the Naval Electronic Systems Command under Contract NESC-N00039-81-C-0569, and a California MICRO Fellowship.

M. J. Carey is with the Department of Computer Science, University of Wisconsin, Madison, WI 53706.
C. D. Thompson is with the Division of Computer Science, University of California, Berkeley, CA 94720.

[1] We follow the Knuthian practice of writing $lg\ N$ for $\log_2 N$.

0018-9340/84/1100-1038$01.00

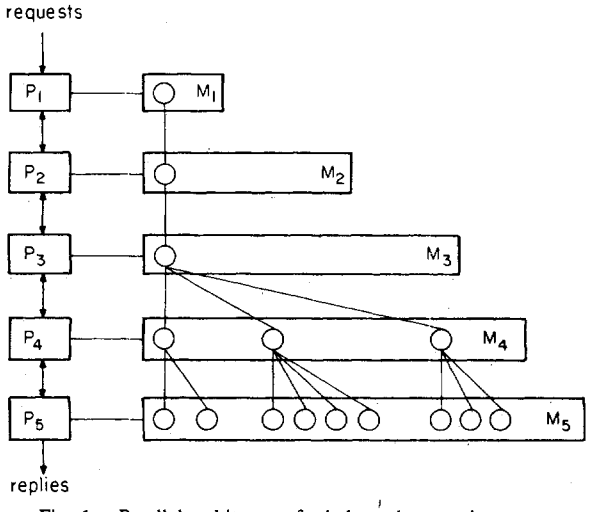

Fig. 1. Parallel architecture for balanced tree maintenance.

Also, only the last processor in our machine $P_{\lceil lg\, N+1\rceil}$ stores the actual data items $I(K)$.

II. Related Research

Our pipelined search tree implements many of the operations performed by the VLSI "search trees," "dictionary machines," and "database machines" of Leiserson [14], Bentley and Kung [6], Song [17], [18], Dohi, Suzuki, and Matsui [10], Ottman, Rosenberg, and Stockmeyer [15], Atallah and Kosaraju [4], Bonucelli, Lodi, Luccio, Maestrini, and Pagli [7], and Somani and Agarwal [16]. These "tree machines" are based on the use of $O(N)$ processors organized in tree-like configurations, and they can handle a large variety of search operations, including "partial match" queries [12], in $O(lg\, N)$ time. In contrast, our scheme requires only $O(lg\, N)$ processors but queries can only be for exact or range matches to the "primary key" field of each data item. In applications for which our scheme's operation set is sufficiently powerful, our pipelined search tree machine will be smaller, cheaper, and thus superior to an $O(N)$-processor tree machine.

Another related scheme is the $\lceil lg\, N + 1\rceil$-processor heapsort/search tree database system of Tanaka, Nozaka, and Masuyama [19]. Their idea is to use one $\lceil lg\, N + 1\rceil$-processor pipeline to heapsort a stream of records, and a second machine with similar structure to arrange the sorted stream into the form of a binary search tree. The main difference between their search tree and ours is that we provide on-line rebalancing, allowing insertions and deletions to run concurrently with search operations without unbalancing the tree.

A final related scheme was proposed as this correspondence was being revised for publication. Fisher [11] developed a scheme for maintaining a "trie" on a pipeline of l processors where l is the length of the longest item to be stored and each processor stores one byte of each item. Fisher's scheme (like ours) is superior to $O(N)$-processor tree machines for simple database applications, namely those in which the only operations are insert, delete, and exact search.

III. Processor Architecture

In the introduction, we defined our architecture as consisting of a linear array of $\lceil lg\, N + 1\rceil$ processors, each with its own memory. In stating our time bounds, we have assumed that each processor can execute one instruction in one unit of time. Before proceeding with the description of our tree maintenance algorithms, we briefly describe the capabilities required for the processors in our scheme.

Memory: Each word of data in the processors and memories of our scheme consists of w bits where w is large enough to represent either a key and two $\lceil lg\, N + 1\rceil$-bit pointers, or else a key, an uninterpreted data field, and one $\lceil lg\, N + 1\rceil$-bit pointer. A pointer can refer to one of N different memory words, or to *nil*. Processor P_i, $1 \le i \le \lceil lg\, N + 1\rceil$, has 2^{i-1} words of storage in its local memory. It has $O(1)$ internal registers, some of which are loaded with predefined constants. Finally, it has its own read-only control store with $O(1)$ instructions.

Instruction Set: The instruction set contains register–register ADD, SUBTRACT, and MOVE instructions, a BRANCH-ON-ZERO(reg, addr) instruction, instructions to READ and WRITE into each of the N (or fewer) locations in each processor's local memory, and SEND and RECEIVE instructions to communicate with the processor's nearest neighbors. For the two processors on the "ends" of the linear array, the SEND and RECEIVE instructions are used to communicate with the outside world. All instructions except RECEIVE are fetched, interpreted, and executed in unit time. An execution of RECEIVE, on the other hand, is not complete until a message is received from the specified processor. This can take an arbitrarily long amount of time.

IV. Operations for Tree Maintenance

In this section, the pipelined 2–3–4 tree manipulation operations are described. We remind the reader that a 2–3–4 tree is a balanced search tree where two, three, or four pointers (and one, two, or three search keys) appear in each internal (index) node, and all data items appear in external (leaf) nodes [13]. The tree manipulations are simple variants of the algorithms for the more common 2–3 trees and B^+ trees [1], [5], [9]. The advantage of 2–3–4 trees over 2–3 trees is that the manipulations can be performed in a top-down fashion [2], [13]. Top-down operations are ideal for our architecture, as they make pipelined operation both possible and simple. Since 2–3–4 tree manipulations have been described elsewhere, our description will be informal, with actual pointer and key manipulations omitted.

A. Searching

The SEARCH operation for the parallel 2–3–4 tree scheme is a simple pipelined version of normal B^+ tree searching. Hence, when processor P_i receives a "SEARCH(key n, using pointer p)" message, it should do the following.

Case 1: P_i contains internal nodes ($i < \lceil lg\, N + 1\rceil$). Follow the pointer p to the appropriate index node in local memory. Use the key value n to select the appropriate pointer (p') to follow from here. Send SEARCH(n, p') to P_{i+1}.

Case 2: P_i contains data nodes ($i = \lceil lg\, N + 1\rceil$). Given key n and pointer p, see if pointer p points at a data node containing key n. If so, send the data to the outside world. If not, send out a message indicating that the desired data were not found.

B. Insertion

The INSERT operation for the parallel 2–3–4 tree scheme is a pipelined version of the top-down node-splitting insert algorithm of Guibas and Sedgewick [2], [13]. When the search encounters a 4-node, the transformation shown in Fig. 2 is applied, ensuring that future node splits will not cause upwardly propagating splits. Note that the insertion transformation results in an increase in the actual tree height when applied at the root node.[2] The figure depicts the transformation in terms of 2–3–4 trees, with optional pointers drawn in dotted lines and the search path pointer indicated via a small black square. Although the figure shows the insertion path as being the leftmost path, the transformation applies in the obvious way regardless of the path.

Hence, when processor P_i receives an "INSERT(key n, using pointer p)" message, it should do the following.

Case 1: P_i contains internal nodes ($i < \lceil lg\, N + 1\rceil$). P_i follows the pointer p to the appropriate index node in its local memory. It then uses the key value n to select the appropriate pointer p' to follow from here. Next it sends INSERT_TRANSFORM(p') to P_{i+1}. P_{i+1} will apply the insertion transformation if it is applicable, splitting the next node on the search path for key n, and sending INSERT_TRANSFORM_REPLY(m, np) to P_i. This reply informs P_i of the

[2]The root node is defined as the first 2-, 3-, or 4-node on the path going from P_1 to $P_{\lceil lg\, N+1\rceil}$.

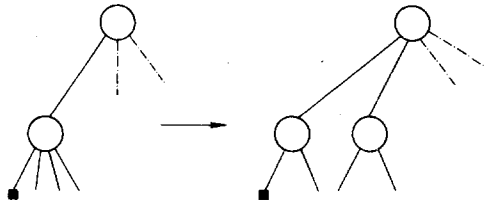

Fig. 2. Insertion transformation.

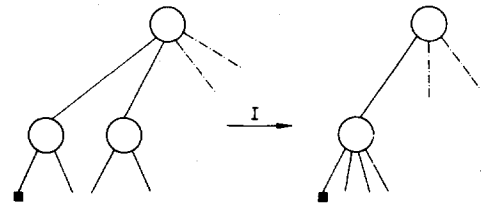

Fig. 3. Deletion transformation I.

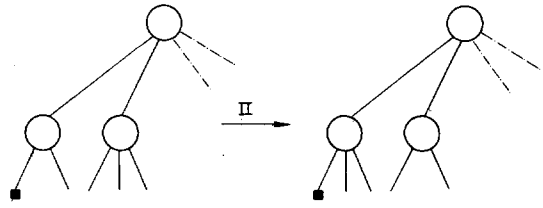

Fig. 4. Deletion transformation II.

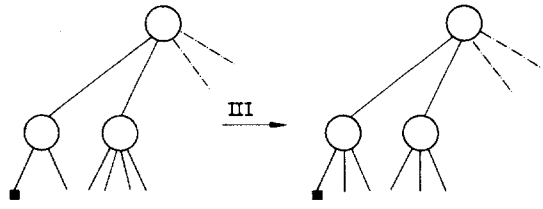

Fig. 5. Deletion transformation III.

new splitting key[3] m and new offspring pointer np resulting from a node split if one occurred. That is, if $np \neq nil$, P_i increases the size of its current index node p. If p is a 4-node, such an action would lead to the formation of a 5-node (an error). By induction, however, this is impossible since INSERT_TRANSFORM(p) was previously requested by processor P_{i-1}. (We consider the basis $i = 1$ of this induction in the next section, in the discussion of storage requirements.)

If processor P_i modifies the current node p in response to an INSERT_TRANSFORM_REPLY, it uses n once again to select the appropriate path p' to follow from here. Finally, P_i sends INSERT(n, p') to P_{i+1}.

Case 2: P_i contains data nodes ($i = \lceil lg\ N + 1 \rceil$). If the indicated data item is not already present and if there is room for another item in the tree, $P_{\lceil lg\ N+1 \rceil}$ installs it in a new data node. It then sends $P_{\lceil lg\ N \rceil}$ a pointer np to the new node in an INSERT_REPLY(np) message, and finally sends the outside world an acknowledgment. If there is no room for the item, or if the indicated data item is already present, $P_{\lceil lg\ N+1 \rceil}$ sends $P_{\lceil lg\ N \rceil}$ an INSERT_REPLY(nil) message, then sends the outside world an error response.

C. Deletion

The DELETE operation is a modified version of the Guibas and Sedgewick top-down deletion algorithm [13]. The modification is based on the observation that old keys may be used to guide searches in B^+ trees [9] since all predecessors of a deleted key are predecessors of its successor key, and all successors of its successor key are also successors of the deleted key. Thus, it is not necessary to delete the instances of a data item's key from the index portion of a 2–3–4 tree when deleting the item.

The basic idea of top-down deletion is that when a node with the minimum allowable number of keys is encountered, a transformation that adds keys must be performed to ensure that deletions cannot propagate upwards [2], [13]. No paper in the literature has described these transformations in terms of standard 2–3–4 trees or B^+ trees in a particularly comprehensible manner, so they will be described here in some detail. There are three such transformations, depicted in Figs. 3, 4, and 5 for 2–3–4 trees. In all cases, the transformations ensure that the next node on the search path has at least 3 descendents. As with Fig. 2, Figs. 3–5 depict the deletion path as being the leftmost path; the transformations can be generalized in the obvious way regardless of the path. Note that deletion transformation I (and only deletion transformation I) can result in a decrease in the tree height when applied at the root node.

Hence, when processor P_i receives a "DELETE(key n, using pointer p)" message, it should do the following.

Case 1: P_i contains internal nodes ($i < \lceil lg\ N + 1 \rceil$). Processor P_i follows the pointer p to the appropriate index node in local memory, using the key value n to select the appropriate pointer p' to follow from here. Then it sends DELETE_TRANSFORM(m, p', p'') to P_{i+1} where p'' is the adjacent path pointer[4] for p' and m is the splitting key for p' and p''. Processor P_{i+1} applies a deletion transformation if one is applicable, either merging the nodes indicated by p' and p'' or moving one of the offspring of the p'' node into the p' node (i.e., redistribution). Next, processor P_{i+1} sends DELETE_TRANSFORM_REPLY(m', np) back to P_i. This reply informs P_i of the new splitting key (m') resulting from a transformation (if one occurred). Also, the reply contains a pointer np to either nil, p', or p'', depending on which node (if any) was deleted by the transformation. Processor P_i uses this information to update its current index node.

Once its index node is updated, P_i again uses n to select the appropriate path p' to follow from here. (The path may be different if the current index node p was rearranged in response to the DELETE_TRANSFORM_REPLY message.) Finally, P_i sends DELETE(n, p') to P_{i+1}.

Case 2: P_i contains data nodes ($i = \lceil lg\ N + 1 \rceil$). If the indicated data item is present, $P_{\lceil lg\ N+1 \rceil}$ deletes its data node, sends $P_{\lceil lg\ N \rceil}$ a DELETE_REPLY(*no error*) message, and sends the outside world an acknowledgment. If the indicated data item is not present, $P_{\lceil lg\ N+1 \rceil}$ sends $P_{\lceil lg\ N \rceil}$ a DELETE_REPLY(*error*) message, then sends the outside world an error response.

D. Range Queries

Range queries can be accommodated with a slight additional amount of storage overhead. Immediately after a SEARCH(n) operation is initiated, any number of NEXT_UPTO(m) operations can be requested. When $P_{\lceil lg\ N+1 \rceil}$ receives a NEXT_UPTO(m) message, it looks for the smallest key k which is larger than the key of the previously outputted item. If there is no such item, or if $k > m$, $P_{\lceil lg\ N+1 \rceil}$ sends out an error indication. The external interface to our search tree should respond to this error indication by stopping the generation of NEXT_UPTO requests, thereby terminating the range query. About $\lceil lg\ N + 1 \rceil/2$ NEXT_UPTO requests will still be in the pipeline at the time the range query is terminated. A range query returning j items can thus be executed by means of a total of $j + \lceil lg\ N + 1 \rceil/2$ search tree operations.

Note that $P_{\lceil lg\ N+1 \rceil}$'s search for an item that is NEXT_UPTO(m) will be very time consuming if that item is not in the same node as the previous output. For this reason, additional pointer fields should be provided with each item (or with each node), forming the items into a doubly linked list or sequence set [9]. Thus, with two extra pointers per item, and one additional message type, our scheme will execute range queries in time proportional to $lg\ N$ plus the number of items in the range. For completeness, an analogously defined

[3] A "splitting key" is a key which P_i stores in its index node to guide future searches to one of two index nodes stored in P_{i+1}.

[4] The "adjacent path pointer" sent by P_i points to an index node in P_{i+1} which is an immediate neighbor of the index node in P_{i+1} on the current search path.

PREV_DOWNTO(m) command should probably be added to any realization of our scheme that implements the NEXT_UPTO(m) command. Once the doubly linked lists are in place, only a minimal amount of extra code is required in $P_{\lceil lg\, N+1\rceil}$ to implement PREV_DOWNTO.

V. Storage Requirements

In this section we prove that our scheme has enough memory to hold all possible 2–3–4 trees of N or fewer items. As defined in Section III, processor P_i, $1 \le i \le \lceil lg\, N + 1\rceil$, has 2^{i-1} words of storage in its local memory. Trees start out on processor $P_{\lceil lg\, N+1\rceil}$, growing upwards towards P_1 as items are inserted. Let $M_i(k)$ be the maximum number of words used by P_i when there are k items in the tree. Since a key, an item, and a pointer field fit into the word length specified in Section III, the last processor $P_{\lceil lg\, N+1\rceil}$ uses just as many words as there are items in the tree

$$M_{\lceil lg\, N+1\rceil}(k) = k.$$

The number of words used by processor $P_{\lceil lg\, N\rceil}$ is somewhat variable, depending on the number of 2-, 3-, and 4-nodes in the last processor. The worst case storage requirement is obtained when $P_{\lceil lg\, N+1\rceil}$ has no 3-nodes or 4-nodes

$$M_{\lceil lg\, N\rceil}(k) = \max\{1, \lfloor k/2\rfloor\}.$$

The "max" function is used to reflect the fact that there is always at least one occupied word on each level, even in an empty tree.

In general, $M_i(k)$ can be expressed in terms of $M_{i+1}(k)$

$$M_i(k) = \max\{1, \lfloor M_{i+1}(k)/2\rfloor\}, \qquad \text{for all } i < \lceil lg\, N + 1\rceil.$$

Solving the above recurrence, and noting that both $\lfloor \lfloor k/2^j\rfloor/2\rfloor$ and $\lfloor k/2^{j+1}\rfloor$ are equal to the binary value of k right-shifted by $j + 1$ bits, we find that

$$\begin{aligned} M_i(k) &= \max\{1, \lfloor k/2^{\lceil lg\, N+1\rceil - i}\rfloor\} \\ &= \max\{1, 2^{i-1}k/N\}. \end{aligned}$$

Evaluating $M_i(N)$, we find that P_i needs no more than 2^{i-1} words to store any N-item 2–3–4 tree. Furthermore, since $M_i(k)$ is monotonically nondecreasing in k, the architecture described in Section III is capable of storing any k-item 2–3–4 tree, $k \le N$. (If range query support is desired, an additional N words of memory will be required in processor $P_{\lceil lg\, N+1\rceil}$.)

VI. Conclusions

We have described a scheme for maintaining a 2–3–4 tree using a pipeline of $\lceil lg\, N + 1\rceil$ processors. Since the pipeline operates using a request/reply paradigm, half of the processors can be processing requests at any given point in time. The factor of two comes from the fact that until a processor P_i receives its reply from P_{i+1}, the keys and/or pointers in P_i may be incorrect. Thus, the level of concurrency in a $\lceil lg\, N + 1\rceil$ processor configuration executing an arbitrary sequence of SEARCH, INSERT, and DELETE commands is $\lceil lg\, N + 1\rceil/2$, as claimed in the Abstract. (A higher degree of concurrency—up to $\lceil lg\, N + 1\rceil$—could be obtained on a long string of SEARCH commands, as these do not involve INSERT_TRANSFORM or DELETE_TRANSFORM messages.) Since our scheme requires $O(lg\, N)$ time per tree operation, but allows $O(lg\, N)$ concurrency on the operations, one operation completes every $O(1)$ time units. This scheme could be a useful component for index maintenance in a machine architecture specialized for information storage and retrieval.

Acknowledgment

The authors gratefully acknowledge the assistance of R. McCord of Tolerant Systems and J. B——y, an anonymous referee with a distinctive style and a perspicacious mind.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
[2] J. Allchin, A. Keller, and G. Wiederhold, "FLASH: A language-independent, portable file access system," in *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1980.
[3] P. K. Armstrong, U.S. Patent 4 131 947, Dec. 26, 1978.
[4] M. J. Attallah and S. R. Kosaraju, "A generalized dictionary machine for *VLSI*," Dep. Elec. Eng. Comput. Sci., Johns Hopkins Univ., Baltimore, MD, Tech. Rep.; also, *IEEE Trans. Comput.*, 1984, to be published.
[5] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," *Acta Informatica*, vol. 1, no. 3, 1972.
[6] J. L. Bentley and H. T. Kung, "Two papers on a tree-structured parallel computer," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Rep. CMU-CS-79-142, 1979.
[7] M. A. Bonuccelli, E. Lodi, F. Lucio, P. Maestrini, and L. Pagli, "A VLSI tree machine for relational data bases," in *Proc. 10th Annu. ACM Int. Symp. on Comput. Arch.*, June 1983, pp. 67–73.
[8] M. J. Carey and C. D. Thompson, "An efficient implementation of search trees on O(log N) processors," Dep. Comput. Sci., Univ. California, Berkeley, CA, Rep. UCB/CSD 82/101, Nov. 1982.
[9] D. Comer, "The ubiquitous B-tree," *Comput. Surveys*, vol. 11, no. 2, June 1979.
[10] Y. Dohi, A. Suzuki, and N. Matsui, "Hardware sorter and its application to data base machine," in *Proc. 9th Annu. ACM Symp. on Comput. Arch., SIGARCH Newsletter*, vol. 10, no. 3, pp. 218–225, Apr. 1982.
[11] A. L. Fisher, "Dictionary machines with a small number of processors," in *Proc. 11th Annu. ACM Int. Symp. on Comput. Arch.*, June 1984.
[12] P. Flajolet and C. Puech, "Tree structures for partial match retrieval," in *Proc. 24th Annu. IEEE Comput. Soc. Symp. on Foundations of Comput. Sci.*, Nov. 1983, pp. 282–288.
[13] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annu. IEEE Comput. Soc. Symp. on Foundations of Comput. Sci.*, Oct. 1978, pp. 8–21.
[14] C. E. Leiserson, "Systolic priority queues," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Rep. CMU-CS-79-115, Apr. 1979.
[15] T. A. Ottmann, A. L. Rosenberg, and L. J. Stockmeyer, "A dictionary machine (for *VLSI*)," *IEEE Trans. Comput.*, vol. C-31, pp. 892–897, Sept. 1982.
[16] A. K. Somani and V. K. Agarwal, "An unsorted dictionary machine for *VLSI*," VLSI Design Lab., McGill Univ., Montreal, P.Q., Canada, 1983.
[17] S. W. Song, "A highly concurrent tree machine for database applications," in *Proc. 1980 IEEE Int. Conf. on Parallel Processing*, pp. 259–268. Aug. 1980.
[18] ——, "On a high-performance VLSI solution to database problems," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, CMU-CS-81-142, Aug. 1981.
[19] Y. Tanaka, Y. Nozaka, and A. Masuyama, "Pipeline searching and sorting modules as components of data flow database computer," in *Proc. Int. Fed. Inform. Processing*, Oct. 1980, pp. 427–432.

A Multiprocessor Architecture for Generating Fractal Surfaces

STEPHEN L. STEPOWAY, DAVID L. WELLS, AND GERALD R. KANE

***Abstract*—Fractal surfaces have recently been shown to be a useful model for generating images of terrain in computer graphics. Unfortunately, the generation of fractal images is very costly in CPU time. A multiprocessor architecture is described which takes advantage of the parallelism inherent in fractals to speed the generation of images. The performance of the processing array is analyzed along with the suitability of implementation in VLSI.**

***Index Terms*—Architecture, computer graphics, fractal surfaces, parallel processing, VLSI.**

Manuscript received March 1, 1984; revised July 14, 1984.

S. L. Stepoway and D. L. Wells are with the Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275.
G. R. Kane is with the Department of Electrical Engineering, Southern Methodist University, Dallas, TX 75275.

0018-9340/84/1100-1041$01.00