Dictionary Machine with Improved Performance

P. GOYAL AND T.S. NARAYANAN
Department of Computer Science, Concordia University, Montreal H3G 1M8, Canada

This paper proposes a dictionary machine which can handle redundant instructions on a modified semi X-tree structure in one pass. The proposed machine which is based on Leiserson's design operates with a pipeline interval of two steps. The cell complexity of the design is comparable to that of Leiserson's. Unlike other designs where cell complexity, pipeline interval and structure are compromised to get optimal response time, in this design, response time is compromised to get better pipeline interval, cell complexity and structure. The reason for taking this approach is presented in the paper.

Received February 1987

1. INTRODUCTION

The dictionary, an important data structure used in applications such as sorting and searching, symbol-table and index-table implementations. A dictionary consists of a set of data elements each of which is composed of a key-record pair (k, r), where k is the search key and r the associated record or a pointer to the record.

A dictionary is formally defined as a finite subset D of K x R, where K is the set of keys {k1, k2, ..., kN} and R the set of records {r1, r2, ..., rN}, such that for each key k there is at most one record r, with (k, r) in D. The pairs (ki, ri) are also called the entries of D. We define a function D: k → (k, r) where r is the associated record if k is present in the dictionary else is NULL.

A dictionary machine supports a set of instructions on its entries. A set of typical dictionary instructions is the following:

1. INSERT(k, r) : Either inserts or updates an entry.
2. DELETE(k) : Deletes an entry from the dictionary.
3. MEMBER(k) : Finds the existence of an entry.
4. XMIN : Returns the entry with the minimum key value.
5. XMAX : Returns the entry with the maximum key value.
6. NEAR(k) : Returns the entry which is 'closer' to the given key k.

INSERT(k, r) is said to be redundant when k is already in D, while DELETE(k) is redundant when k is not in D.

In addition to the above instructions, COMPRESS and NOP are also supported by some machines. These instructions differ from the other instructions because they neither alter the logical contents of the dictionary nor return any response.

7. COMPRESS(k) : Moves entries with key values > k to the location occupied by their predecessor.
8. NOP : Is used to adjust the timing between instructions.

There is scope to define many more possible instructions.

Wide variations in the implementation of the instructions exist. For example a redundant INSERT is treated either as an update or as an ignore. The instructions, MEMBER, XMIN, XMAX and NEAR, which return an entry as a response, can be defined to be destructive or non-destructive. A destructive instruction deletes the entry from the dictionary while a non-destructive instruction causes no change to its contents. The destructive or non-destructive nature of the implemented instructions and the type of redundant INSERT plays a significant part in the machine design.

Performance of a dictionary machine is measured in terms of the following parameters.

Response time. The elapsed time between initiation and completion of an instruction.

Pipeline interval. The minimum time required between initiations of any two consecutive user instructions (excluding COMPRESS and NOP).

Step. The maximum time taken to execute any specified instruction at a node.

Thruput. The average ratio of the user instructions to total instructions executed in a unit time.

Pass. Time taken for instruction to travel from the root to the logical last level and back to the root.

The maximum number of elements that may be stored in a dictionary machine is denoted by N, and the number of elements at any particular time is denoted by n.

This paper proposes a dictionary machine which can handle redundant instructions on a modified semi X-tree structure in one pass. The proposed machine which is based on Leiserson's design operates with a pipeline interval of two steps. The cell complexity of the design is comparable to that of Leiserson's. Unlike other designs where cell complexity, pipeline interval and structure are compromised to get optimal response time, in this design response time is compromised to get better pipeline interval, cell complexity and structure. A survey of the dictionary machine can be found in Ref. 7.

Section 2 presents a brief description of the Leiserson's dictionary machine. Proposed design is presented in Sections 3 and 4. Section 5 presents the correctness of execution and Section 6 presents the performance.

2. LEISERSON'S DESIGN

The design proposed by Leiserson had its elements stored in sorted order (unless otherwise stated, the order will be ascending) over a linear chain of cells formed by
The following are the design objectives.
1. To execute all instructions in one pass.
2. To achieve better through than formerly.\(^4\,5\,6\)
3. To achieve reasonably good storage utilisation.
4. To achieve all of the above objectives without increasing the cell complexity.

Optimal response time \((O(log n))\) is not one of the design objectives because of the following reasons. Machines designed for optimal response time\(^5\,6\) resort to either pass instructions\(^5\,6\) or X-tree structure.\(^4\) Designs\(^5\,6\) give optimal response time \(O(log n)\) only when \(n\) is very small. When \(n\) is greater than \(N + 1\), then the response time becomes worse than \(O(log N)\). Thus for a machine of 1024 cells, the optimum \(n\) is limited to 32, and for a machine of 4096 cells, the optimum \(n\) is only 64. Design\(^4\) suffers due to the X-tree structure, and needs \(2N\) cells for a machine of capacity \(N\). Also, because of cell complexity, the resultant large pipeline interval and the step size may lead to poor response time. Thus, it is felt that reducing the pipeline interval and executing all instruction in a single pass will indirectly contribute to the improvement in the response time.

3.1 Instructions

**INSERT**\((k, r)\)

An **INSERT**\((k, r)\) instruction makes all entries with a key value greater than \(k\) shift to their right. If the instruction is non-redundant, this does not cause any problem. If it is redundant, then it creates a hole to the right of the entry with the key value \(k\). The presence of holes in the structure will break the sequence of the keys and make subsequent **INSERT**s impossible, as illustrated in Fig. 3. An **INSERT**\((5, 2)\) cannot be executed because of the discontinuity between the entry \((4, x)\) and \((6, y)\) due to the presence of holes. The problem can be resolved by duplicating an entry instead of creating a hole. Whenever an **INSERT**\((k, r)\) is redundant, the cell with key value equal to \(k\) creates a replica of itself in the cell next to itself. This is done only if the next cell contains a key greater than \(k\), thus creating only one replica (hole) per instruction. Now any redundant insert will not break the key sequence.

The replication of an existing entry simulates redundant **INSERT** (ignore). Redundant **INSERT** (update) is implemented by replicating the input values associated with the instruction instead of the existing entry.

**DELETE**\((k)\)

The **DELETE**\((k)\) instruction performs a logical delete. An entry to be deleted, if it exists, is marked as being deleted by setting its \(r\) value to \(\#\); the key value is not altered and, thus, the deleted entry ('hole') does not break the logical key sequence because of the presence of the key values. Entries with key values greater than \(k\) are therefore not required to shift left. For a non-redundant delete, the created 'hole' is removed by using the COMPRESS instruction after a specified time. For a redundant **DELETE**\((k)\), no record will be affected and no action takes place.

**Handling Replicates**

Whenever an **INSERT**\((k, r)\) finds a set of deleted (marked) entries with the key value \(k\), all these marked entries are
changed to \((k, r)\) from \((k, *)\). \textit{INSERT}(k, r) also creates a replica, as described in \textit{INSERT}, and thus an extra duplicate entry is created. Whenever a \textit{DELETE}(k) finds a set of duplicate entries with the same key value \(k\), it marks all of them as deleted.

\textbf{COMPRESS}

\textit{COMPRESS} is used to remove the duplicate entries and deleted entries from the dictionary. Whenever an \textit{INSERT} or \textit{DELETE} instruction is executed, a \textit{NOP} instruction must be executed \(2 \log N\) steps later. This \textit{NOP} instruction will meet the response from the \textit{INSERT} or the \textit{DELETE} instruction which was entered 2 \(\log N\) steps ahead. The \textit{NOP} and the response from the \textit{INSERT} or the \textit{DELETE} are combined to form a \textit{COMPRESS} instruction at the root. This \textit{COMPRESS} instruction is broadcast to all the leaf nodes and when executed, removes the hole (duplicate or deleted entry) created by the \textit{INSERT} or the \textit{DELETE}. Response from non-redundant \textit{DELETE}(k) and redundant \textit{INSERT}(k) are combined with \textit{NOP} to form \textit{COMPRESS}(k), while response from redundant \textit{DELETE}(k) and non-redundant \textit{INSERT}(k) are ignored. All instructions except \textit{INSERT} and \textit{DELETE}, are non-destructive.

\textbf{MEMBER}(k)

The node which finds the match returns \(D(k)\).

\textbf{NEAR}(k)

This instruction is executed using the inner nodes of the tree. As a response each leaf node sends its value to its parent. The parent node in turn sends the potential entry among the two entries it has received to its parent, and so on. Finally the root node returns the entry corresponding to \textit{NEAR}(k) to the user.

\textbf{PRED}(k), \textit{SUCC}(k), \textit{XMAX} and \textit{XMIN}

\textit{SUCC}essor returns an entry which succeeds the specified key, while \textit{PRED}essor returns an entry which precedes the given key. A strategy similar to \textit{NEAR}(k) is used in executing the \textit{PRED}(k), \textit{SUCC}(k), \textit{XMAX} and \textit{XMIN} instructions.

All the user instructions except \textit{INSERT} and \textit{DELETE} are non-destructive (does not change the content of the dictionary).

The advantages of this design are

1. Leaf cells are as simple as Leiserson’s.
2. It can execute redundant instructions in one pass.
3. The pipeline interval is one step.

Disadvantage

1. It cannot execute destructive instructions in one pass.
2. Response time is of \(0(\log N)\).

\section{MACHINE DESIGN II – WITH DESTRUCTIVE \textit{XMIN}}

The design described in the previous section can execute all the nine instructions listed above in one pass. However, it cannot execute destructive instructions in one pass. A modification to the above design is described in this section that allows destructive \textit{XMAX} or \textit{XMIN} to be executed in one pass. For the remaining instructions (MEMBER, NEAR, SUCC and PRED) destructive execution is of lesser importance.

The proposed machine makes use of a modified semi-X tree structure (Fig. 4). The modified part of the semi-X tree is known as partition \(A\) and the other part is known as partition \(B\). Entries of the dictionary are stored only at the leaf nodes in ascending order of the key values. Partition \(A\) contains 2 \(\log N\) leaf nodes. All these nodes share a common register (reg) through a bus. This register is used to exchange the information that a given instruction is redundant (‘red’) or non-redundant (‘nr’). Cell design and programming code for partition \(A\) and partition \(B\) are different. Instructions in partition \(B\) are executed as described in the previous design. Description of the instructions in partition \(A\) can be found in the following paragraphs.

The leaf nodes of the partition \(A\) execute \textit{INSERT} and \textit{DELETE} instructions in two phases. In the first phase redundancy is ascertained and in the second phase, the instruction is executed. These two phases can be executed in one pipeline interval or one step. In addition, a \textit{NOP} must be entered exactly after 2 \(\log N\) steps to remove the hole.

\textbf{INSERT}(k, r)

In partition \(A\) the insert instruction is executed in two phases. In phase 1, if one of the leaf nodes in partition \(A\) contains \(D(k)\) then the instruction is redundant. The
node which contains the \( D(k) \) sets the reg to ‘red’. If no leaf node in \( A \) contains \( D(k) \), then the reg remains in the null state. If one of the leaf nodes finds that the instruction is not redundant then it sets the reg to ‘nr’. In phase 2, if the reg contains ‘nr’ then \( (k, r) \) is inserted, and all entries in \( A \) with their key values greater than \( k \) are shifted to their right by one position. If the reg contains ‘red’/‘null’ then no action is taken by any of the leaf nodes in \( A \). At the beginning of each instruction cycle reg sets itself to ‘null’.

**DELETE\( (k) \)**

Delete instruction is also executed in two phases in partition \( A \). In phase 1, if a leaf node in partition \( A \) contains \( D(k) \), then this node sets the reg to ‘nr’. If no leaf node in \( A \) contains \( D(k) \), then the reg retains the null value. In phase 2 all the leaf nodes in partition \( A \) with their key values less than \( k \) are shifted to their right by one position if reg is set to ‘nr’, otherwise no action takes place. Right shift is used to prevent holes getting into partition \( A \) among stored entries, and the reason for doing this is explained in Lemma 2. The response for a non-redundant DELETE from partition \( A \) is combined with NOP to form \( COMPRESS(k_{min}) \) at the root.

**XMIN**

The extract minimum (\( XMIN \)) instruction has the same execution code in partition \( A \) and in \( B \). \( D(k_{min}) \) will be returned to the root from the dictionary, and it is also deleted. All the entries are moved to their left by one node.

**MEMBER\( (k) \)**

Execution of the MEMBER\( (k) \) instruction is simple. The node which finds the match either in partition \( A \) or \( B \) returns \( D(k) \), if there is no match, nothing is returned. A null response is treated as an absence of \( D(k) \).

**NEAR\( (k) \)**

A similar strategy to the one used in the previous design is used for NEAR\( (k) \).

5. **CORRECTNESS OF EXECUTION**

The following Lemmas and Theorems are proved to show that the above design works correctly for the specified instruction set.

**Lemma 1**

At most one hole is created per instruction.

**Proof**

Proof of this is obvious from the instruction code for redundant INSERT and non-redundant DELETE [Appendix A].

**Theorem 1**

There can be at most 2 \( \log N \) holes from the leftmost leaf node to the last stored entry in the dictionary.

**Proof**

Only one hole is created per instruction (Lemma 1). A hole created by an instruction is removed after 2 \( \log N \) instructions. Thus, there cannot be more than 2 \( \log N \) holes.

As a result of Theorem 1, a dictionary with \( N \) leaf nodes can store a worst case maximum of \( N - 2 \log N \) entries. Comparing this with the design where \( 2N \) storage elements are required for \( N \) entries, this is quite an improvement.

**Corollary**

There is at least one entry in the first 2 \( \log N + 1 \) nodes whenever the dictionary is non-empty.

**Proof**

Assume that there are no entries in the first 2 \( \log N + 1 \) nodes. If the dictionary is not empty, this implies that there are more than 2 \( \log N \) holes in the dictionary. This is contrary to Theorem 1. Thus, there should be at least one entry in the first 2 \( \log N + 1 \) cells.

**Theorem 2**

The existence of holes in the structure does not prevent storing new entries in the right order (ascending).

**Proof**

Existence of holes does not break the key sequence due to the presence of the key values in the holes. Thus, entries can be inserted in proper order even when there are holes.

**Theorem 3**

All the instructions are executed in one pass.

**Proof**

Since INSERT and DELETE are allowed to create holes they are executed in one pass. Executing other instructions in one pass is trivial.

**Lemma 2**

There are no holes between the first stored entry and the last stored entry in the partition \( A \).

**Proof**

When a DELETE (non-redundant) is executed in the partition \( A \), the entries of \( A \) are compressed from the left purging the hole (the trivial entry \((-\infty, *)\) is replicated to its right). When the DELETE is redundant, it is ignored by all the entries (in \( A \) and \( B \)). An INSERT (redundant) does not create a hole in partition \( A \). Thus, no holes appear between the stored entries in \( A \).
Theorem 4
A node in the first $2 \log N + 1$ nodes can decide locally if it contains the minimum key entry or not.

Proof
From Lemma 2 it is clear that the entry with non-trivial key which sees a $(−\infty, *)$ as its left neighbour is the entry with the minimum key. No global information is required to check this.

Theorem 5
Thruput of the machine is equal to or more than 50%.

Proof
Thruput is minimum when only INSERTs and DELETEs are executed. This is due to the presence of NOP which follow an INSERT and a DELETE exactly after $2 \log N$ instructions. There is no loss of generality in assuming that the NOP is interleaved with INSERTs and DELETEs. Thus, every alternate instruction is a non-user instruction, leading to a thruput of 50%.

REFERENCES

APPENDIX A
Instruction Code for Design 1:

\begin{verbatim}
INSERT(k, r)
  If k_i = k and
    r_i != '*' and
    k_{i+1} > k then
      D(k_{i+1}) ← D(k_i) /*ignore*/
      Response ← 'red k'
    /*for update D(k_{i+1}) ← (k, r)*/
    /* D(k_i) ← (k, r) for all k_i = k */
  If k_i = k and
    r_i = '*' and
    k_{i+1} > k then
      Response ← 'nonred k'
      D(k_{i+1}) ← (k, r)
  If k_i = k and
    r_i = '*' then
      D(k_i) = (k, r)
      Response ← 'nonred k'
  If k_i < k < k_{i+1} then
    D(k_{i+1}) ← (k, r)
    Response ← 'nonred k'

DELETE(k)
  If k_i > k then
    D(k_{i+1}) ← D(k_i)
  If k_i = k then
    r_i ← '*'
    Response ← 'nonred k'
  If k_{i+1} < k < k_i then
    Response ← 'red k'

MEMBER(k)
  If k_i = k then
    then Response ← (k_i, r_i)
    else Response ← 'not exists'

XMIN
  Each intermediate level node gets two values from its children. The node sends the smaller of the two entries to its parent. The parent node receives two entries and repeats the same procedure and so on. Finally the root gets two entries and chooses the smaller one, which is the Minimum key entry.

  Similar procedure is followed in executing PRED, SUCC and XMAX.
\end{verbatim}
Instructions for Design 2:

**INSERT(k, r)**

In partition A:

phase 1: If \( k_i = k \) then set reg to 'red'
If \( k_i < k < k_{i+1} \) then set reg to 'nr'
phase 2: If reg = 'nr' and
\( k_i > k \) then
\( D(k_{i+1}) \leftarrow D(k_i) \)
If reg = 'nr' and
\( k_{i-1} < k \) and
\( k_i > k \) then
\( D(k_i) \leftarrow (k, r) \)
If reg = 'red' then
Response \( \leftarrow \) 'red k'
\( 1 \leq i \leq 2 \log N \)

In partition B:

phase 1: no action.
phase 2: same as in design 1.

**DELETE(k)**

In partition A:

phase 1: If \( k_i = k \) then set reg to 'nr'
phase 2: If reg = 'nr' and
\( k_i < k \) then
\( D(k_{i+1}) \leftarrow D(k_i) \)
Response \( \leftarrow \) 'comp all'
\( 1 \leq i \leq 2 \log N \)

In partition B:

phase 1: no action
phase 2: as in design 1.

**XMIN**

If \( k_{i-1} = -\infty \) then
Response \( \leftarrow D(k_i) \)
If \( k_{i-1} = \infty \) then
\( D(k_{i-1}) \leftarrow D(k_i) \)
\( 1 \leq i \leq N \)

Other instructions are executed exactly as they are in design 1.