Concurrent Maintenance of Data Structures in a Distributed Environment

F. B. BASTANI,* S. S. IYENGAR† AND I-LING YEN‡

*Department of Computer Science, University of Houston, University Park, Houston, Texas 77004, USA

†Department of Computer Science, Louisiana State University, Baton Rouge, Louisiana 70803, USA

‡Valid Logic, Inc., 2820 Orchard Parkway, San Jose, CA 95134, USA

We consider a distributed system consisting of a collection of clients and servers in which each server runs on a processor dedicated to it. Each server can be viewed as an instance of an abstract data-type module that provides a set of facilities for its clients. In such systems it may be possible to improve the performance of a server by scheduling its housekeeping activities to occur during periods when the server is waiting for requests from clients or is transmitting a response to a client. We consider the case where the client requests are handled by a foreground process while the maintenance tasks are performed by one or more background processes. We illustrate how the code for these processes can be developed in a stepwise manner, proceeding from coarse-grained concurrency to fine-grained concurrency. This approach is augmented with the use of rely/guarantee conditions which serve to simplify the proof of non-interference. The method is illustrated using a search-table abstraction.

Received November 1986, revised July 1987

1. INTRODUCTION

A popular model of a distributed system is to structure it as a set of clients that request services from one or more servers. Each server is basically an instance of an abstract data-type module that provides a collection of functions that are invoked by the clients. When implementing such servers, it often happens that the choice of the data structure permits efficient implementation of a set of operations at the expense of other operations. For example, consider a directory server that is implemented using a perfectly balanced binary search tree. In this case it is efficient to look up a name in the directory. However, adding a new name or removing an existing name is expensive due to the cost of rebalancing the tree after such operations.

It has been observed that the performance of such systems *may* be improved by having one or more additional processes that are responsible for maintaining the data structure of the server, i.e. converting it into an efficient one.[11,12] These processes are called maintenance processes and generally run as background (low-priority) processes. Clients interact with a foreground (high-priority) process that handles their requests. The foreground process operates on data structures that allow efficient implementation (up to a certain point!) of *all* the operations of the server. For example, the foreground process for the above directory server may simply mark an item as 'deleted' in response to a delete request. It is up to the background processes to transform the data structure into an efficient one. Thus in our example a maintenance process tries to rebalance the tree whenever it is activated.

In this paper we first discuss a class of data structures, called multilevel data structures, which allows the maintenance processes to run on separate processors. Section 2 gives the definition of multilevel data structures and discusses their limitations and some performance issues. This definition is then generalised to include data structures that can be characterised by weak and strong invariants. Section 3 outlines a method of developing the code for the foreground and background processes by combining Dijkstra's 'coarse grained concurrency to fine grained concurrency' approach[6] with Jones' rely/guarantee conditions.[9] Section 4 briefly compares the performance of concurrent maintenance strategy with those of some other maintenance techniques. Finally, Section 5 summarises the paper and outlines some research issues.

2. MULTILEVEL DATA STRUCTURES

In this section we first review the use of background processing for improving the performance of hierarchical storage systems. Then we discuss multilevel data structures, namely data structures that consist of two or more different types of data structures having different performance characteristics. We also discuss the use of multiprocessors in implementing these data structures. Finally, we present a more abstract notion of multilevel data structures characterised by strong and weak invariants.

2.1 Concurrent maintenance

Consider the system shown in Fig. 1. It provides a database containing a set of records stored on secondary memory. These records can be inspected and possibly updated by a user. The user interacts with the foreground process by requesting that a specified operation be performed on a specified record. This process searches a directory and, if necessary, fetches the desired record from secondary memory. However, it does not write back modified records. Instead, it stores these records in primary memory, leaving the task of moving them to secondary memory to a background process. As long as there is space in the primary memory, the user sees a faster response time than a system in which updated records are immediately written back.

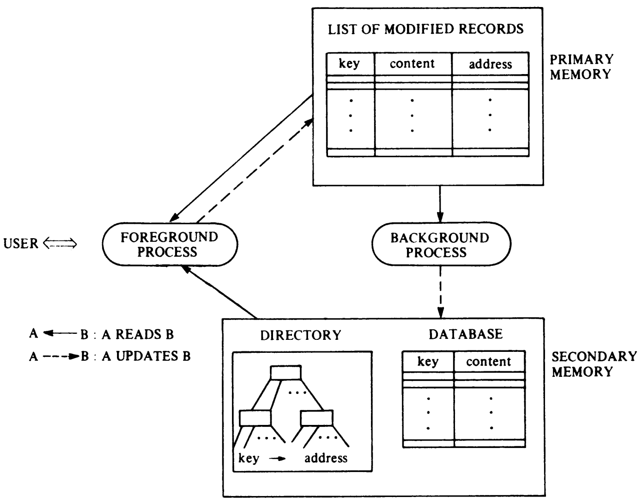


Figure 1. A multilevel storage system.

2.2 Multilevel data structures

The notion of multilevel data structures is based on a novel view of what a data organisation is. We view this generalisation by selecting data structures which have different performance features. For example, suppose that we have to implement a directory server or a name server. Then, one possibility is to select a sorted array for fast lookup and a queue for fast insertion. Also, a tag is associated with each item. It takes on the values 'alive' and 'dead'. This is used for achieving efficient deletion. The background process removes records which are marked 'dead' and, also, moves live records from the queue to the sorted array. This is shown in Fig. 2(*a*). In this case the queue corresponds to primary memory (short-term storage), while the sorted array corresponds to secondary memory (long-term storage). It should be emphasised that both of these data structures can reside at the *same* level in the storage hierarchy, i.e. the queue and the array can both be stored in primary memory. Another possibility is to select a binary search tree for short-term storage instead of a queue as shown in Fig. 2(*b*). The combination in Fig. 2(*b*) is better than simply using a binary search tree since the sorted array has an average lookup cost of $\log_2(n)$ while the tree has an average lookup cost of $1.44 * \log_2(n)$.

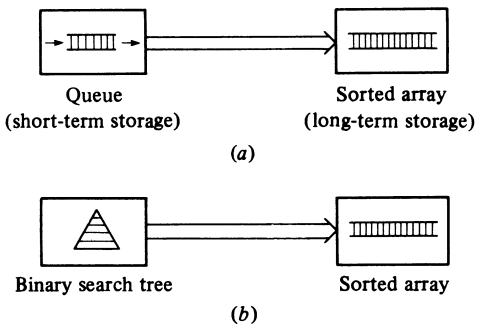


Figure 2. Examples of multilevel data structures.

Let $S(D)$ be some measure of the amount of information stored in a data structure D. For example, if D is any of the data structures shown in Fig. 2, then $S(D)$ could be the number of keys stored in that data structure.

Definition

A multilevel data structure D consists of a sequence of data structures $D_1, D_2, \ldots, D_n$, with the following properties: (*a*) $S(D_i)$ can be decreased by increasing $S(D_{i+1})$ without changing the information contained in D, (*b*) decreasing $S(D_i)$ by increasing $S(D_{i+1})$ results in an improvement in the performance of D, and (*c*) restructuring of D_i does not require restructuring of D_j, $j \neq i$. For example, in Fig. 2(*b*) the size of the binary tree can be decreased by moving some keys from the tree to the array. This change (*a*) does not affect the information contained in the combined data structure, (*b*) improves the performance of the combined data structure; moreover, (*c*) the binary tree can be restructured (e.g. balanced) without having to modify the array, and vice versa.

Multilevel data structures are limited to the implementation of abstract data types which deal with aggregation of items such as sets, bags and search tables. However, this class includes a large number of data-structure combinations such as queue combined with binary search tree, sorted array, hash table, balanced binary tree, B-tree, etc.

An important feature is that it is possible to use multiprocessors in implementing these data structures. For example, one processor can remove deleted slots from the queue, another from the array, while a third processor moves items from the queue to the array. The foreground processor can deal with client requests. This architecture is shown in Fig. 3, where the queue is stored in memory bank M 1 which is shared by processors F, B 1 and B 2, and the array is contained in memory bank M 2, which is shared by F, B 2 and B 3.

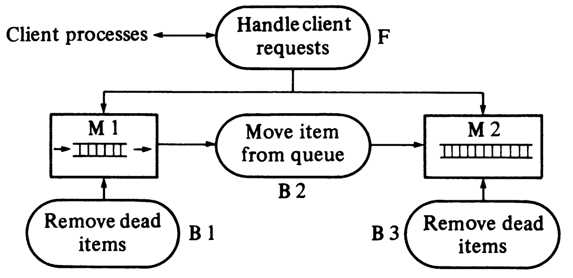
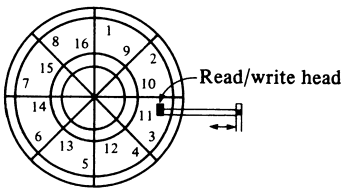


Figure 3. Multiprocessor implementation of multilevel data structures.

In addition to synchronisation problems, care must be taken in implementing such systems due to performance considerations. The basic requirement is that the background processes should not slow down the foreground process. First, this means that there should not be too much delay between the time a client request is received and the time the foreground process starts processing it. This requires that the indivisible actions of the background process be as brief (fine-grained) as possible and that the overhead associated with providing indivisible actions be relatively small when compared with the time required for processing a client's request. (Section 3 gives examples of coarse-grained and fine-grained indivisible actions.) However, fine-grained actions often result in less efficient code for the background process, since it must make fewer assumptions about the state of the data structure. Thus the background process may not be able to clean up the data structure fast enough, so that gradually the performance of the system deteriorates.

Even apparently efficient background processes can entail inefficiencies when disc resident data structures are considered. In these cases we must avoid hidden interferences which can arise due to the movements of the disc arm. For example, consider the case illustrated in Fig. 4.



```
repeat
  parallel:
    background: write back 1,2,3,4,5,6,7,8
    foreground: update 9,10,11,12,13,14,15,16
  -------------------------
  parallel:
    background: write back 9,10,11,12,13,14,15,16,
    foreground: update 1,2,3,4,5,6,7,8
forever
```

Figure 4. Interference due to disc resident data structures.

Assume that (1) there are 16 data records distributed evenly over the two extreme tracks of the disc, (2) there is space for only 8 records in primary memory, and (3) the client repeatedly updates the records located at one extreme followed by those located at the other extreme. Then the disc arm can oscillate back and forth, greatly reducing the performance of the system due to excessive seek time. In this case, the system with just a foreground process which immediately writes back updated records has superior performance.

Another way of improving the performance of concurrent maintenance systems is to let the background processes update the data structure after a group of requests have been served and not after every request. For example, in the 'queue/sorted array' implementation of search tables, the background process can wait till the queue reaches a certain size. Then it can sort the queue and merge it with the array. The use of 'binary tree/sorted array' combination eliminates the need for sorting. This is shown in Figure 5.

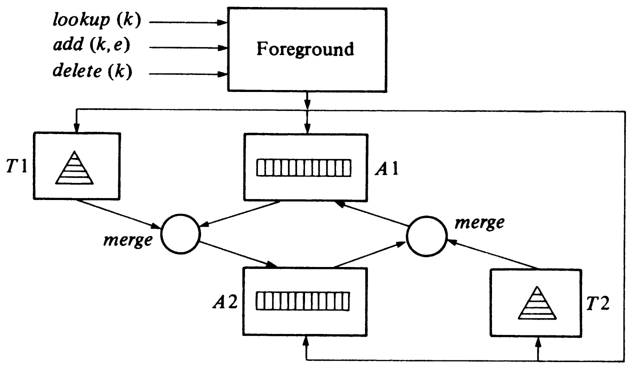


Figure 5. Multiprocessor implementation of the maintenance of a multilevel data structure for the search-table module.

Here we have two binary trees $T1$ and $T2$, and two sorted arrays $A1$ and $A2$. In phase 1 the foreground process appends new nodes to $T2$, while a background process merges $A1$ and $T1$ into $A2$. At the end of this phase, $A1$ and $T1$ are empty. Now phase 2 is started, in which the foreground process appends new items to $T1$ while the background process merges $A2$ and $T2$ into $A1$. At the end of this phase $A2$ and $T2$ are empty and the cycle is repeated by starting phase 1. This can be implemented using multiprocessors. An outline of the algorithm is as follows:

```
repeat
  MergeTreeAndArray($T1$, $A1$, $T2$, $A2$)
  MergeTreeAndArray($T2$, $A2$, $T1$, $A1$)
forever
where MergeTreeAndArray($T_a$, $A_a$, $T_b$, $A_b$) ≡
processor $T_a$: output live items in $T_a$ by doing an inorder
    traversal of $T_a$
processor $A_a$: output live items in $A_a$ by sequentially
    scanning $A_a$
processor $A_b$: merge the outputs of processors $T_a$ and
    $A_a$ into $A_b$
foreground processor: search $A_b$, then $A_a$ and $T_a$, then $T_b$
```

2.3 Generalisation

The performance of the multilevel storage system shown in Fig. 1 can be improved by maintaining a directory in primary memory, since this will eliminate searching disc records. However, condition (*c*) in the definition given in Section 2.3 is no longer satisfied, as modification of the information kept on the disc (e.g. moving a record to another location) may require changes in the information kept in primary memory. Thus this structure cannot be classified as a multilevel data structure even though it has most of their properties. Hence, to model concurrent maintenance for a broader class of data structures, we generalise the definition of multilevel data structures to include the above type of data structures.

Our approach is based on a more abstract notion of multilevel data structures characterised by weak and strong invariants. Each operation on the abstraction can be implemented efficiently provided that the strong invariant holds. The processing of an operation may falsify the strong invariant (though not the weak invariant). This can impair the performance. Hence, a maintenance process modifies the data structure so as to re-establish the strong invariant.

Definition

A general multilevel data structure D has associated with it a sequence of invariants $I_1, I_2, \ldots, I_n$, with the following properties: (*a*) D always satisfies I_1, (*b*) $I_n \rightarrow I_{n-1} \rightarrow \ldots \rightarrow I_1$, (*c*) if D does not satisfy I_j it can be made to satisfy I_j without changing the information contained in D; further, this modification will improve the performance of D. I_1 is the weak invariant while I_n is the strong invariant. The multilevel data structures considered in Section 2.2 are covered by this definition since we can write:

$$\forall_i,\ 1 \leqslant i \leqslant n,\ I_i \equiv \forall_j,\ 1 \leqslant j < i,\ S(D_j) \text{ is minimum.}$$

The generalised multilevel data structures can be used in the implementation of many types of server. However, they cannot be efficiently implemented using multiprocessor systems since more coordination is required between the foreground and background processes.

In this section we have defined multilevel data structures and discussed some implementation considerations. There are two important issues in studying programs used in such systems. The first issue is an approach for systematically developing these programs. This is addressed in Section 3. The second important issue is the evaluation of the performance of such systems. This is briefly discussed in Section 4. Additional details can be found in Refs. 2, 3 and 14.

3. DEVELOPMENT OF CONCURRENT MAINTENANCE ALGORITHMS

In this section we discuss a method of systematically developing programs for foreground and background processes and illustrate it using a 'queue/sorted array' data structure for a search-table abstract data type.

One of the earliest systematic methods is to start with coarse-grained concurrency and proceed to fine-grained concurrency.[6] However, this approach does not simplify the proof of correctness. Also, the addition of new operations for the abstract data type requires the proof of non-interference to be redeveloped even for processes which have not been modified. The general proof method is to identify an invariant and show that it is true after every indivisible action by any process.[4] Manber gives the algorithms and proofs for maintenance processes for a special form of binary search trees called *external trees*,[12] and in Manber and Ladner[13] a binary search structure is considered. In Ellis[7] a distributed version of an extendible hash file useful in distributed databases is developed by using a distributed data structure.

Here we combine the 'coarse-grained concurrency to fine-grained concurrency' method[6] with the stepwise development method proposed by Jones.[9] For the foreground and background processes we identify rely/guarantee conditions and goals (pre/post conditions) [see Ref. 9]. A process can assume that other concurrent processes will satisfy its rely condition. In turn, it must ensure that it maintains its guarantee condition, which must imply the rely condition of other processes. Further details appear in Ref. 9. The use of rely/guarantee conditions eliminates the need for extensive proof of non-interference when new operations are added to the server.

The stepwise development proceeds in the following coarse-grained concurrency to fine-grained concurrency sequence. (i) In case (A) only the foreground process is present – this is the sequential version. (ii) Case (B) extends case (A) to include a background process. Both the processes are non-interruptible, i.e. the foreground process completes a client request before yielding the processor and the background process completes a clean-up cycle before yielding the processor. (iii) Case (C) is an extension of case (B) wherein the background process is interruptible. (iv) In case (D), both the foreground as well as the background processes are interruptible. Since we do not wish to slow down the foreground process unnecessarily, the overheads for ensuring non-interference are mainly taken care of by the background process. That is, it is the responsibility of the background process to ensure that certain assertions are true at the moment it initiates an indivisible action.

In the following subsections we illustrate this technique by developing the code for foreground/background processes for a server which provides operations for storing and retrieving information associated with names (or keys), such as a name server or a directory server. Specifically, it provides the following functions: (i) '*LookUp*(k)', which returns the information corresponding to key k; if the key does not exist then it returns an error; (ii) '*add*(k, i)' for adding k to the database, and (iii) '*delete*(k)' for removing k from the database.

Section 3.1 considers the case where we have no background process. Section 3.2 considers a background process which completes a cycle before it can be interrupted. Section 3.3 permits the background process to be interrupted even within a cycle. Finally, Section 3.4 allows the foreground process to be interrupted also, though the non-interference overheads are primarily managed by the background process.

3.1 Case (A) – no background process

This is a conventional single-process case. We select the following data structure:

```
type
  node =
    record
      K: KeyType;
      I: info
    end;
var
  a: array [1..MaxArray] of node;
  n: 0..MaxArray := 0;
```

The program should satisfy the specification of the server. Also, we choose to keep the array sorted in order to speed-up the *LookUp* operation. Hence the program should ensure that the data structure satisfies the following invariant assertion:

$$\forall_i 1 \leqslant i < n : a[i].K < a[i+1].K$$

3.2 Case (B) – non-interruptible foreground, non-interruptible background

As discussed earlier, the response time of the server may be improved by performing the computations required for readjusting the array during idle periods. Case (B) can be developed from case (A) by using the following steps.

ab1. Modify the data structure to allow the foreground process to indicate clean-up tasks for the background process,
ab2. Update the strong invariant,
ab3. Specify the weak invariant,
ab4. Specify the rely condition for the foreground process; generally this asserts that the background process does not change the client-visible portion of the information contained in the data structure,
ab5. Modify the code for the foreground process,
ab6. Develop the code for the background process.

For our example, we have to select data structures for efficiently implementing the *add* and *delete* operations. *delete* can be implemented efficiently by introducing a status field associated with each node. If the value of the status field is *alive* then the node is in the data structure, otherwise it is not in the data structure. *add* can be implemented efficiently by using a queue-like data structure. If the item is not in the ordered array then it is added at the end of the queue. The queue impairs the performance of *LookUp* since it has to be searched linearly. This is also true if many items in the array are marked as deleted. For this reason the background process removes deleted entries and puts the items in the queue in their proper position in the array.

The data structure is:

```
type
  node =
    record
      $K$: KeyType;
      $I$: info;
      $S$: (alive, dead)
    end;
  nodePtr = ^node;
var
  $a$: array [1..MaxArray] of nodePtr;
  $n$: 0..MaxArray := 0;
var
  $q$: array [0..MaxQueue - 1] of nodePtr;
  $h$, $t$: 0..MaxInt := 0;
strong invariant
  (1) $h = t$ – the queue is empty.
  (2) $\forall_i: 1 \leqslant i < n$: $a[i]\hat{}.K < a[i+1]\hat{}.K$ – the array is
        strictly ordered.
  (3) $\forall_i: 1 \leqslant i \leqslant n: a[i]\hat{}.S = alive$ – all the items are
        marked as alive.
weak invariant
  (1) $h \leqslant t$ – the queue may not be empty.
  (2) $\forall_j: h < j \leqslant t$:
          not $\exists i$: $1 \leqslant i \leqslant n$: $a[i]\hat{}.K = q[j']\hat{}.K$ – key in the
            queue does not occur in the array.
          (We use the notation $x'$ to mean $x$ mod
            MaxQueue.)
          not $\exists i$: $h < i \leqslant t, i \neq j$: $q[i']\hat{}.K = q[j']\hat{}.K$ – a key
            occurs only once in the queue.
  (3) $\forall_i: 1 \leqslant i < n: a[i]\hat{}.K < a[i+1]\hat{}.K$ – the array is
        strictly ordered.
```

(2) and (3) together imply that all the keys in the queue and the array are distinct.

Foreground process. Its goal is to satisfy the specification and to maintain the weak invariant. It assumes that each indivisible action of the background process leaves the set of nodes with status = *alive* unchanged, though their position in the array/queue may be changed. This is the *rely* condition[9] of the foreground process. This can be stated formally as follows:

$$S(a)_{start(N+1)} \cup S(q)_{start(N+1)} = S(a)_{end(N)} \cup S(q)_{end(N)}$$

where

```
$S(a) \equiv sa := \varnothing$;
    for $i := 1$ to $n$ do
      if $a[i]\hat{}.S = alive$ then $sa := sa \cup \{a[i]\hat{}\}$;
    return $sa$;
```

and

```
$S(q) \equiv sq := \varnothing$;
    for $i := h+1$ to $t$ do
      if $q[i']\hat{}.S = alive$ then $sq := sq \cup \{q[i']\hat{}\}$;
    return $sq$;
```

Also, the notation $S_{start(N)}$ means the value of S at the start of indivisible action N of the process, while $S_{end(N)}$ means the value of S at the end of indivisible action N.

The implementation of the foreground process is similar to a typical one for case (A), except that we have to search both the queue and the array. An outline of the search procedure used by the foreground process is:
search the sorted array using binary search;
if *the element is found* **then** *return it*
else *sequentially search the queue*;
The code for the search procedure used by the foreground process is shown in Fig. 6
search array and queue (k):

```
  $l := 1$; $u := n$;
  while $l \leqslant u$ do
    begin $m := (l+u)$ div 2;
      if $a[m]\hat{}.K = k$ then exit
      else if $a[m]\hat{}.K < k$ then $l := m+1$
      else $u := m-1$
    end;
  if $l \leqslant u$ then return $a[m]$
  else begin $i := h+1$;
      while $i \leqslant t$ and then $q[i']\hat{}.K \neq k$ do $i := i+1$;
      if $i \leqslant t$ then return $q[i']$
      else return nil
    end;
```

Figure 6. Search procedure for non-interruptible foreground process for case (B).

Given that its rely condition holds, it is clear that the code for the foreground process is correct.

Background process. Its goal is to establish the strong invariant in a finite time if it is not interrupted by the foreground process. Also, it must guarantee that the rely condition of the foreground process is satisfied after every indivisible action.

An outline of the code for the background process is

```
loop forever
  ≪remove all deleted items from the array and the
      queue;
    if there is an item in the queue then
      move it into the array≫
```

Here ≪...≫ indicates an indivisible action. Specifically, for a uniprocessor '≪' may be interpreted as *disable interrupt*, while '≫' may be interpreted as *enable interrupt*. The complete code for the background process is shown in Fig. 7.

```
loop forever
  ≪shrink array;
  shrink queue;
  if $h < t$ then
    begin expand array (loc); transfer (loc)
    end≫

  shrink array: $i := 0$; $j := 0$;
    while $j < n$ do
      begin $j := j+1$;
        if $a[j]\hat{}.S = alive$ then
          begin $i := i+1$; $a[i] := a[j]$ end
    end;
  $n := i$;
shrink queue: while $h \leqslant t$ and then
                    $q[(h+1)']\hat{}.S = dead$ do $h := h+1$;

expand array (loc):
  $loc := n+1$;
  while $loc > 1$ and then $a[loc-1]\hat{}.K$
                              $> q[(h+1)']\hat{}.K$ do
    begin $a[loc] := a[loc-1]$; $loc := loc-1$
    end
  $n := n+1$;

transfer (loc): $a[loc] := q[(h+1)']$; $h := h+1$;
```

Figure 7. Implementation of non-interruptible background process for case (B).

It is straightforward to show that this code satisfies the *guarantee* condition of the background process – (i) *shrink array* retains all nodes which are alive, (ii) *shrink queue* increases h only if the node at the head of the queue is dead, (iii) *expand array* and *transfer* copy the item at the head of the queue into its proper place in the array and advance h, thereby removing it from the queue. Intermediate assertions can be used to show that the code meets the goal of the background process using conventional program proof methods.

Note. Since we have two interfering processes, the guarantee condition of one is the rely condition of the other, and vice versa. In this case the background process does not rely on any conditions other than the weak invariant. Hence its rely condition is **true**, so that the guarantee condition of the foreground process is also **true**.

Note. In order to simplify the programs, we assume that always $t\text{-}h < MaxQueue$ and $n < MaxArray$. Similar assumptions are made in Refs 4 and 6. Suitable interlocks can be provided for the general case.

3.3 Case (C) – non-interruptible foreground, interruptible background

This is the most important case in a multiprocessor system – the background process is active only when the processor dedicated to the abstract data-type module has no request from other processors. The following steps can be used in order to develop the code for case (C) from that for case (B).

bc1. The data structure and the strong invariant are unchanged.

bc2. Select indivisible actions of the background process.

bc3. The weak invariant is modified in order to allow the background process to be interrupted within its cycle; the amount of modification required depends on the coarseness of the indivisible actions selected in (bc2).

bc4. Specify the rely condition of the background process; this also depends on (bc2); for example, variables which are accessed by the background process outside indivisible actions cannot be modified by the foreground process.

bc5. Develop the code for the background process; iterate from (bc2) till a desired level of granularity of indivisible actions is achieved.

For our example, the invariant is almost the same as before, except that now we must have at least one duplicate key in the array; since the background process can be interrupted while it is adjusting (shrinking/expanding) the array.

strong invariant
Same as for case (B),
weak invariant
Condition 3 is replaced by:

(3.1) there exists at most one i, $1 \leqslant i < n$, such that $a[i]\hat{}.K = a[i+1]\hat{}.K$ – at most one duplicate occurs in the array.

(3.2) $a[i]\hat{}.K = a[i+1]\hat{}.K \rightarrow$ **not** $a[i]\hat{}.S = a[i+1]\hat{}.S = alive$ – at most one of the duplicates is alive.

(3.3) $\forall_i: 1 \leqslant i < n: a[i]\hat{}.K \leqslant a[i+1]\hat{}.K$ – the array is ordered.

Foreground process. Its goal and rely conditions are the same as in case (B). The search procedure used by it is also almost identical to the earlier one, except that now we have to consider weak invariants (3.1) and (3.2). Thus we have:

```
search the sorted array using binary search;
if the key is not found then sequentially search the queue
else begin check whether either the right or left neighbour
    of this index has the same key and is alive;
    if yes then return the pointer corresponding to
        neighbour,
    else return the pointer corresponding to index
  end;
```

The complete code appears in Figure 8.

```
search array and queue (k):
  $l := 1$; $u := n$:
  while $l \leqslant u$ do
    begin $m := (l+u)$ div 2;
      if $a[m]\hat{}.K = k$ then exit
      else if $a[m]\hat{}.K < k$ then $l := m+1$
      else $u := m-1$
    end;
  if $l \leqslant u$ then
    begin if $m < n$ and then ($a[m+1]\hat{}.K = k$ and
                    $a[m+1]\hat{}.S = alive$) then $m := m+1$
      else if $m > 1$ and then ($a[m-1]\hat{}.K = k$ and
                    $a[m-1]\hat{}.S = alive$) then $m := m-1$;
      return $a[m]$
    end
  else begin $i := h+1$;
      while $i \leqslant t$ and then $q[i']\hat{}.K \neq k$ do $i := i+1$;
      if $i \leqslant t$ then return $q[i']$
      else return nil
    end;
```

Figure 8. Search procedure for non-interruptible foreground process for case (C).

The only additional proof required here is to show that it satisfies the rely conditions of the background process (these are given below). Clearly $a[i]\hat{}.K$, h and n are not changed by it, since the only change it makes to the data structure (besides changes to the status and info fields) is to add an item at the tail of the queue.

Background process. Its goal is the same as in case (B). It relies on the following conditions (which must be guaranteed by the foreground process):

(1) $a[i]\hat{}.K$, h and n are not changed by other processes, i.e.

$\forall_i,\ 1 \leqslant i \leqslant n,\ a[i]\hat{}.K_{start(N+1)} = a[i]\hat{}.K_{end(N)}$,
$h_{start(N+1)} = h_{end(N)}$,
$n_{start(N+1)} = n_{end(N)}$;

(2) $\forall_i,\ h < i \leqslant t''$, $q[i']\hat{}.K$ is not changed by other processes where t'' is the value of t last observed by the background process, i.e.

$\forall_i, h < i \leqslant t_{end(N)},\ q[i']\hat{}.K_{start(N+1)} = q[i']\hat{}.K_{end(N)}$.

The outline of the code for the background process is also similar to that for case (B), except that now the indivisible actions are much more fine-grained. The code for the background process is shown in Figure 9.

It exits the **repeat** ... **until** loop provided it does not detect any dead node after a complete scan of the array. Otherwise it shrinks the array (as far as possible) and scans the array again. The rest is similar to the earlier code. The five indivisible actions (enclosed within $\lll\dots\ggg$) ensure that the weak invariant and the rely condition of the foreground process are satisfied. Specifically, these indivisible actions are used to ensure that only one key in a duplicate is alive whenever it shrinks or expands the array. Also, it evaluates the status of an item within an indivisible action before deleting it. In *shrink array* and *shrink queue* it exits the loop if it finds that the status is alive. However, this cannot be done in *expand array* and *transfer*. In these cases it uses the assertion that if $a[loc]\hat{}.S$ is *alive* then *loc* must be less than $n+1$ and $a[loc]\hat{}.K$ must be equal to $a[loc+1]\hat{}.K$, so that $a[loc+1]\hat{}.S$ must be equal to *dead*. Hence, it transfers $a[loc]$ to $a[loc+1]$ prior to changing $a[loc]$.

Note. As observed in Ref. 12, it is possible that the background process will never complete a cycle if requests repeatedly come for adding and then deleting the same key.

```
loop forever
  begin repeat
      shrink array (DeadFound)
    until not DeadFound;
    shrink queue;
    if $h < t$ then
      begin expand array (loc); transfer (loc)
      end
  end;

shrink array (DeadFound):
    $i := n$;
    while $i \geqslant 1$ and then $a[i]\hat{}.S \neq dead$ do $i := i-1$;
    if $i = 0$ then DeadFound: = false
    else begin DeadFound: = true;
        while $i < n$ do
          $\lll$ if $a[i]\hat{}.S = alive$ then $\ggg$ exit
            else begin $a[i] := a[i+1]$;
                         $a[i+1]\hat{}.S := dead \ggg$; $i := i+1$
              end;
        $\lll$if $a[n]\hat{}.S = dead$ then $n := n-1 \ggg$
      end;

shrink queue:
    while $h < t$ do
      $\lll$if $q[(h+1)']\hat{}.S = alive$ then$\ggg$ exit
      else $h := h+1 \ggg$;

expand array (loc):
    $loc := n+1$
    $a[loc]\hat{}.S := dead$;
    while $loc > 1$ and then
                  $a[loc-1]\hat{}.K > q[(h+1)']\hat{}.K$ do
      begin $\lll$if $a[loc]\hat{}.S = alive$ then $a[loc+1] := a[loc]$;
          if $loc = n+1$ then $n := n+1$;
          $a[loc] := a[loc-1]$;
          $a[loc-1]\hat{}.S := dead \ggg$;
        $loc := loc-1$
      end;

transfer (loc):
    $\lll$if $a[loc]\hat{}.S = alive$ then $a[loc+1] := a[loc]$;
    if $loc = n+1$ then $n := n+1$;
    $a[loc] := q[(h+1)']$;
    $h := h+1 \ggg$;
```

Figure 9. Implementation of interruptible background process for case (C).

3.4 Case (D) – interruptible foreground, interruptible background

Though the previous case is satisfactory for most situations, further flexibility is possible if we allow the background process to proceed even though the foreground process has not fully completed its operation. This may occur if one CPU can be dedicated to the foreground process and another to the background process. It is also useful for cases where the foreground process has to wait for some disc I/O operations when the data structure is disc resident.

This case can be implemented using *explicit* locks in addition to the indivisible actions used in the previous case. These locks exclude the background process from accessing areas of the data structure which may be accessed or modified by the foreground process during the *remaining* part of its current operation. In order to allow the foreground process to proceed at full speed, most of these locking actions are taken care of by the background process. That is, it is the responsibility of the background process to ensure that it is safe to access the data structure before doing so. Generally, the code for this case is very closely related to that for the previous case.

For our example, the array and queue data structures and invariants are as in case (C). Three new variables are used by the foreground process in order to block access to selected areas of the data structure: H indicates the position of the foreground process in the queue; L, U indicate the position of the foreground process in the array.

Foreground process. Its goal is to satisfy the specification and to maintain the weak invariant. It relies on three conditions, namely (i) the set of nodes with status = *alive* are unchanged by other processes, though their position in the queue/array may be changed; (ii) no node is transferred from the array to the queue; and (iii) there is no maintenance activity on portions of the data structure which it is searching. These are given by:

(i) $S(a)_{start(N+1)} \cup S(q)_{start(N+1)} = S(a)_{end(N)} \cup S(q)_{end(N)}$

(ii) $S(a)_{start(N+1)} \supseteq S(a)_{end(N)}$ and
$S(q)_{start(N+1)} \subseteq S(q)_{end(N)}$

(iii) $\forall_i, H < i \leqslant t$, $q[i']$ is not modified, i.e.,
$$q[i']_{start(N+1)} = q[i']_{end(N)};$$
$\forall_i, L \leqslant i \leqslant U$: $a[i]$ is not modified, i.e.,
$$a[i]_{start(N+1)} = a[i]_{end(N)}.$$

The code for the search procedure used by the foreground process appears in Figure 10. Note that after each *LookUp*(*k*), *add*(*k*, *i*), *delete*(*k*) operation we have to add $H := t; L := U+1$

```
search array and queue (k);
  ≪H := h≫; – locks access to the queue
  while H < t and then q[(H+1)′]^.K ≠ k do H := H+1;
  if H < t then return q[(H+1)′]
  else begin L := 1; ≪U := n≫;
                               – locks access to the array
      while L ≤ U do
        begin m := (L+U) div 2;
          if a[m]^.K = k then exit
          else if a[m]^.K < k then L := m+1
          else U := m−1
        end;
      if L > U then return nil
      else begin if m < n and then (a[m+1]^.K = k
                and a[m+1]^.S = alive) then m := m+1
                else if m > 1 and then (a[m−1]^.K = k
               and a[m−1]^.S = alive) then m := m−1;
                L := m; U := m;
                return a[m]
           end
  end
```

Figure 10. Search procedure for interruptible foreground process for case (D)

It is similar to that in Figure 8, except that the queue is searched first and that values are assigned to *H*, *L* and *U* to prevent the background process from modifying the portion of the data structure being accessed by the foreground process. (The queue is searched first because the foreground process relies on the condition that a node cannot be moved from the array to the queue, though it may be moved from the queue to the array.) Its correctness follows from that of Fig. 8 and the rely condition regarding *H*, *L* and *U*. The only new proof is to show that if m is set $m+1$ or $m-1$ (as the case may be) then we still have $L \leqslant m \leqslant U$.

Background process. Its goal and rely conditions are the same as in case (C). The code for the background process is shown in Fig. 11. We use the primitive:

```
shrink array (DeadFound):
  i := n;
  while i ≥ 1 and then a ≠ dead do i := i−1;
  if i = 0 then DeadFound := false
  else begin DeadFound := true;
      while i < n do
        when not (L ≤ i ≤ U or L ≤ i+1 ≤ U) do
          ≪if a[i]^.S = alive then≫ exit
          else begin a[i] := a[i+1];
                         a[i+1]^.S := dead≫; i := i+1
          end;
      when not L ≤ n ≤ U do
        ≪if a[n]^.S = dead then n := n−1≫
  end;

shrink queue:
  while h < t do
    when H ≠ h+1 do
      ≪if q[(h+1)′]^.S = alive then≫ exit
      else h := h+1≫;

expand array (loc):
  loc := n+1;
  α[loc]^.S := dead;
  while loc > 1 and then a[loc−1]^.K > q[(h+1)′]^.K do
    begin when not (L ≤ loc−1 ≤ U or L ≤ loc ≤ U
                                    or L ≤ loc+1 ≤ U) do
      ≪if a[loc]^.S = alive then a[loc+1] := a[loc];
      if loc = n+1 then n := n+1;
      a[loc] := a[loc−1];
      a[loc−1]^.S := dead≫;
      loc := loc−1
    end;

transfer(loc):
  when H ≠ h and not (L ≤ loc ≤ U or
                                  L ≤ loc+1 ≤ U) do
    ≪if a[loc]^.S = alive then a[loc+1] := a[loc];
    if loc = n+1 then n := n+1;
    a[loc] := q[(h+1)′];
    h := h+1≫;
```

Figure 11. Implementation of interruptible background process for case (D).

when B **do** $\ll A \gg$

to mean that indivisible action $\ll A \gg$ is executed only when B becomes true. We can implement this by using busy-waiting:

label: $\ll$**if not** B **then**$\gg$ **goto** *label* **else** $A \gg$

Busy-waiting is acceptable for the two scenarios we have considered, namely, a dedicated processor for the foreground process or when the foreground process is waiting for some events to occur. The correctness of the program then follows from that shown in Figure 9.

3.5 Comments

To recapitulate, the code for the foreground and background processes may be developed by systematically proceeding from coarse-grained concurrency to fine-grained concurrency. In addition, rely/guarantee conditions are used to simplify the proof of non-interference at each stage. To ensure that the semantics of the abstraction are satisfied, the maintenance process must guarantee that the 'rely' conditions of the foreground process are not violated. Similarly, every indivisible action of the foreground process must satisfy the 'rely' condition of the maintenance process.

We have used this approach for developing concurrent maintenance algorithms for strings,[18] equivalence relations, linear lists,[16] hash tables,[16] and a multiway tree data structure for a directory server.[5]

4. PERFORMANCE EVALUATION

In this section we briefly discuss the performance of various methods of maintaining data structures. Additional details appear in Refs. 2, 3 and 14.

As evident from the example in the previous section, the algorithms for cases (C) and (D) are significantly more complicated than those for cases (A) and (B). Hence they should be considered only if there is clear evidence that the use of such algorithms will improve the performance of the system. The average waiting time for a client request to be processed by the server is an important performance measure, which can be used to evaluate different data-structure maintenance policies.

Clearly, it is not appropriate to compare the performance of the algorithms for maintaining an ordered array (such as those developed in the previous section) with, say, the concurrent maintenance of a hash table or a tree. We assume that a particular data structure has been selected for implementing a server based on certain design considerations. The important performance question is what maintenance approach is suitable for that data structure.

In our study we have considered the following maintenance techniques.

(1) Case A 1: no background process along with the use of a standard form of the data structure; the foreground process performs all the maintenance tasks required by a client request *before* sending any response to the client,

(2) Case A 2: no background process, but perhaps a self-reorganising form of the data structure is used; this includes case A 1,

(3) Case B 1: separate maintenance process that is invoked after the completion of every foreground operation; all the maintenance tasks required by a client request are completed before processing a new request,

(4) Case B 3: separate maintenance process that is invoked after the completion of every $M (M \geqslant 1)$ foreground operation; this includes case B 1,

(5) Case C: concurrent maintenance process; the maintenance process runs whenever the foreground process is idle,

We have compared cases A 1, B 1 and C, both analytically and experimentally, for a single-client system such as the implementation of a distributed program using remote procedure calls. The results are as follows.

Case B 1 is always better than case A 1,

Case C is better than cases B 1 and A 1 when the load is below a certain value; the cross-over point increases as the proportion of look-up requests increases relative to update requests.

For multiple-client systems, such as a general network server, we have the following results:

If we assume that the foreground process has a higher priority than the background process, then case C is basically unstable, i.e. with probability 1 the system will eventually reach a state in which the response time is infinite; hence we only consider case C with scheduling policies which dynamically adjust the relative priorities of the foreground and maintenance processes,

Case B2 has the smallest average response time (M must be found experimentally),

Case C has the smallest variance and the smallest worst-case performance.

From these observations we conclude that concurrent maintenance may be suitable for real-time applications, especially if the operating system supports efficient synchronisation primitives. However, it should be noted that the algorithms for case C are significantly more complex than those for the other cases.

5. SUMMARY

In this paper we have considered the implementation of servers incorporating low-priority maintenance processes dedicated to housekeeping activities. We first defined multilevel data structures which are similar to hierarchical storage systems, and discussed their implementation using multiprocessor systems. Then we generalised these to include data structures which can be characterised by weak and strong invariants. These allow each operation of the server to be implemented efficiently, provided that the data structure is in a state which satisfies the strong invariant. It is the responsibility of the maintenance processes to bring the data structure to such a state. We discussed a stepwise development method for the special case where we have only one maintenance process by proceeding from coarse-grained concurrency to fine-grained concurrency.

We have also briefly discussed the performance of different maintenance strategies. (Details appear in Refs 2, 3 and 14.) The results indicate that concurrent maintenance is suitable for real-time applications.

Some research issues are (i) developing methods of deriving the rely/guarantee conditions from the strong/weak invariants since it is relatively straightforward to determine the latter, (ii) considering the case where we have multiple foreground processes (see Refs 10, 12 and 15).

Acknowledgements

The authors wish to thank Professors Abha Moitra and Bahaa-El-Din, and an anonymous referee, for their detailed comments and suggestions that have helped to enhance the clarity of the paper.

REFERENCES

1. F. B. Bastani, Performance improvement of abstractions through context dependent transformations. *IEEE Trans. Softw. Eng.* **SE-10** (1), 100–116 (1984).
2. F. B. Bastani, I. L. Yen, A. Moitra and S. S. Iyengar, Impact of parallel processing on software quality. *Proc. 1st Intl. Conf. SuperComp. Sys., St Petersburg, FL* (1985).
3. F. B. Bastani, W. Hilal and I. R. Chen, Performance analysis of concurrent maintenance policies for servers in a distributed environment. *Proc. FJCC '86, Dallas, TX* (1986).
4. M. Ben-Ari, Algorithms for on-the-fly garbage collection. *ACM Trans. Prog. Langs. and Sys.* **6** (3), 333–344 (1984).
5. I. R. Chen, A Distributed Directory Server in a UNIX Network Environment. *M.S. Thesis*, Department of Computer Science, University of Houston, University Park (1985).
6. E. W. Dijkstra *et al.*, On-the-fly garbage collection: an exercise in cooperation. *Comm. ACM* **21** (11), 966–975 (1978).
7. C. S. Ellis, Distributed data structures: A case study. *Proc. 5th Intl. Conf. Distr. Proc. Sys., Denver, CO*, 201–209 (1985).
8. T. Hickey and J. Cohen, Performance analysis of on-the-fly garbage collection. *Comm. ACM* **27** (11), 1143–1154 (1984).

9. C. B. Jones, Tentative steps towards a development method for interfering programs. *ACM Trans. Prog. Langs. and Sys.* **5** (4), 596–619 (1983).
10. L. Lamport, Specifying concurrent program modules, *ACM Trans. Langs. and Sys.* **5** (2), 190–222 (1983).
11. B. W. Lampson, Hints for computer system design, *IEEE Softw.* **1** (1), 11–28 (1984).
12. U. Manber, Concurrent maintenance of binary search trees. *IEEE Trans. Softw. Eng.* **SE-10** (6), 777–784 (1984).
13. U. Manber and R. E. Ladner, Concurrency control in a dynamic search structure. *ACM Trans. Database Sys.* **9** (3), 439–455 (1984).
14. A. Moitra, S. S. Iyengar, F. B. Bastani and I. L. Yen, Multilevel data structures: models and performance. *Tech. Rep.* to appear in *IEEE Trans. Softw. Eng.*
15. P. M. Schwarz and A. Z. Spector, Synchronizing shared abstract types. *ACM Trans. Comp. Sys.*
16. J.-E. Teng, An Experimental Evaluation of Maintenance Strategies for Servers in UNIX Local Area Network Environment, *M.S. Thesis*, Department of Computer Science, University of Houston, University Park (1986).
17. N. Wirth, *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, N.J. (1976).
18. I.-L. Yen, The Role of Parallel Processing in Application Programs, *M.S. Thesis*, Department of Computer Science, University of Houston, University Park (1985).

Announcements

28 NOVEMBER – 2 DECEMBER 1988

International Conference on Fifth Generation Computer Systems 1988, Tokyo, Japan

FGCS '88 is an international conference following on from the last conference, FGCS '84. The conference will be held for five days. The first two days will be mainly devoted to reports of the FGCS project's results, and the last three days to technical sessions for presentation of papers and related discussions.

The scope of technical sessions encompasses the technical aspects of new-generation computer systems which are particularly within the framework of **knowledge information processing, logic programming and parallel architectures.** This conference is intended to promote interaction among researchers in all disciplines related to fifth-generation computer technology. Of special interest are papers discussing the future direction or prospects of new-generation computing. The topics of interest include (but are not limited to) the following.

Program areas

Foundation
Formal semantics
Computation models
Theory of parallel computation
Automated reasoning
Foundation of AI
Prospect of new-generation computing
Social impact of FGCS

Architecture
Inference machines
Knowledge-base machines
Parallel architectures
AI architectures
VLSI architectures
Human–machine architectures

Software
Logic/functional/object-oriented programming
Parallel programming languages and methodologies
Program verification/debugging
Program analysis/transformation
Implementation techniques

Applications
Knowledge-based systems
Natural language understanding/machine translation
Real-time AI systems
Application of parallel systems
Games/simulation

Paper submission requirements

Authors should send six copies of manuscripts to:

Professor Hidehiko Tanaka, FGCS '88 Program Chairman, ICOT, Mita Kokusai Building 21F, 1-4-28 Mita, Minato-ku, Tokyo 108, Japan.

Papers to be received by 10 May 1988.

Papers are restricted to 20 double-spaced pages (about 5000 words) including figures. Each paper must contain a 200 to 250 word abstract. Papers must be written and presented in English.

Papers will be reviewed by international referees. Authors will be notified of acceptance by 15 July 1988, and will be given instructions for final preparation of their papers at that time. Camera-ready papers for the proceedings should be sent to the Program Chairman prior to 15 September 1988.

General information

Date: 28 November to 2 December 1988
Venue: Tokyo Prince Hotel, Tokyo, Japan
Host: Institute for New Generation Computer Technology

Outline of the conference program
General sessions
- Keynote speeches
- Report of research activities on Japan's FGCS Project
- Panel discussions

Technical sessions (parallel sessions)
- Presentation by invited speakers
- Presentation of submitted papers

Special events
- Some of the research results including parallel software and hardware systems, natural-language understanding systems and expert systems will be demonstrated at the conference site.

Further information
Conference information will be available from the Secretariat from December 1987.

Organisation of the Conference

Conference Chairman: Hideo Aiso, Keio University
Conference Vice-Chairman: Kazuhiro Fuchi, ICOT
Programme Chairman: Hidehiko Tanaka, The University of Tokyo
Programme Vice-Chairman: Koichi Furukawa, ICOT
Publicity Chairman: Kinko Yamamoto, JIPDEC
Publicity Vice-Chairman: Fumio Mizoguchi, Science University of Tokyo

Secretariat
FGCS '88 Secretariat, Institute for New Generation Computer Technology (ICOT), Mita Kokusai Building 21F, 1-4-28 Mita, Minato-ku, Tokyo 108, Japan. (Phone: 03-456-3195. Telex: 32964 ICOT.)