Multilevel Data Structures: Models and Performance[†]

A. Moitra[1]
S. Iyengar[2]
F. Bastani[3]
I. Yen[3]

TR 85-679
May 1985

Department of Computer Science
Cornell University
Ithaca, New York 14853

[†] The work of the third author was supported in part by the National Science Foundation under Grant MCS-83-01745.

[1] Professor A. Moitra is with the Department of Computer Science, Cornell University, Ithaca, NY 14853.

[2] Professor S. Iyengar is with the Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803.

[3] Professor F. Bastani and Ms. I. Yen are with the Department of Computer Science, University of Houston, Univeristy Park, Houston, TX 77004.

MULTILEVEL DATA STRUCTURES: MODELS AND PERFORMANCE

A. Moitra, S.S. Iyengar, F.B. Bastani and I.L Yen

ABSTRACT

We advocate a stepwise method of deriving high performance implementation of a set of operations. This method is based on the ability to organize the data into a multilevel data structure so as to provide an efficient implementation of all the operations. Typically, for such data organization the performance may deteriorate over a period of time and that can be corrected by reorganizing the data. This data reorganization is done by the introduction of maintenance processes.

For a particular example we consider the multilevel data organization and the different models of maintenance processes possible. We sketch a correctness proof for the implementation we develop. Performance behaviour for the different models are derived and we also present some simulation studies of the performance.

1. INTRODUCTION

System performance can be improved by scheduling house keeping activities to occur during periods when the processor is waiting for messages or is transmitting messages. In other words performance of a distributed system can be improved by scheduling the computations required to maintain the data structure during the idle periods. This approach is particularly suitable for distributed systems since the data reorganization can be done while a response is being sent to the invoker or while the local processor is waiting for a new request.

For example, it is efficient to search for a key in a perfectly balanced binary search tree. However, rebalancing the tree after every insertion/deletion operation is inefficient. A maintenance process in a distributed system can rebalance the tree while a response is being sent to the invoker or when the processor is waiting for a new request. This method is proposed by Lampson[6] and Manber[7]. Manber gives the algorithms and proofs for maintenance processes for a special form of binary search trees data structure called 'external trees'. Bastani et al. [1,2] have investigated the improvement of the performance of data abstractions in a distributed system through the use of low priority maintenance process. They consider the case where the interface operations are implemented by foreground processes while the maintenance tasks are performed by background processes. They give algorithms for these processes in a stepwise manner, proceeding from coarse grained concurrency to fine grained concurrency.

In this paper we discuss one approach for improving the average response time for a set of operations. This approach is based on the usage of multilevel data structures along with maintenance processes [1,2,3,6,7]. We illustrate this approach by considering one example in detail. For this example we present different models which capture the types of interactions between the processes providing the original operations and the maintenance processes. Even though we take a specific example for illustration purposes, the different models of interactions are general and can be modified to deal with other problems. Finally, we also present some simulation studies.

2. EXAMPLE

The problem we consider is that of implementing a SearchTable which should provide the following operations:

ADD(k,e) : add element e with key k to the SearchTable

DELETE(k) : delete the element associated with key k

LOOKUP(k) : return the element associated with key k

Standard implementations cannot simultaneously optimize all the above three operations. For example,

(1) a linked list is poor for LOOKUP, but good for ADD and DELETE

(2) an unsorted list is poor for LOOKUP and DELETE, but good for ADD

(3) a sorted list is good for LOOKUP, but poor for ADD and DELETE

(4) a binary search tree has a much better performance for ADD and DELETE than a sorted array, though it has a slightly worse LOOKUP performance

(5) a balanced tree improves on the performance time for LOOKUP at the expense of that of ADD and DELETE

(6) hash tables with open addressing are not suitable for DELETE while those with chaining have degraded performance as the chained lists get longer

3. MULTILEVEL DATA STRUCTURES

Typically, the data on which the operations are performed is organized in some particular data structure. However, in the multilevel data structure approach the data is organised in a number of different data structures so as to provide efficient implementation for all the operations. As a sequence of operations are performed, the organization of the data is altered. Now since the different data structures have different performance characteristics, the performance of the operations may deteriorate as a result of the new organization of the data. Consequently, some maintenance processes [7] have to be invoked to reorganize the data so as to improve the performance of the operations.

Since, an operation may be performed under different data organization, the implementation of the operation should be consistent with all the different data organizations. This is achieved by defining two types of invariants for the data organization; a strong and weak invariant, such that the following hold

(1) If the strong invariant is true then any operation can be implemented efficiently. The processing of an operation may falsify the strong invariant.

(2) If the weak invariant holds then execution of an operation will maintain the weak invariant.

The maintenance process when invoked attempts to establish the strong invariant given that the weak invariant holds. Whether the maintenance process actually succeeds in this attempt depends on the model considered; however, for any model if no new requests arrive from the client for a certain period of time then the maintenance process will establish the strong invariant. This point will become clearer when we discuss the different models.

For the data type SearchTable and its associated operations, a multilevel data structure consisting of a binary search tree and a sorted array can be used as given in Fig. 1. Elements to be added are added to the binary search tree and for efficient implementation of DELETE, each element has a flag (with values ('dead' and 'alive') associated with it. The strong invariant is that the binary search tree is empty and that all the elements in the array have flag value 'alive'. One or more maintenance processes (which we will also call as background processes) transfer elements from the binary search tree to the sorted array and also remove dead elements.

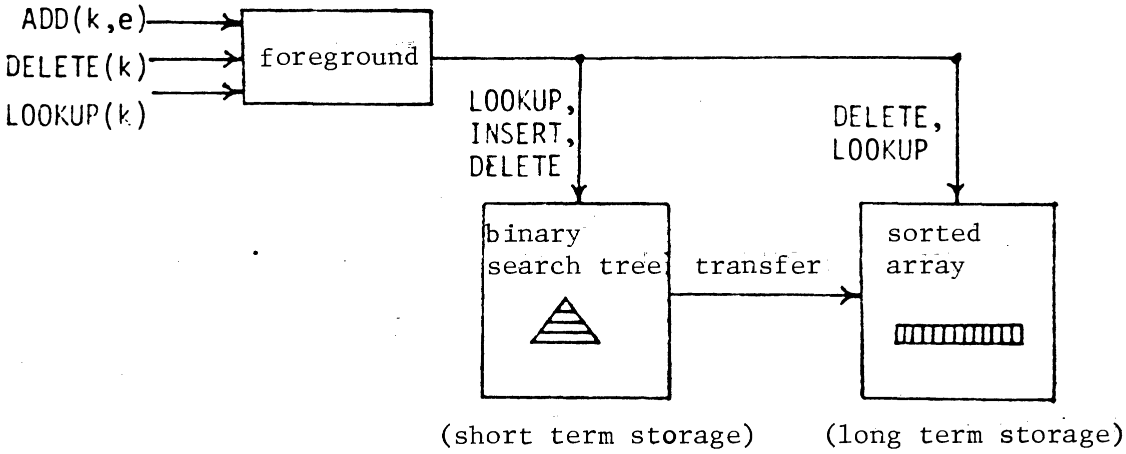

Fig. 1. Multilevel data structure for the SearchTable module.

The reason for calling such a data organization as a multilevel data structure is that it is similar to the memory hierarchy used in computer systems. The binary search tree in the above example can be viewed as a cache storage while the sorted array corresponds to long term storage.

One of the earliest systematic methods of developing interfering programs is the stepwise development method of Jones [5] which uses rely/gurantee conditions for each of the processes. A general method for showing noninterference is to identify an invariant and show that execution of each statement preserves the invariant [8]. A basic approach for implementing such programs is to start from programs with coarse grained concurrency and refine them to ones with fine grained concurrency [3].

In [1] the 'coarse grained concurrency to fine grained concurrency' approach is combined with the stepwise approach of [5]. This allows modular proofs of the different procedures to be undertaken and hence the addition of new operations only requires proofs about the added code.

The stepwise development from coarse grained concurrency to fine grained concurrency proceeds in the following sequence.

1. In Case A, only the foreground process is present, the data structure used is a binary search tree.

2. In Case B, background processes are added but all the processes are nonpreemptible. The multilevel data structure of Fig. 1 is used.

3. In Case C, a background process is preemptible by a foreground process. The multilevel data structure of Fig. 1 is used.

4. MODELS FOR MULTILEVEL DATA STRUCTURES

We now develop formal models for the three cases and derive their performance behaviour (see also [4]). We assume that the client (i.e. the program using the abstract data type) repeatedly executes the following cycle:

> Compute for a period having the exponential distribution with parameter λ; then invoke an operation of the abstract data type.

If the average response time is R, then the productive work, PW, performed by the client is

$$PW = \frac{1/\lambda}{1/\lambda + R}$$

This is the proportion of time that the client is doing some other (presumably more useful) chore than waiting for an operation of the abstract data type to complete. Clearly, if $1/\lambda$ is large then this abstract data type is rarely used by the client, so that optimization is not critical. However, if $1/\lambda$ is small then operations on the abstract data type could be a bottleneck.

4.1. Case A - No Background Processes

The state transition diagram is shown in Fig. 2

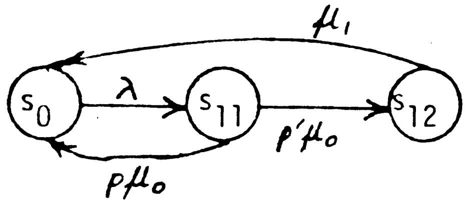

Fig. 2. State transition for Case A.

In state s_0, the client is doing some other work. In state s_{11}, a LOOKUP operation is being performed. In state s_{12}, an update operation (to complete the ADD or DELETE operation) is in progress. The probability that a client request needs to update the data structure is p'. The search operation is completed with rate μ_0, while the update operation has rate μ_1. If P_i denotes the probability of being in state s_i then the following equations are obtained for the states.

for $s_{12} : p' \mu_0 P_{11} = \mu_1 P_{12}$

for $s_{11} : \mu_0 P_{11} = \lambda P_0$

$P_0 + P_{11} + P_{12} = 1$

Solving these we obtain

$$P_0 = \frac{1/\lambda}{1/\lambda + 1/\mu_0 + p'/\mu_1}$$

Now $PW = P_0$ and hence we obtain

$$PW = \frac{1/\lambda}{1/\lambda + 1/\mu_0 + p'/\mu_1}$$

$$R = \frac{1}{\mu_0} + \frac{p'}{\mu_1}$$

Remarks

(1) As λ increases, PW decreases since this module then becomes a bottleneck.

(2) If the search algorithm is improved, then μ_0 increases and hence PW increases.

(3) If there are fewer ADD/DELETE requests then p' decreases and hence PW increases.

(4) If the update algorithm is improved then μ_1 increases and hence PW increases.

4.2. Case B - Nonpreemptible Background Processes

The state transition diagram is shown in Fig. 3.

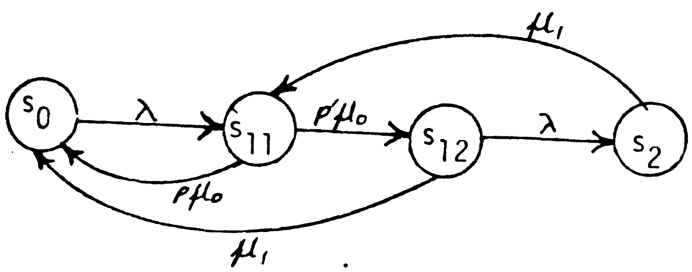

Fig. 3. State transition for Case B.

In this implementation, the foreground process returns control to the client process as soon as the LOOKUP portion of the operation is completed. The background processes do the actual update in state s_{12}. Since there is no preemption, these processes complete the update even if the client issues a new operation. This is indicated by the transition from state s_{12} to s_2. For this case we obtain the following equations,

for $s_2 : \lambda P_{12} = \mu_1 P_2$

for $s_{12} : (\lambda + \mu_1) P_{12} = p' \mu_0 P_{11}$

for $s_{11} : \mu_0 P_{11} = \lambda P_0 + \mu_1 P_2$

$P_0 + P_{11} + P_{12} + P_2 = 1$

Solving these we obtain

$$P_0 = \frac{1}{1 + [\lambda(\lambda + \mu_1)/(p \lambda + \mu_1)][1/\mu_0 + p'/\mu_1]}$$

$$R = \frac{1}{\mu_0} + \frac{\lambda p'}{(\lambda + \mu_1)\mu_1}$$

$$\leq \frac{1}{\mu_0} + \frac{p'}{\mu_1} \qquad \text{since } \frac{\lambda}{\lambda + \mu_1} \leq 1$$

$$= \text{response time for Case A}$$

$$PW = P_0 + P_{12}$$

$$= \frac{1/\lambda}{1/\lambda + 1/\mu_0 + [\lambda/(\lambda + \mu_1)]p'/\mu_1}$$

$\geq PW$ for Case A since $\frac{\lambda}{\lambda + \mu_1} \leq 1$

Thus, implementation for Case B is always better than implementation for Case A.

4.3. Case C - Preemptible Background Processes

The state transition diagram is shown in Fig. 4.

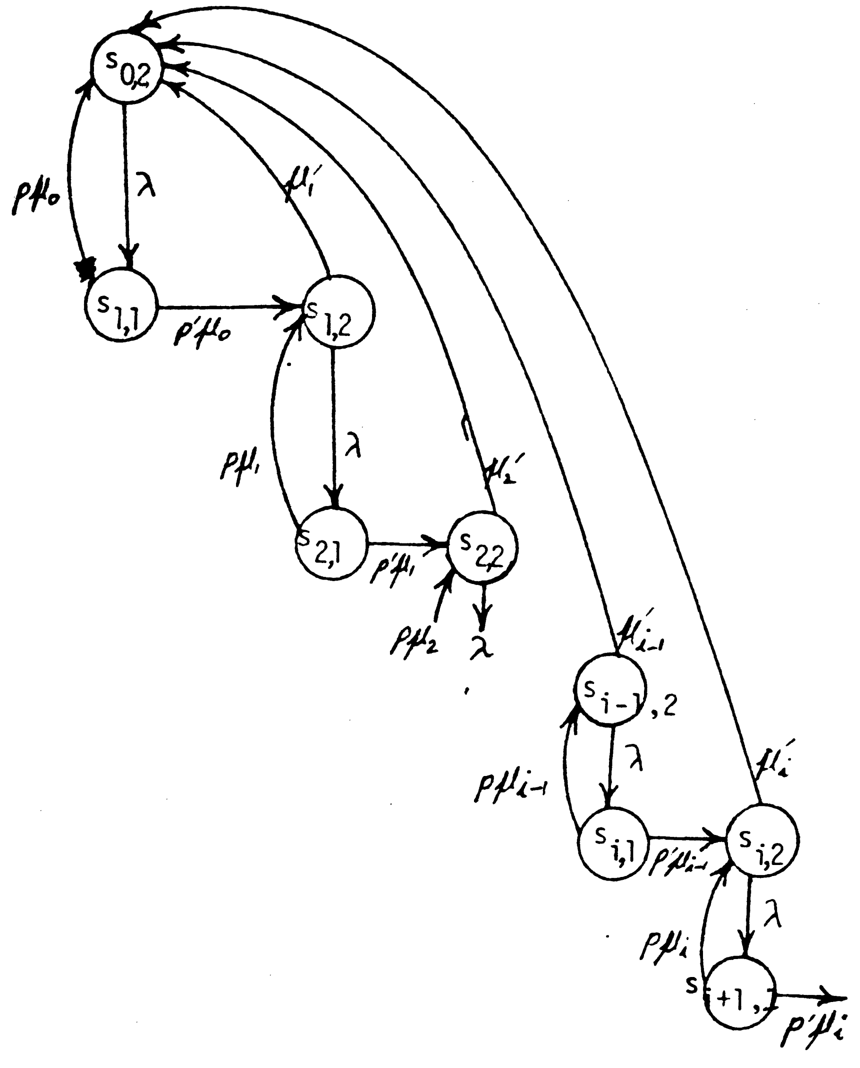

Fig.4. State transition for Case C.

In this implementation, the foreground process can interrupt the background processes. The background processes then continue to do the actual updates in states $s_{1,2}$, $s_{2,2}$,..,$s_{i,2}$,.. The general equations are

for $s_{i+1,1}$, $i \geq 0 : \mu_i \; P_{i+1,1} = \lambda \; P_{i,2}$

for $s_{i,2}$, $i \geq 1 : (\lambda + \mu_i{}') P_{i,2} = p' \mu_{i-1} P_{i,1} + p \mu_i P_{i+1,1}$

$$\sum_{i=0}^{\infty} P_{i,2} + \sum_{i=1}^{\infty} P_{i,1} = 1$$

And these can be solved to give

$$P_{0,2} = \frac{1}{\sum_{i=0}^{\infty} (1 + \lambda/\mu_i) \prod_{j=1}^{i} [p'\lambda/(p'\lambda + \mu_j{}')]}$$

And we have

$$PW = \sum_{i=0}^{\infty} P_{i,2} \qquad = [\sum_{i=0}^{\infty} \prod_{j=1}^{i} [p'\lambda/(p'\lambda + \mu_j{}')]] \, P_{0,2}$$

Now

$$R = \frac{\sum_{i=0}^{\infty} (1/\mu_i) P_{i,2}}{\sum_{i=0}^{\infty} P_{i,2}} = \frac{\sum_{i=0}^{\infty} (1/\mu_i) \prod_{j=1}^{i} [p'\lambda/(p'\lambda + \mu_j{}')]}{\sum_{i=0}^{\infty} \prod_{j=1}^{i} [p'\lambda/(p'\lambda + \mu_j{}')]}$$

A closed form solution is useful to draw some conclusions regarding the performance of this implementation. This is possible under the following assumptions.

(1) $\mu_i{}' = \mu =$ constant

(2) $\mu_i = \dfrac{\mu_0}{i + 1}$

Assumption 1 is optimistic, though it is a reasonable approximation for certain update operations such as removing dead elements. Assumption 2 is pessimistic for a number of processor structures; a more reasonable assumption would be $\mu_i = \dfrac{\mu_{01}}{log(i+1)} + \mu_{02}$ where μ_{02} is the average rate for searching the sorted array while μ_{01} is the average rate for searching the binary tree. We also assume that a steady state is reached so that probabilities for the operations ADD and DELETE are equal; otherwise, the search table will overflow or be empty in the steady state.

With these assumptions we obtain

$$R = \frac{1}{\mu_0}(1 + \frac{p'\lambda}{\mu})$$

$$PW = \frac{1/\lambda}{1/\lambda + (1 + p'\lambda/\mu)/\mu_0}$$

For small λ, these can be simplified to give

$$R \approx \frac{1}{\mu_0}$$

$$PW \approx \frac{1/\lambda}{1/\lambda + 1/\mu_0}$$

Since this is the best possible response time, this implementation is very efficient for reasonable rates of client requests. However, as λ increases, R tends to infinity and PW tends to 0. Thus, at some point the performance of Case C becomes worse than that of Case A.

5. PROGRAM FOR CASE C

The implementation of Case A is straightforward and the implementation of Case B is a special case of that of Case C. So, we now present the details of the implementation for Case C.

5.1. DATA STRUCTURES

The data structure is given below and consists of two binary search trees (with t1, t2 pointing to the roots, and for conciseness we will call them tree t1 and tree t2 respectively) and two arrays, a1 and a22. The reason for using two binary search trees is as follows. The foreground process adds an element to tree t1 when executing an ADD operation. The job of the background process is to merge the contents of a tree with the elements of an array. However, since the background process is preemptible we should allow for elements to be added while this merging is in progress. Hence, the usage of two binary search trees.

Of the two arrays, we use a1 for copying the elements from tree t2 in inorder traversal. This array is not really necessary but its usage simplifies the code. The array a2 contains a sorted list of elements. This array actually has two partitions, a21 and a22, so that the background process alternates between merging a21 and a1 into a22, and merging a22 and a1 into a21. In this process of merging, a21 and a22 grow in opposite directions. We assume that sufficient space is allocated for a2 so that the two partitions do not overlap.

It is possible to write an in-place merge of two array wherein the elements of two sorted arrays are merged without making use of a third array. Such a scheme would avoid extra storage and is particularly efficient for this problem as one of the arrays contains a number of 'dead' elements and hence the data movement can be made minimal. We wrote such an in-place merge, however for simplicity we did not use it in our simulation studies and hence we do not present the code here.

Data Structure

```
type element = record
            I: info;
            S: (dead,alive);
            K: key
          end;
   ptr = ^node;
   node = record
          E: element;
          left,right: ptr:= nil
        end;

var t1,t2: ptr;

var a1: array [1..num1] of element;
   start1,end1: 0..num1;

var a2: array [1..num2] of element;            -- Two partitions:  a21 = a2[start21..end21]
   start21,end21,start22,end22: integer        --                  a22 = a2[start22..end22]
```

5.2. PROGRAM

For simpler and cleaner presentation we take some liberties with PASCAL syntax as summarized below.

1. '{' and '}' are used for 'begin' and 'end' respectively; these will also be used for enclosing assertions. The usage will be clear from the context.

2. Comments are preceded by '--'. Explicit return statement is used.

4. A number of values can be returned as a result of a procedure call using for example the notation ret_value1 X ret_value2 := procedure_name(..).

5. To maintain uniformity between arrays and trees we use 'address of a[m]' to refer to the index 'm' in the array a. For simplicity, if p is an address we use p.I etc. to refer to elements in trees as well as elements in the arrays.

6. '<' and '>' are used for enclosing atomic actions.

The main features of the program presented below are the following. A search for an element proceeds by searching a2, (a1 or t2), t1, in that order until the search is successful. Further, the array a2 is searched by searching only one of its two partitions. The background process ensures that new elements are added only to t1 by switching around the pointers to the roots of the two trees.

```
t1:= nil;                                   -- initialization
t2:= nil;
start1:= 1; end1:= 0;
start21:= 1; end21:= 0;                     -- start21 is always 1
start22:= num2+ 1; end22:= num2             -- end22 is always num2

                -- Foreground Process

LOOKUP(k): found X p:= search_data_structures(k,false);
           if not found or else p.S = dead then return error message
           else return p.I;

ADD(k,i):  found X p:= search_data_structures(k,true);
           if found and p.S = alive then return error message
           else {p.I:= i; p.S:= alive}

DELETE(k): found X p:= search_data_structures(k,false);
           if not found or else p.S = dead then return error message
           else p.S:= dead;

search_data_structures(k:key; add: boolean):
        found X p:= if end21 < start21 or a2[end21].K < k
                    then binary_search(a2,start22,end22)
                    else binary_search(a2,start21,end21);
        if found and p.S = alive then return true X p;
        found X p:= if end1 < start1 or a1[end1].K < k
                    then search_tree(t2,false)
                    else binary_search(a1,start1,end1);
        if found and p.S = alive then return true X p;
        return search_tree(t1,add);

binary_search(var a: array[<>] of element; l,u: integer):
        while l <= u do
          {m:= (l+ u) div 2;
           if a[m].K = k then return true X address of a[m]
```

```
            else if a[m].K < k then l:= m+1
            else u:= m-1};
          return false X nil;

search_tree(var t: ptr; add: boolean):
          if t = nil then
            if not add then return false X nil
            else {new(t); t^.E.K:= k; return false X address of t^.E}
          else if t^.E.K = k then return true X address of t^.E
          else if t^.E.K < k then search_tree(t^.right,add)
          else search_tree(t^.left,add);
```

– **Background Process**

```
while true do
  {<t2:= t1; t1:= nil;>
   transfer_t2_to_a1;
   expand_a22;
   <t2:= t1; t1:= nil;>
   transfer_t2_to_a1;
   expand_a21;
   end1:= 0; start1:= 1};

transfer_t2_to_a1: inorder(t2)

inorder(var t: ptr) = if t <> nil then
                  {inorder(t^.left);
                   <if t^.E.S = alive then {end1:= end1+ 1; a1[end1]:= t^.E};
                   x:= t;
                   t:= t^.right>;
                   dispose(x);
                   inorder(t)}

expand_a22: while end21 >= start21 and end1 >= start1 do
                if a2[end21].S = dead then end21:= end21-1
                else if a1[end1].S = dead then end1:= end1-1
                else if a2[end21].K > a1[end1].K
                    then {<start22:= start22-1; a2[start22]:= a2[end21]; end21:= end21-1>}
                else                  -- a2[end21].K < a1[end1].K
                  {<start22:= start22-1; a2[start22]:= a1[end1];end1:= end1-1>};

              while end21 >= start21 do
                if a2[end21].S = alive
                then {<start22:= start22-1; a2[start22]:= a2[end21]; end21:= end21-1>}
                else end21:= end21-1;

              while end1 >= start1 do
                if a1[end1].S = alive
                then {<start22:= start22-1; a2[start22]:= a1[end1]; end1:= end1-1>}
                else end1:= end1-1;
```

```
expand_a21: while end22 >= start22 and end1 >= start1 do
              if a2[start22].S = dead then start22 := start22+ 1
              else if a1[start1].S = dead then start1:= start1+ 1
              else if a2[start22].K < a1[start1].K
                  then {<end21:= end21+ 1; a2[end21]:= a2[start22]; start22:= start22+ 1>}
              else          -- a2[start22].K < a1[start1].K
                {<end21:= end21+ 1; a2[end21]:= a1[start1]; start1:= start1+ 1>};

            while end22 >= start22 do
              if a2[start22].S = alive
              then {<end21:= end21+ 1; a2[end21]:= a2[start22]; start22:=start22+ 1>}
              else start22:=start22+ 1;

            while end1 >= start1 do
              if a1[start1].S = alive
              then {<end21:= end21+ 1; a2[end21]:= a1[start1]; start1:= start1+ 1>};
              else start1:= start1+ 1;
```

6. PROOF OF CORRECTNESS

We discuss the correctness of the program presented in the previous section. To avoid excessive details we will only sketch the correctness proof leaving out some of the details. Let

```
D = the multiset of all the alive elements in a1[start1:end1], a2[start21:end21],
    a2[start22:end22], tree t1 and tree t2

INV =  no duplicates in D and D is the set of all the elements in the system
       and a1[start1:end1] o inorder(t2), a2[start21:end21] o a2[start22:end22],
           inorder(t1) are all sorted on key
       and start21=1 and end22=num2
```

where '*o*' denotes concatenation.

The invariant INV is actually the weak invariant that we referred to earlier. The strong invariant for this program, SINV, is given below

```
SINV =  no duplicates in D and D is the set of all the elements in the system
        and there are no dead elements in the system
        and t1=nil and t2=nil and end1 < start1 and [end21 < start21  ∨ end22 < start22]
        and  a2[start21:end21] o a2[start22:end22] is sorted on key
```

The background process in Case B establishes the strong invariant, however, in Case C this is established only if the background process is not preempted by the foreground process for a period of time. We will come back to this point later.

Consider the foreground process, we first prove that INV is indeed an invariant. For this we first show that

```
A1 : {INV}
    found X p := search_data_structures(k, false)
    {INV and (found  ⇒  p.K = k)
         and (not found  ⇒  (∀ j member D : j.K ≠k))}
A2 : {INV}
    found X p := search_data_structures(k, true)
    {INV and (found  ⇒  p.K = k)
         and (not found  ⇒  (∀ j member D - {p} : j.K ≠ k
                                  and p points to a new location added to t1))}
```

From INV it follows that to search for an element in a2[start21:end21] and a2[start22:end22] it is enough to search only one partition. Similarly, it is enough to search for an element in a1[start1:end1] and t2 by just searching one of them. Proving A1 is then straightforward. In proving A2 the only complication is that if a new location is allocated we require that it be in tree t1 and this is achieved because search of t2 is done always by search_tree(t2, false).

Now, LOOKUP and DELETE both maintain the invariant INV since search_data_structures(k,false) maintains it. For ADD, executing 'p.I:= i; p.S:= alive' maintains the invariant INV because of A2.

We now consider the background process. Program annotation for the background process will be more complicated as this process can be preempted by the foreground process. We show the following annotation to be valid.

```
l1  : {INV and t2 = nil and start1=1 and end1=0 and start22=num2+ 1}
l2  : while true do
l3  :    {INV and t2 = nil and start1=1 and end1=0 and start22=num2+ 1}
l4  :   {<t2:= t1; t1:= nil;>
l5  :    {INV and start1=1 and end1=0}
l6  :    transfer_t2_to_a1;
l7  :    {INV and t2 = nil and start1=1}
l8  :    expand_a22;
l9  :    {INV and t2 = nil and start1=1 and end1=0 and end21=0}
l10 :    <t2:= t1; t1:= nil;>
l11 :    {INV and start1=1 and end1=0}
l12 :    transfer_t2_to_a1;
l13 :    {INV and t2 = nil and start1=1}
l14 :    expand_a21;
l15 :    {INV and t2 = nil and end1 < start1 and start22=num2+ 1}
l16 :    end1:= 0; start1:= 1
l17 :    {INV and t2 = nil and start1=1 and end1=0 and start22=num2+ 1}
        }
```

We now proceed sequentially to show that the above annotation is valid. Assertion at l1 holds because of the initialization. We now show that

INV and t2 = nil and start1=1 and end1=0 and start22=num2+ 1

is the loop invariant for the background process. We do that by assuming it holds at the start of the execution of this loop and show that it will then hold after the loop has been executed once more. The validity of the triples l3,l4,l5; l9,l10,l11 and l15,l16,l17 is straightforward. We therefore consider the assertions for transfer_t2_to_a1, expand_a22 and expand_a21.

The effect of the code for transfer_t2_to_a1 is that values from t2 are copied in inorder traversal into a1. Now since the precondition for executing this is that start1=1 and end1=0 it follows that a1[start1:end1] *o* t2 is sorted; everything else in INV remains as before.

For the code expand_a22, the following loop invariants can be used for the three while loops respectively.

I1: a2[start21:end21] *o* a2[start22:end22], a1[start1:end1] *o* a2[start22:end22] is sorted on key and t2=nil
I2: I1 and ((start1=1 and end1=0) $\vee$ end21=0)
I3: I1 and end21=0

and consequently the following assertion will hold at the end of expand_a22

INV and t2=nil and start1=1 and end1=0 and end21=0

For the code expand_a21, the following loop invariants can be used for the three while loops respectively.

I4: a2[start21:end21] *o* a2[start22:end22], a2[start21:end21] *o* a1[start1:end1] is sorted on key and t2=nil
I5: I4 and (end1 < start1 $\vee$ start22=num2+ 1)
I6: I4 and start22=num2+ 1

and consequently the following assertion will hold at the end of expand_a21

INV and t2=nil and end1 < start1 and start22=num2+ 1

Consequently the background process maintains the invariant INV.

We now prove that the foreground and background processes correctly implement their respective tasks.

Foreground Process : Its 'rely condition' is that the multiset D is not changed by the background process. The background process 'guarantees' this since (a) whenever it executes <t2:= t1; t1:= nil>, t2 is nil, and (b) all data structure modifications required in order to transfer items with flag=alive from one data structure to another (e.g. in procedures transfer_t2_to_a1, expand_a22, expand_a11) are atomic and correct.

Background Process : Its 'rely condition' is that the only modifications by the foreground process to t2, a1, a21, a22 is to change the flag of an item from alive to dead. The foreground process 'guarantees' this since the only modifications to t2, a1, a21, a22 is to change the flag of an item from alive to dead in operation DELETE(k). All the other modifications by the foreground process occur in t1, and these are of no consequence to the background process.

The task of the background process is to achieve the strong invariant, SINV, if it is not interrupted by the foreground process. Assume that it is not interrupted between l3 and l9. Then, (a) t1 is nil since the foreground process has not added any items to t1 (by assumption), and (b) there are no items with flag=dead since (i) all such items which exited at l3 have been removed by the background process by the time execution reaches l9, and (ii) the foreground process has not changed the flag of any item (by assumption). Hence, at l9 we can assert:

INV and t2=nil and start1=1 and end1=0 and end21=0 and t1=nil

and there are no items with flag=dead in the system

This implies SINV. A similar argument can be given for the phase starting at l9 and terminating at l17.

7. PERFORMANCE STUDIES

Fig. 5 shows the results of of an experiment conducted to compare the performance of cases A and C using VAX/VMS interrupt and timing system services. The details of the experiment are discussed in [9]. The parameters of the experiment are:

1. λ, the rate at which the client issues a request;

2. p, the probability that the request is for LOOKUP.

Also, for each request there is a parameter which specifies the probability that the request is valid. For the data shown here, these probabilities are all 0.9.

From Fig. 5, we observe that

1. For small to medium λ, Case C is much better than Case A.

2. Case C is better than Case A as the LOOKUP probability p increases.

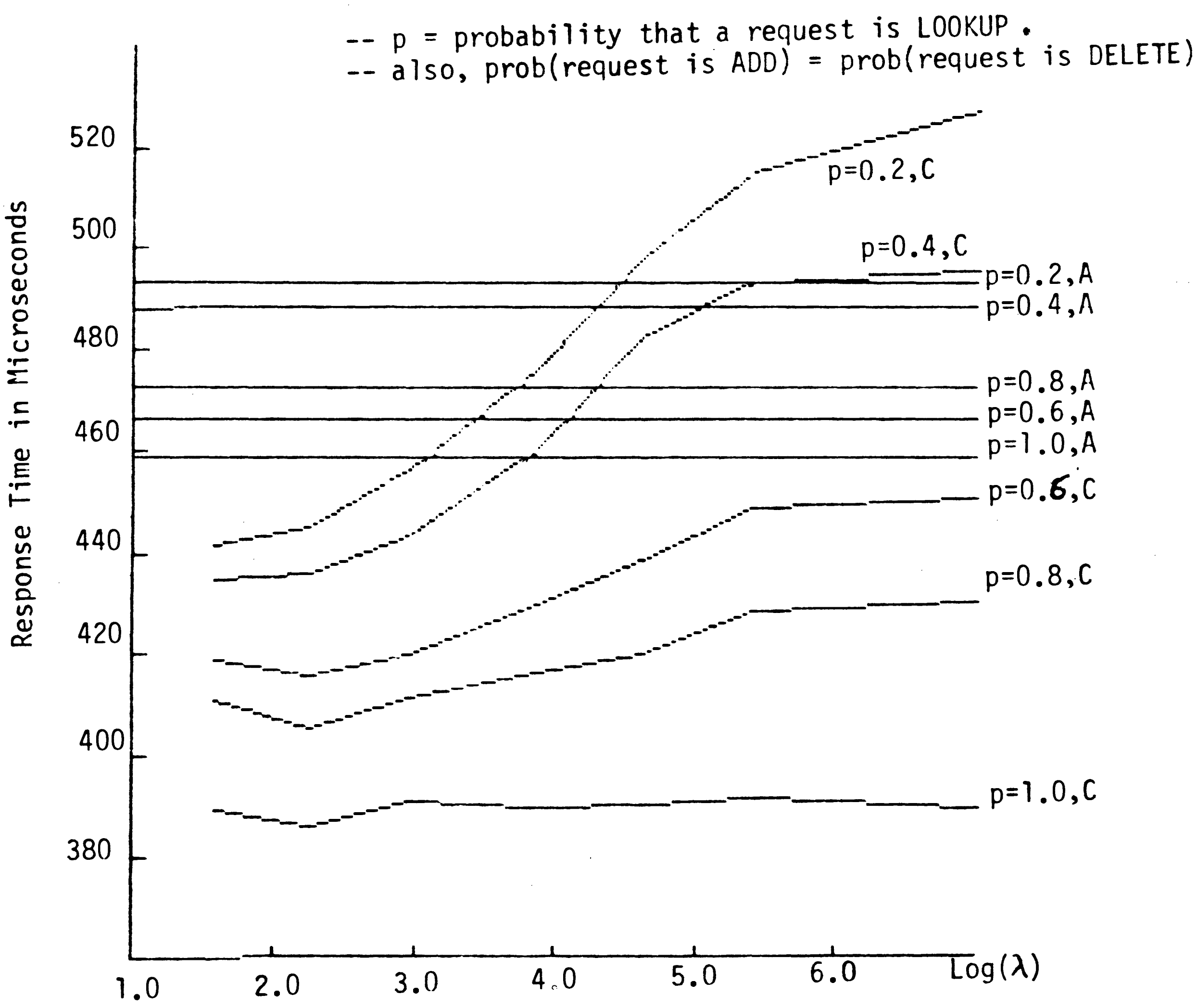

Fig. 5. Experimental comparison of Cases A and C.

SUMMARY

In this paper we have discussed one method for developing high performance implementation for abstract data types. This method relies on the usage of multilevel data structures and maintenance processes and achieves high performance by scheduling the computations required to maintain the data structure during the idle periods. This approach is particularly suitable for distributed systems as the data reorganization can be done while response is being sent to the client. We have presented various models for the maintenance processes. Both the derived performance and the experimental results favor the usage of maintenance processes for a wide range of parameters.

Possible research directions in this area include developing performance models for multilevel data structures when the foreground process is also preemptible and when there are multiple clients concurrently accessing the data type.

REFERENCES

1. F.B. Bastani, S.S. Iyengar, and I.L. Yen, "Concurrent maintenance of multilevel data structures," Tech. Rep. UH-CS-85-3, Dept. Comp. Sc., Univ. Houston - Univ. Park, Jan. 1985.
2. F.B. Bastani, I.L. Yen, A. Moitra, and S.S. Iyengar, "Impact of parallel processing on software quality," Tech. Rep. UH-CS-85-5, Dept. Comp. Sc., Univ. Houston - Univ. Park, Apr. 1985.
3. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens, "On-the-fly garbage collection: an exercise in cooperation," Comm. ACM, Vol. 21, No. 11, Nov. 1978, pp. 966-975.
4. T. Hickey, and J. Cohen, "Performance analysis of on-the-fly garbage collection." Comm. ACM, Vol. 27, No. 11, Nov. 1984, pp. 1143-1154.
5. C.B. Jones, "Tentative steps towards a development method for interfering programs," ACM Trans. Prog. Langs. and Sys., Vol. 5, No. 4, Oct. 1983, pp. 596-619.
6. B.W. Lampson, "Hints for computer system design," IEEE Softw., Vol. 1, No. 1, Jan. 1984, pp. 11-28.

7. U. Manber, "Concurrent maintenance of binary search trees," IEEE Trans. Softw. Eng., Vol. SE-10, No. 6, Nov. 1984, pp. 777-784.

8. S. Owicki, and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," Comm. ACM, Vol. 19, No. 5, May 1976, pp. 279-285.

9. I-L. Yen, The role of Parallel Processing in Application Programs, M.S. Thesis, Dept. Comp. Sc., Univ. Houston - Univ. Park, May 1985.