Efficient Abstract Data Type Components for Distributed and Parallel Systems

Farokh Bastani, Wael Hilal, and S. Sitharama Iyengar

One way of improving a software system's comprehensibility and maintainability is to decompose it into several components,[1] each of which encapsulates some information concerning the system.[2] These components can be classified into four categories, namely, abstract data type, functional, interface, and control components. Such a classification underscores the need for different specification, implementation, and performance-improvement methods for different types of components. (See sidebar.) This article focuses on the development of high-performance abstract data type components for distributed and parallel environments.

An abstract data type component provides a collection of operations that can be invoked by other components. In a distributed system, an abstract data type can be modeled as a *server* receiving requests for its operations from *clients*.[3] The server and its clients interact using the interprocess communication (IPC) primitives provided by the operating system and can run on either the same or different machines. (Here we extend the terms "server" and "client" to refer also to the corresponding entities in sequential and parallel environments.)

The use of abstract data types reduces the complexity of a large program. The program becomes easier to develop and understand since the code for the client and the abstract data type can be dealt with separately. Further, the program is easier to modify since changes in the client code do not require changes in the abstract data type component and vice versa, so long as the specification of their interface is not changed.

Abstract data type components reduce the complexity of a large program by providing a collection of operations that can be invoked by other components.

However, a side effect of using abstractions is that system performance deteriorates due to both the cost of procedure calls and IPC primitives, and the encapsulation of the abstraction's data structures.[4] On the one hand, the client knows the sequence of operations it should perform, but it cannot control their cost (that is, the time required to perform an operation). On the other hand, the server knows the cost of each operation, but it cannot control the sequence in which the operations will be invoked. Hence, neither the server nor the client has the knowledge to minimize the time required to perform the operations.

This article discusses two methods of improving abstract data type performance—one for distributed systems and another for parallel systems. Both methods have the desirable feature that they do not affect the time spent by the client between successive invocations of the server. Our performance measure in both cases is the *response time*—the time a client spends waiting for the reply to a server operation it has invoked. The response time is equal to the sum of the waiting and service times of requests at the server plus the request and response transmission times. The maximum response time is important for real-time applications, while the average response time is important for other applications.

For distributed programs, we consider the use of multilevel data structures, which efficiently implement all the abstraction's operations if the state of the data structure satisfies certain conditions. The process-

0018-9162/87/1000-0033$01.00©1987 IEEE

ing of an operation may move the data structure to a state that does not satisfy these conditions. If the data structure is not restructured to satisfy these conditions, the performance can deteriorate. We discuss and compare two policies for restructuring multilevel data structures.

For parallel systems, we consider an implementation of abstract data type components that provides extremely high performance—typically, constant time implementation of all the operations of the data type. Our approach is based on the notion of broadcasting sequential processes.[5]

Distributed programs

In distributed programs, the server runs on a dedicated processor, and clients invoke its operations via remote procedure calls. In this environment, each component comprising the software system can be implemented as a group of processes with a dedicated processor allocated to it. If an abstract data type component has several clients, the sequence of operations invoked by one client can be interleaved in arbitrary ways with those of other clients. Hence, performance improvement methods available for sequential programs—such as maintaining additional variables to dynamically detect invocation sequences or transforming the source code of the client—either will not be useful for or cannot be applied to distributed (or even concurrent) programs. Instead, the abstract data type component must provide efficient implementation of all its operations. However, this objective is difficult to achieve with conventional data structures since a data structure typically permits efficient implementation of some operations at the expense of others. For example, using an array for a linear list

Software components

Our classification of software components is based on the relationship between their input (some sequence $I \equiv < i_1, i_2, \ldots, i_n >$), output (some sequence $O \equiv < o_1, o_2, \ldots, o_m >$), and state, S.[12] A component can be implemented as either a module in a sequential program or a group of processes in a distributed computation.[3]

Abstract data type components. These components provide a collection of functions that implement some mathematical objects, such as lists, queues, and sets. These objects have states that can be inspected by using a set of value-returning functions (V-functions).[4] The state of an object can be changed by invoking operation-performing functions (O-functions). For abstract data types, O and the next state, S', are functions of I and S. They can be specified using the axiomatic specification method,[11] which essentially describes the effect of each O-function on each V-function. An abstract data type component can be implemented by following these steps:

(1) Choose a data structure. The choice will depend upon the desired performance of the implementation.

(2) Specify the representation invariant. The representation invariant is a predicate that characterizes the set of states that the implementation may cause the selected data structure to assume.

(3) Specify the mapping function for each V-function. That is, show how each V-function can be implemented using the selected data structure.

(4) Implement the initialization O-functions to establish the representation invariant.

(5) Write the code for the remaining O-functions to change the V-functions as required by the specification, assuming that the representation invariant is true. Ensure that the representation invariant is true after the execution of each O-function.

Methods of implementing efficient abstract data type components for a sequential program include

(1) the addition of special purpose operations to the set of operations provided by the component,
(2) the use of a sophisticated implementation that keeps additional state information to dynamically detect and optimize invocation of sequences of operations,
(3) the use of context-dependent transformations, and
(4) the use of self-reorganizing data structures.

Functional components. These components correspond to mathematical functions that operate on abstract data types (for example, procedures for inverting matrices, sorting lists, etc.). Functional components do not retain any state information, so O depends only on I. Further, o_i, $1 \le i \le m$, cannot be generated without knowing all i_j, $1 \le j \le n$. They can be specified using input/output assertions. They can be systematically developed using Dijkstra's weaker-relation-to-stronger-relation program construction method. The performance of a functional component can be improved by source code transformation or recursion elimination methods. Systolic arrays are used to obtain efficient functional components in parallel environments.

Interface components. These components operate on unbounded input and output data streams or procedure call streams. For example, an interface module for a single character at a time I/O device may provide functions for 80-character card I/O. As in the case of functional components, for interface components O is a function of only I. However, $\exists j$, $1 \le j \le m$, such that $\forall p$, $1 \le p \le j$, o_p can be generated on the basis of $< i_1, i_2, \ldots, i_{k_p} >$, $1 \le k_1 \le k_2 \le \ldots \le k_p < n$. They can be specified using translation grammars.[13] They can be systematically constructed using the recursive descent technique if the translation grammar is a single-symbol look-ahead grammar, LL(1). One way of achieving efficient implementation of interface components is to decompose the translation into a network of processes and assign each process to a different processor.

Control components. These components usually occur at the upper levels of a program where they implement the program's requirements specification by making use of its lower level components. They can be specified using requirement specification languages such as operational specifications[14] or goals. Efficient implementation of control components is possible by decomposing the overall goal into subgoals that can be assigned to different processes using AND/OR-parallelism. The resulting control structure ranges from a fully decentralized one (wherein the processes are autonomous) to a hierarchical control structure.

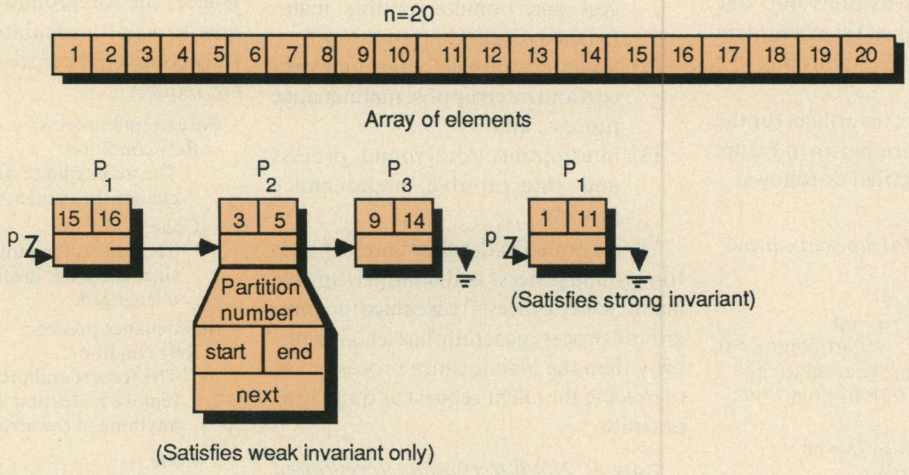

Figure 1. A multilevel data structure for the linear list module.

component optimizes retrieval or update of the ith item in the list while penalizing its addition or removal.

Multilevel data structures facilitate efficient implementation of all the operations of the data type, provided the data structure is in a state that satisfies certain conditions.[6] The data structure may need to be restructured to satisfy these conditions so as to avoid deterioration in the service time (that is, the time required to process an operation). One approach is to assign this task to a separate maintenance process running concurrently with the foreground process that accepts and processes client requests.[3,7,8]

Multilevel data structures. Multilevel data structures can be characterized by weak and strong invariants. When the data structure satisfies the strong invariant, the average service time is small. Processing an operation-performing function (O-function) may leave the data structure in a state that does not satisfy the strong invariant but does satisfy the weak invariant. As more and more O-functions are processed, the average service time increases. Hence, the data structure has to be restructured to reestablish the strong invariant.

Definition. A multilevel data structure D has associated with it a sequence of invariants $I_1, I_2, \ldots, I_n$, such that

(1) D always satisfies I_1;
(2) $I_n \rightarrow I_{n-1} \rightarrow \ldots \rightarrow I_1$, where $\rightarrow$ denotes logical implication; and
(3) if D does not satisfy I_j, then 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 referred to as the *weak invariant* of D while I_n is the *strong invariant* of D. An activity that causes D to satisfy I_j is called a *maintenance* action. Two important maintenance strategies for distributed systems are *periodic* maintenance and *concurrent* maintenance. A maintenance strategy is said to be periodic if the strong invariant is established periodically, for example, after every T seconds or after the completion of M requests. A maintenance strategy is said to be concurrent if a separate process, called the *maintenance process*, performs the maintenance work. The maintenance process establishes the strong invariant after a finite period of time in which there are no further client requests. For each client request, the foreground process executes the desired operation and sends back the response to the client.

Multilevel data structures can be used in implementing many abstract data types. For example, a linear list component provides the following operations:

(1) ith(l:list; i:integer) → return the ith element in list l.
(2) Append(l:list; i:integer; e:element) → add element e after the ith element in list l.
(3) Delete(l:list; i:integer) → remove the ith element from list l.

The ith element can be located efficiently using an array, while insertion of a new element or deletion of an existing element can be done efficiently using a linked list. We can combine these two data structures to take advantage of their best features, as suggested by Lampson.[7] Figure 1 illustrates this multilevel data structure.

Each node in the linked list pointed to by p contains the starting and ending positions of a partition (that is, a sequence of elements in the array). The strong invariant is that the linked list contains at most one node. The weak invariant is that each partition is nonempty and partitions corresponding to different nodes do not overlap. When the strong invariant is satisfied, the foreground process can efficiently locate an element. It can update the list by splitting a partition into two and creating a new partition, if necessary. This procedure increases the number of nodes in the linked list. In this case, only the weak invariant is satisfied. If no maintenance action is taken, the data structure can

gradually degenerate into an ordinary linked list resulting in a loss of performance. The maintenance process must consolidate the various partitions into one partition so that the linked list will contain only one node, thus reestablishing the strong invariant.

The strong and weak invariants for the multilevel data structure shown in Figure 1 can be formally specified as follows:

Strong invariant:
$p = \textbf{nil} \vee (p.next = \textbf{nil} \wedge 1 \le p.start \le p.end \le n)$;
Weak invariant:
$\forall q$: Reachable(q,p)$\rightarrow q = \textbf{nil} \vee |Partition(q)| > 0$;
$\forall q,r$: Reachable(q,p)$\wedge$Reachable(r,p)
$\wedge q \neq r \rightarrow$ Partition(q)$\cap$Partition(r)$= \varnothing$;
where
Reachable(q,p)$\equiv p = q \vee (p \neq \textbf{nil}$
$\wedge$Reachable(q,p.next));
Partition(p)$\equiv \{i | p \neq \textbf{nil}$
$\wedge 1 \le p.start \le i \le p.end \le n\}$;
$|S| \equiv$ cardinality of set S.

The strong invariant must be selected so that all operations of the abstract data type can be executed quickly when the data structure is in a state satisfying it. The weak invariant characterizes the set of states after each indivisible action of any process accessing the data structure. It depends on the granularity of the concurrency (that is, the coarseness of the indivisible actions).

Stepwise development method for concurrent maintenance. The code for the foreground and maintenance processes can be developed by systematically proceeding from coarse-grained concurrency to fine-grained concurrency.[8] In addition, rely/guarantee conditions[9] can be used to simplify the proof of noninterference at each stage. The *rely condition* of a process is the assumption it makes about the behavior of other processes, while the *guarantee condition* of the process is the condition other processes expect it to satisfy after each of its indivisible actions. To ensure that the semantics of the abstraction is satisfied, the maintenance process must ensure that its guarantee conditions (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. In the following example, we illustrate this stepwise development methodology by applying it to the linear-list abstract data type. We assume that during Append operations, the foreground process adds elements to a new partition it creates in a separate area called the *work area*. There are three steps:

(1) noninterruptible foreground process and noninterruptible maintenance process,
(2) noninterruptible foreground process and interruptible maintenance process, and
(3) interruptible foreground process and interruptible maintenance process.

We do not consider the "interruptible foreground process and noninterruptible maintenance process" case since the foreground process generally has a higher priority than the maintenance process so as to execute the client request as quickly as possible.

Case 1: Noninterruptible foreground process and noninterruptible maintenance process. In this case, the foreground process completes a client request before yielding the processor, and the maintenance process completes a maintenance cycle before yielding the processor.

Foreground process:
Rely condition:
The maintenance process does not change the contents of the string. This condition is required to correctly implement the operations of the abstract data type.
Code:
<< search the linked list until the appropriate partition is found based upon the position specified by the client;
if the operation is Append or Delete **then**
split the node into two nodes, if necessary;
if the operation is Append **then**
begin
create a new node for a new partition in the work area; add element to this partition
end; >>

(Note: The symbol "<<" denotes the start of an indivisible action; the symbol ">>" denotes the end of an indivisible action.)

Maintenance process:
Rely condition: TRUE.
Code:
loop forever
<< compact the partitions in the array towards the high end of the array—no new nodes are created in this phase; merge the partitions in the array and those in the work area towards the low end of the array—after this phase the strong invariant is true; >>

Case 2: Noninterruptible foreground process and interruptible maintenance process. This case is an extension of the previous case, wherein the granularity of the indivisible actions of the maintenance process is gradually made finer and finer. Hence, the foreground process does not have to wait for a maintenance cycle to be fully completed before serving a new client request.

Foreground process:
Rely condition:
The maintenance process does not change the contents of the list.
Code:
The code is the same as in Case 1 since the weak invariant is unchanged.
Maintenance process:
Rely condition:
The foreground process does not remove nodes and does not change anything in the array.
Code:
Compact and merge as above using indivisible actions for moving elements one at a time; other indivisible actions are used for creating and deleting nodes.

Case 3: Interruptible foreground process and interruptible maintenance process. In this case, the granularity of the maintenance process is kept unchanged, while that of the foreground process is steadily reduced. This procedure allows the maintenance process to operate concurrently with the foreground process. Also, the foreground process uses explicit locks to block access to portions of the data structure that it needs (or may need) to access to complete processing a client's request. Since we do not wish to slow down the foreground process unnecessarily, the overheads for guaranteeing the semantics of the lock mechanism are mainly taken care of by the maintenance process.

Foreground process:
Rely condition:
The maintenance process does not change the contents of the list.
Code:
Use indivisible actions to split a node (for Append and Delete operations) and to create and lock a node for a new partition in the work area for the Append operation; indivisible actions are not required for filling the partition and then unlocking the corresponding node.
Maintenance process:
Rely condition:
The foreground process does not remove nodes and does not change anything in the array.
Code:
Before moving an element out of a partition, ensure that the foreground process has not locked it; if the partition is locked, then either wait until it is unlocked or else skip over this par-

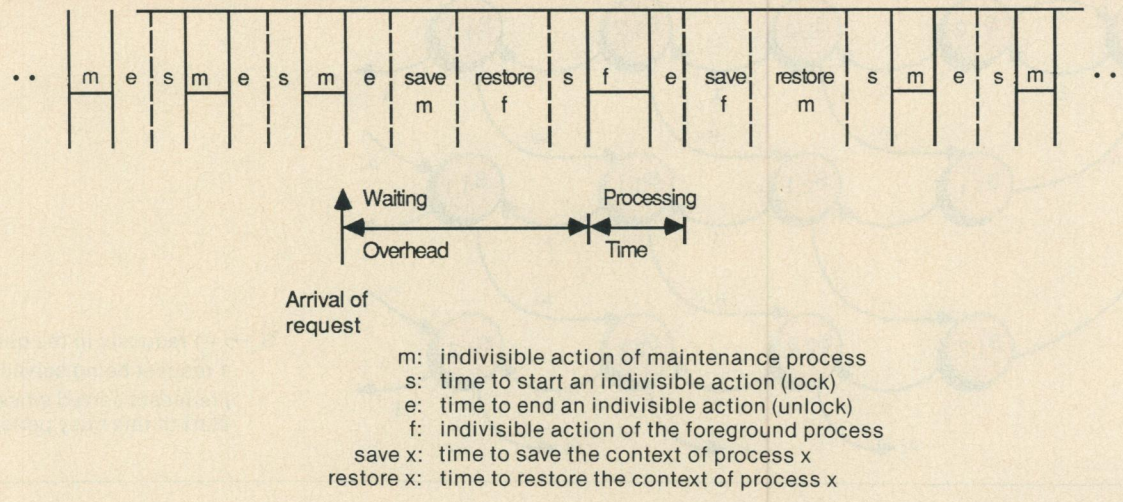

Figure 2. The overhead for Case 2 (noninterruptible foreground process and interruptible maintenance process).

tition during this iteration; the rest is similar to the code in Case 2.

We have also used this approach for developing algorithms for maintaining a queue/sorted-array combination, a binary tree/sorted-array combination, equivalence relations, hash tables, and a multiway tree structure for a directory server.

Performance. The main advantage of concurrent maintenance is that the maintenance tasks can be performed whenever the foreground process is idle. Ideally, the foreground process will always see the data structure in a state in which it satisfies the strong invariant. However, several factors affect the performance, and trade-offs are required for achieving optimal performance.

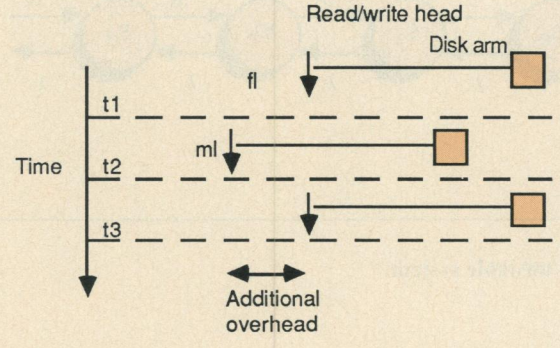

Figure 3. The overhead due to disk seek-time.

As shown in Figure 2, the performance for Case 2 (the noninterruptible foreground process and interruptible maintenance process) is affected by the cost of implementing indivisible actions and context switches, and by the average length of the indivisible actions of the maintenance process. The time taken to switch from the maintenance process to the foreground process and the time required to start and end indivisible actions cannot be easily changed by the application programmer. One parameter that can be controlled is the average length of the indivisible actions of the maintenance process. From Figure 2 we see that, for the first request processed by the foreground process after an idle period, the average waiting time decreases as the average length of the indivisible actions of the maintenance process decreases (in other words, as concurrency becomes finer grained). However, the efficiency of the maintenance process (defined as the proportion of time it does useful work in an idle period of the foreground process) decreases as concurrency becomes finer grained, since the overhead of implementing indivisible actions increases relative to their average length. As a result, the foreground process is less likely to see an optimal data structure and the processing time for the request increases.

The average time that the foreground

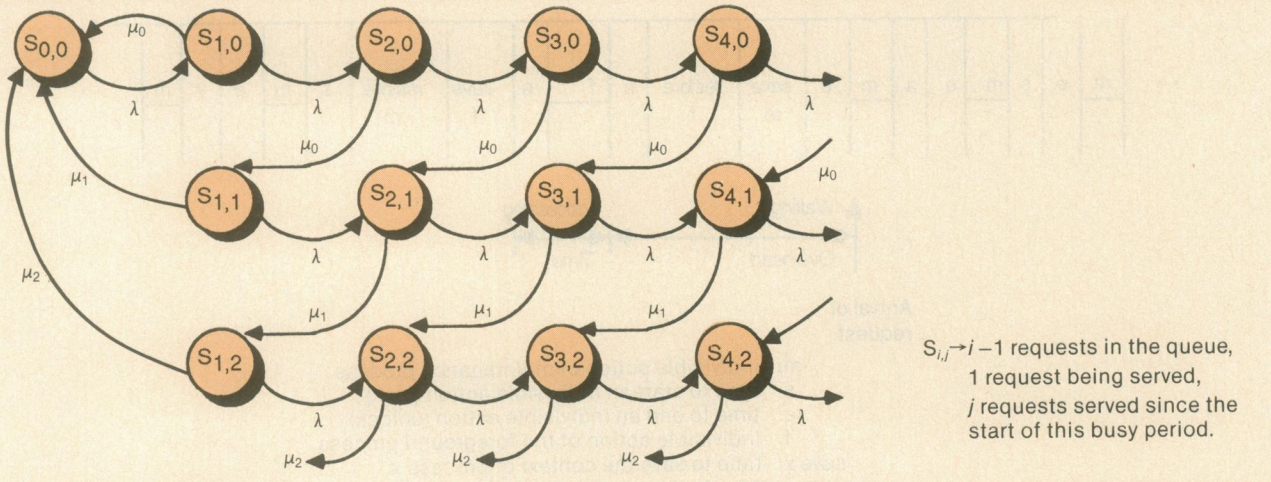

Figure 4. Instability of pure concurrent maintenance.

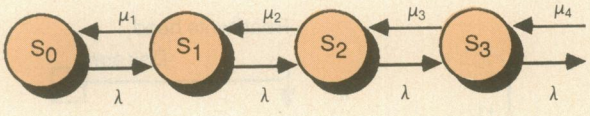

Figure 5. A simple unstable system.

process must wait when it receives a request can also be increased by unexpected interactions between the foreground and maintenance processes when disk-resident data structures are involved. Figure 3 illustrates this situation.

Assume that the foreground process services the last request in a busy period and leaves the disk arm positioned at location $f1$. The maintenance process wakes up and leaves the disk arm at position $m1$ just before the start of the next busy period of the foreground process. If the foreground process needs to position the disk arm near position $f1$ and far from $m1$, this adjustment causes a delay that would not have occurred without the presence the maintenance process. (In this example, we assume that the disk is part of the hardware system dedicated to the component.)

So far we have assumed that the foreground process needs to wait at most until the maintenance process completes its current indivisible action (that is, that the maintenance process always has a lower priority than the foreground process). For Poisson arrivals, this condition can make the system unstable no matter how efficient the maintenance process is, since there is a nonzero probability that the maintenance process will never be scheduled.[10] Figure 4 shows the state transition diagram for a simplified model of foreground/background processes. In this model, the arrival rate of requests is Poisson with parameter λ, the service time for the ith request in a busy period is exponential with parameter μ_i, and the background process can reestablish the strong invariant in zero time. Due to degradation in the data structure, μ_i is a decreasing function of i. Figure 5 shows another system that has a smaller average service time. Simple analysis shows that if $\exists\, i_0 \geq 0$ such that $\mu_{i_0} < \lambda$, then the probability that the system is in state s_i is $0 \forall i \in [0, \infty)$. The system will have an infinite number of customers in the queue in the steady state. This instability can be qualitatively attributed to the fact that a busy period may never end if the average service time exceeds the average interarrival time.

One solution to this problem is to vary dynamically the relative priorities of the foreground and maintenance processes. A strategy that our simulation study shows to be quite good uses the following procedure[10]: After the foreground process has completed a client request, the maintenance process is invoked to do a complete cleanup with probability $1 - p^d$, where d is a measure of the increase in the average service time of abstract data type operations. For example, for the multilevel data structure shown in Figure 1, d can be the number of nodes in the linked list. As p increases, the performance of the system improves until a critical point is reached. If p is increased beyond that point, the performance deteriorates as shown in Figure 6.

Another method of reestablishing the

strong invariant is to perform the maintenance tasks periodically rather than concurrently—for example, after every M update operations or after every T seconds. This approach has two advantages:

(1) It does not require any locking or context switches since it can be coded as a procedure within the foreground process.
(2) Very efficient restructuring code can be developed since we can make the assumption that the data structure will not be modified by any other process while it is being restructured.

The disadvantage, of course, is that maintenance is not invoked only during idle periods. Bastani, Hilal, and Chen[10] analyze the case where the maintenance is done after every M update requests. As shown in Figure 7, as M increases, the average cost (per request) of maintenance decreases, while the average service time increases. Hence, there is a value of M that optimizes system performance.

We have compared the performance of concurrent maintenance with periodic maintenance experimentally. Our measure of performance is the total processing time for requests, defined as the sum of the waiting time and the service time of requests at the server and the request and response transmission times. The results indicate that periodic maintenance yields a better average processing time while concurrent maintenance gives a smaller maximum response time. Hence, periodic maintenance is the best approach for applications that do not require real-time response, assuming that the arrival process can be modeled as a Poisson process. The algorithms for periodic maintenance are relatively efficient as well as simple. Concurrent maintenance with stochastically scheduled complete maintenance has a small variance. Therefore, concurrent maintenance is viable for real-time applications, especially if the operating system provides efficient process synchronization and context switch facilities.

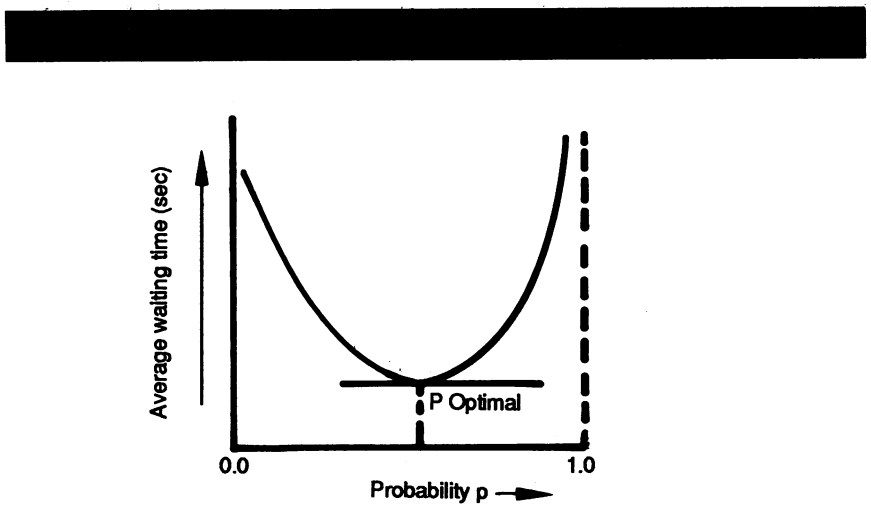

Figure 6. Optimal parameter for stochastic maintenance.

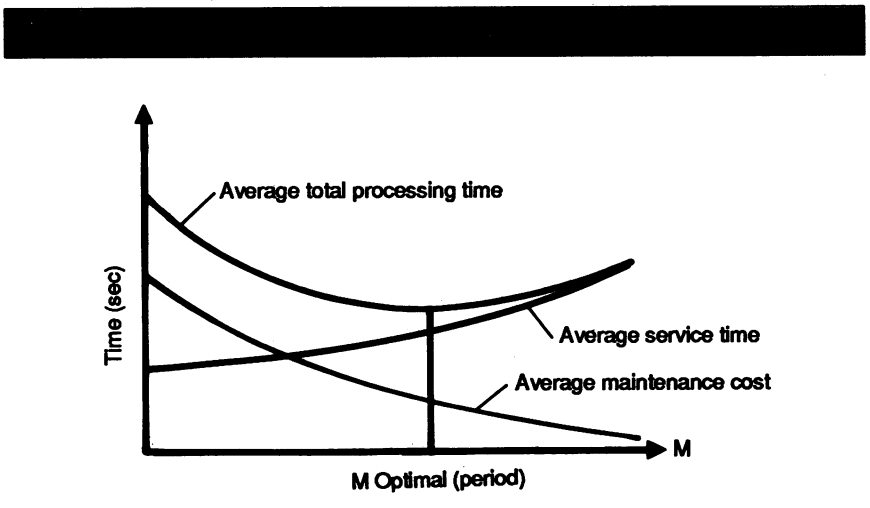

Figure 7. Optimal parameter for periodic maintenance.

Parallel maintenance. A restricted version of multilevel data structures can be maintained using several processes running simultaneously on different processors. Let $S(D)$ be some measure of the amount of information stored in a data structure D. For example, for the data structure shown in Figure 1, $S(D)$ can be the number of elements stored in the data structure.

Definition. A restricted multilevel data structure D consists of a sequence of data structures $D_1, D_2, \ldots, D_n$, with the following properties:

(1) $S(D_i)$ can be decreased by increasing $S(D_{i+1})$ without changing the information contained in D,
(2) decreasing $S(D_i)$ by increasing $S(D_{i+1})$ results in an improvement in the performance of D, and
(3) restructuring D_i does not require restructuring $D_j, j \neq i$.

The data structure shown in Figure 1 does not satisfy the third condition of this definition. In fact, restricted multilevel data structures are limited to the implementation of abstract data types that deal with aggregates such as sets, bags, and

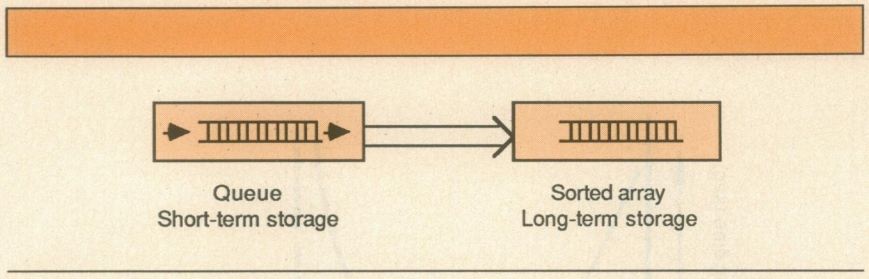

Figure 8. A restricted multilevel data structure.

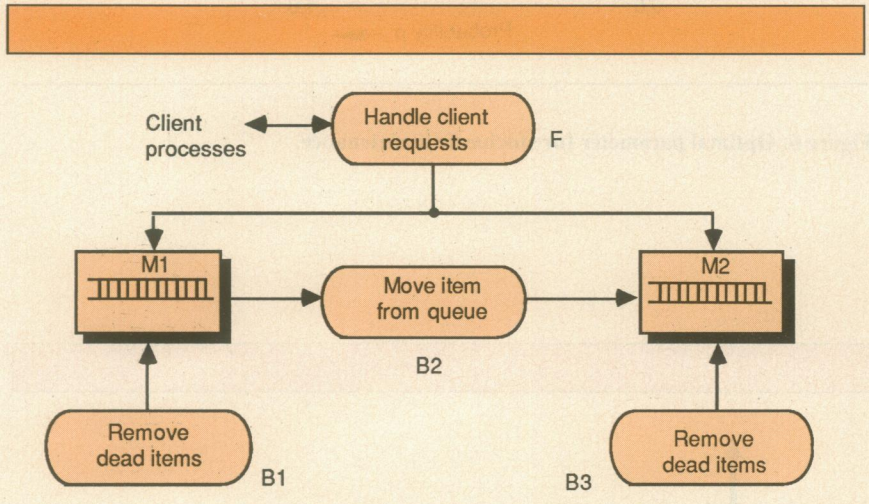

Figure 9. Parallel maintenance of a restricted multilevel data structure.

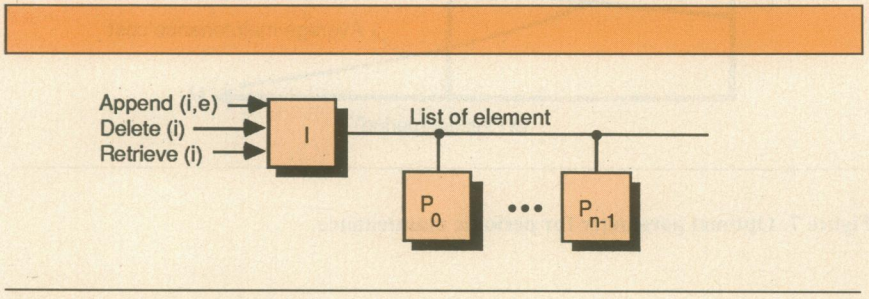

Figure 10. Parallel implementation of the linear list component.

search tables. However, this class includes a large number of data structure combinations such as queue combined with a binary search tree, sorted array, hash table, balanced binary tree, B-tree, etc. The class of restricted multilevel data structures is a subset of the class of multilevel data structures since $\forall i, 1 \leq i \leq n$, we can define $\mathbf{I}_i$ as $\mathbf{I}_i \equiv \forall j,\ 1 \leq j \leq i,\ S(D_j)$ is minimum.

As an example of restricted multilevel data structures, assume that we have to implement a set abstract data type. One possible data structure is shown in Figure 8. A sorted array is used for fast lookup and a queue for fast insertion. Also, a tag assigns the value "dead" or "alive" to each item. The tag is used for achieving efficient deletion. This data structure satisfies the above definition since:

(1) The size of the queue can be decreased by moving some keys from the queue to the sorted array, thus increasing the size of the sorted array without affecting the information contained in the combined data structure.

(2) This change improves the performance of the combined data structure.

(3) We can restructure the queue or the sorted array (for example, by removing dead items) without having to modify the sorted array or the queue, respectively.

Restricted multilevel data structures have the important feature that it is possible to use multiprocessors to achieve efficient implementations. For example, one processor can remove dead entries from the queue, another processor can remove dead entries 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 Figure 9 where the queue is stored in memory bank M1 which is shared by processors F, B1 and B2, and the sorted array is contained in memory bank M2 which is shared by F, B2 and B3.

Parallel implementation of abstract data types

The multilevel data structures discussed above yield good average response time if maintenance strategies are selected carefully. But, under a heavy load, performance may become worse than it would be with a conventional data structure. In this section, we discuss an approach to implementing abstract data type components based on broadcasting sequential processes.[5] Each process runs on a separate dedicated processor and can broadcast messages that will be delivered to all processes in the broadcast group. Clients access the abstraction via an interface process. This approach has the potential for providing very short response time for all the operations of the abstract data type. Further, the time required is independent of the size of the data structure, and the program is also quite straightforward, at least for basic abstract data types such as stacks, queues, lists, sets, and symbol tables.

Example 1. Figure 10 shows n processes ($P_0, P_1, \ldots, P_{n-1}$) implementing the linear list abstract data type component discussed earlier. I is the interface process that accepts client requests. Each process P_i has a counter denoted by ci, a status flag denoted by si, and an element denoted by ei. The status flag si takes on two values—"occupied" if ei contains a valid value, and "free" if it does not. If si is "occupied," then ei is the cith element in the linear list. If si is "free," then P_i is the cith process on the stack of free processes. The process with counter 0 is at the head of the stack. Initially, $ci = i$ and si = "free" for $0 \leq i \leq n-1$.

The algorithm for each P_i is shown in Figure 11. The primitive **send** e transmits message e to the interface process while **skip** denotes a null action. This algorithm is an efficient constant time algorithm. Its correctness can be easily proven by specifying the representation (or data structure) invariant and showing that it is established initially and reestablished after the completion of each update operation. The mapping function relates the value-returning functions (in this case only ith) to the representation.[11] The representation invariant and the mapping function of the implementation shown in Figure 11 are given below.

Let FREE = {i|si = free}
Representation invariant:
(1) ∀j, 1 ≤ j ≤ |FREE| ∃ k:
sk = free ∧ ck = j − 1
(2) ∀j, 1 ≤ j ≤ n − |FREE| ∃ k:
sk = occupied ∧ ck = j
Mapping function:
ith(i) = ek where sk = occupied ∧ ck = i

If I does not receive an OK (for Delete and Append) or an element value (for ith) in the next cycle, it raises the appropriate exception. Because only one process can respond for each command, there is no delay for collision resolution. The code for the interface process is shown in Figure 12. It uses a variable named "size" (initialized to 0) to track the number of elements in the list. The primitive **broadcast** e transmits message e to all processes in the broadcast group.

Example 2. As another illustration, consider the problem of implementing a SearchTable abstract data type to 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.

```
repeat
    await command
    case command of
        ith(i)→
            si = occupied and ci = i→send ei
            si ≠ occupied or ci ≠ i→skip
        Delete(i)→
            si = occupied and ci < i→skip
            si = occupied and ci = i
                →si: = free; ci: = 0; send OK
            si = occupied and ci > i→ci: = ci − 1
            si = free→ci: = ci + 1
        Append(i,e)→
            si = occupied and ci > i→ci: = ci + 1
            si = occupied and ci ≤ i→skip
            si = free and ci > 0→ci: = ci − 1
            si = free and ci = 0
                →si: = occupied; ei: = e; ci: = i + 1; send OK
    end case
forever
```

Figure 11. Code for the linear list component.

```
ith(i)→
    if i < 1 then raise IndexUnderflow
    elsif i > size then raise IndexOverflow
    else
        begin
            broadcast ith(i)
            if response received in the next cycle then send element to client
            else raise HardwareFailure
        end;

Delete(i)→
    if i < 1 then raise IndexUnderflow
    elsif i > size then raise IndexOverflow
    else
        begin
            broadcast Delete(i)
            if no response received in the next cycle then raise HardwareFailure
            else size: = size − 1
        end;

Append(i,e)→
    if i < 0 then raise IndexUnderflow
    elsif i > size then raise IndexOverflow
    else
        begin
            broadcast Append(i,e)
            if no response received in the next cycle then raise ListOverflow
            else size: = size + 1
        end;
```

Figure 12. Interface process for the linear list component.

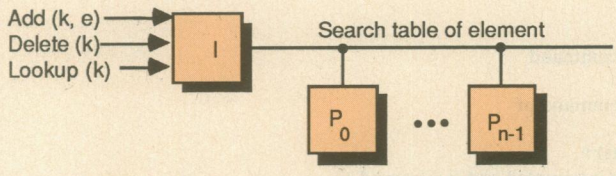

Figure 13. Parallel implementation of the search table module.

```
repeat
    await command
    case command of
        Add(k, e)→
            ci = 0→ki = k; ei: = e; ci: = −1
            ci > 0→ci: = ci − 1
            ci < 0→skip
        Delete(k)→
            ci < 0 and ki = k→ci: = 0
            ci < 0 and ki ≠ k→skip
            ci ≥ 0→ci: = ci + 1
        LookUp(k)→
            ci < 0 and ki = k→send ei
            ci ≥ 0 or ki ≠ k→skip
    end case
forever
```

Figure 14. Code for search table component.

```
Add(k,e)→
    broadcast LookUp(k)
    if no response received in the next cycle then broadcast Add(k,e)
    else raise DuplicateKey

Delete(k)→
    broadcast LookUp(k)
    if response received in the next cycle then broadcast Delete(k)
    else raise NoSuchKey

LookUp(k)→
    broadcast LookUp(k)
    if response received in the next cycle then send element to caller
    else raise NoSuchKey
```

Figure 15. Interface process for the search table component.

• LookUp(k)→return the element associated with key *k*.

Standard implementations of a SearchTable abstract data type cannot simultaneously optimize LookUp, Add, and Delete. For example,

(1) an unsorted array is poor for LookUp and Delete but good for Add;
(2) a sorted array is good for LookUp but poor for Add and Delete;
(3) a binary search tree has a much better Add and Delete performance than a sorted array, but it has a slightly worse LookUp time;
(4) a balanced tree improves the LookUp time of a binary search tree at the expense of Add and Delete;
(5) the performance of a hash table using open addressing becomes poor as the number of Delete requests increases, unless it is restructured periodically; similarly, a hash table using chaining deteriorates in performance as the individual lists get longer.

Figure 13 shows the structure for an implementation of the SearchTable component using broadcasting sequential processes. Each *Pi* contains a key (*ki*), an element (*ei*), and a counter (*ci*). Initially, $ci = i$ for $0 \le i \le n-1$. The code for each *Pi* is shown in Figure 14. Process P_i contains an element if $ci = -1$; otherwise it is the *ci*th process on the stack of free processes. The code for the interface process is shown in Figure 15. This implementation is simpler than most standard implementations. At the same time, since there is no need for collision resolution, this implementation has a very rapid response time.

Implementation considerations. The above method allows us to implement abstract data types in the form of hardware components using VLSI technology. The interconnection network is simple, and all the processors execute the same code. Further, the code is short, and the memory requirement per processor is just a few words. Hence, several processors can be packed onto one chip. This method can significantly improve the performance of heavily used systems software. For example, a priority queue component can be used to speed up operating system schedulers, a symbol table component can be used for compilers, and search table components can be used for implementing log files and data dictionaries in database systems.

Some trade-offs involved in the VLSI implementation of abstract data types concern the interconnection structure for the processors, the placement of the code, and the limitation imposed on the size of the abstract data type by the number of processors on a chip. Two possible interconnection options are (1) to connect all the processors to the same bus and (2) to have a tree-structured bus. The performance in the latter case is better than in the former, since the signal traverses a shorter distance. However, if the bus loops back to the interface processor in the former case, the interface processor can monitor the bus after transmitting a request to verify that there are no transmission errors. This option can significantly improve system reliability without incurring the performance penalty that reliable broadcast protocols incur.

The size of each processor can be reduced by placing its code in the interface processor. In this case, the interface processor broadcasts the request and the code associated with it. Each processor has only the CPU and its private data. This method allows packing a larger number of processors on one chip but increases the service time. Another possibility is to replicate the code at each processor, but this reduces the number of processors available for the abstract data type. An alternative approach is to replicate the code for the frequently executed commands (for example, LookUp) and to broadcast the code for the remaining commands (for example, Add and Delete).

The limitation due to the number of processors on a chip can be overcome by using both broadcasting sequential processes and conventional data structures. When the processors are all occupied, the interface processor can store information in its own memory. Whenever it receives a request, it can broadcast it to the processors while simultaneously searching its own data structure. If it receives a positive response from one of the processors, it can abort its own search; otherwise it can continue to process the request itself. When a processor becomes free, the interface processor can move some information from its data structure to the free processor. In this way, the broadcasting sequential processes can serve as a cache for the abstract data type. This procedure improves system flexibility and allows use of a standard chip for applications where the size of the abstract data type may exceed chip capacity.

The implementation using broadcasting sequential processes can be made fault-tolerant to processor failures by using redundancies. However, unlike methods for conventional data structures, there is no central processor to detect and repair errors in the data structure. One strategy is to use primary and secondary processors for storing each item. (The number of processors need not be doubled since a processor can be the primary processor for one item and the secondary processor for another item.) Also, the response period for each request that is broadcast by the interface processor can be extended to include responses from both the primary and secondary processors. In this way, if either response is absent, a free processor can assume the identity of the failed processor for that item. This provision enables the system to recover from a single processor failure. Periodic unsolicited responses can also be generated by both the primary and secondary processors to reduce the likelihood of having more than one unrecovered failure at any given time.

In this article, we have considered efficient implementation of abstract data type components for two classes of programs. The motivation for implementing these abstract data type components is to overcome the frequent performance loss of abstractions while preserving their advantages—comprehensibility, modifiability, provability, and reusability. For distributed systems, multilevel data structures allow efficient implementation of all the operations of the abstract data type. For parallel environments, a fast implementation of some abstract data types can be achieved using broadcasting sequential processes.

Several interesting research issues remain, especially in the parallel implementation of abstract data types. These include investigating

(1) whether architectures other than broadcasting sequential processes can be used to implement abstract data types;
(2) what range of abstract data types can be efficiently implemented on each architecture; and
(3) what methods tolerate memory failures, processor failures, and transient transmission errors without greatly degrading performance. □

Acknowledgments

The authors gratefully acknowledge several fruitful discussions with I. Chen, T. Law, D. Leu, A. Moitra, J. Teng, and I. Yen. The authors also wish to thank the six anonymous referees for their extensive comments which have greatly improved the quality of the article. This article was supported in part by NSF Grant MCS-83-01745.

References

1. P. Wegner, "Capital-Intensive Software Technology," *IEEE Software*, July 1984, pp. 7-45.
2. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, Dec. 1972, pp. 1053-1058.
3. B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Trans. Programming Languages and Systems*, July 1983, pp. 381-404.
4. D.L. Parnas and D.P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Comm. ACM*, July 1975, pp. 401-408.
5. N.H. Gehani, "Broadcasting Sequential Processes," *IEEE Trans. Software Engineering*, June 1984, pp. 343-351.
6. F.B. Bastani et al., "Impact of Parallel Processing on Software Quality," *Proc. First Int'l Conf. Supercomputing Systems*, St. Petersburg, Fla., Dec. 1985, pp. 369-376.
7. B.W. Lampson, "Hints for Computer System Design," *IEEE Software*, Jan. 1984, pp. 11-28.
8. E.W. Dijkstra et al., "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Comm. ACM*, Nov. 1978, pp. 966-975.
9. C.B. Jones, "Tentative Steps Towards a Development Method for Interfering Programs," *ACM Trans. Programming Languages and Systems*, Oct. 1983, pp. 596-619.
10. F.B. Bastani, W. Hilal, and I.R. Chen, "Performance Analysis of Concurrent Maintenance Policies for Servers in a Distributed Environment," *Proc. FJCC 1986*, Computer Society Press, Los Alamitos, Calif., pp. 611-619.
11. J.V. Guttag, "Notes on Type Abstraction" (version 2), *IEEE Trans. Software Engineering*, Jan. 1980, pp. 13-23.
12. F.B. Bastani and C.V. Ramamoorthy, "A Methodology for Assessing the Correctness of Control Programs," *Computers and Electrical Engineering*, Pergamon Press, Nov. 1984, pp. 115-144.
13. D. Coleman, J.W. Hughes, and M.S. Powell, "A Method for the Syntax-Directed Design of Multiprograms," *IEEE Trans. Software Engineering*, Mar. 1981, pp. 189-196.
14. P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Trans. Software Engineering*, May 1982, pp. 250-269.

Farokh Bastani is an associate professor in the Dept. of Computer Science, University of Houston, University Park. His research interests include software design and validation techniques, distributed systems, and fault-tolerant systems. He is presently investigating methods of developing inherently fault-tolerant programs for critical applications.

Bastani received the PhD degree in computer science from the University of California at Berkeley in 1980 and was a visitor there during the 1986-87 academic year.

Wael Hilal is currently an assistant professor in the Dept. of Computer Science, University of Houston, University Park. His research interests are computer-communication protocols, routing and flow control, local area networks, performance evaluation and modeling of computer networks, integration of voice and data on networks, and distributed operating systems.

Hilal received a BS and MS in computer science from the University of Alexandria, Egypt in 1976 and 1979 respectively. He received his PhD in computer science from Ohio State University in 1984.

S. Sitharama Iyengar is currently professor of Computer Science and supervisor of robotics research at Louisiana State University. He has authored more than 60 articles in parallel algorithms, data structures and design and navigation of intelligent mobile robot. His current research interests are applying neural network techniques for path planning and learning in mobile robots.

Iyengar received his PhD in Engineering from Mississippi State. He is presently a visiting scientist at Oakridge National Laboratories.

Readers may write to Bastani at the Department of Computer Science, University of Houston, University Park, Houston, TX 77004.