Systematic Incorporation of Efficient Fault Tolerance in Systems of Cooperating Parallel Programs *

I-Ling Yen
Department of Computer Science
Michigan State University
East Lansing, MI 48824-1027

Farokh B. Bastani
Department of Computer Science
University of Houston
Houston, TX 77204-3475

Abstract

Cooperating parallel programs are being increasingly used in critical applications that require both high performance and high reliability. A promising technique for simultaneously achieving these objectives is to embed the fault tolerance within the program instead of superimposing it via external mechanisms. We develop one such approach for a group of processes that cooperate via shared data structures. The scheme uses data structures having two or more invariant assertions. When the strong invariant is true, the performance is good. When it is false, the performance may be adversely affected, but it is guaranteed that the system will operate correctly provided the weak invariant is true. The algorithms are designed to ensure that processor failures will never cause the weak invariant to be false and to restore the strong invariant within a finite number of recovery actions. We develop a robust task handling mechanism to support the approach and illustrate it for three common data structures.

1 Introduction

There are two methods of making a program fault-tolerant, namely, by superimposing a general mechanism on top of the program, such as rollback recovery [5] [16], TMR, or general self-stabilization [13], or by integrating the fault tolerance within the algorithm, such as algorithm-based fault tolerance [11], application-specific self-stabilization [9] [17], or inherent fault tolerance [4]. The advantage of the latter methods is that the program can be designed to incur very little overhead under failure-free situations. The disadvantage is that these methods have to be developed from scratch for each algorithm. This limits their applicability in practice.

It may be possible to develop systematic, if not automated, methods of incorporating efficient fault tolerance within specific classes of programs. For example, a general method for achieving high performance fault-tolerant SIMD and SPMD programs from the corresponding fault-intolerant versions is given in [25] and a method of developing algorithm-based fault-tolerant circuits is presented in [24]. In this paper, we develop a method similar to stepwise refinement for achieving efficient fault tolerance for a class of MIMD program. The program consists of a collection of servers that operate on a shared data structure. Clients issue requests to the servers and it is guaranteed that all the requests will be processed eventually as long as there is at least one active processor. The overhead under failure-free situations is moderate, consisting of simple integrity checks and restructuring operations. Once a processor failure occurs, the performance may deteriorate for a period of time and then the system readjusts to reduce the overhead.

The approach works as follows. Normally, the data structure of an abstract data type is characterized by a representation invariant which specifies formally all possible legitimate states of the data structure. When the data structure is accessible by several processes, before every access a process can assume that the invariant assertion is true; this is called its rely condition [12]. Similarly, the process must ensure that the assertion is true after it completes the operation; this is called its guarantee condition [12]. For this type of shared data structures, the rely and guarantee conditions are identical.

When a processor fails, it will not be able to reestablish the invariant if it has violated it and this can cause failures in other processes. Our approach is to replace the invariant assertion by two assertions, I_1 and I_2, such that $I_1 \Rightarrow I_2$. (This can be extended to a series of invariant assertions, $I_1, I_2, ..., I_q$, $q \geq 2$, such that $I_1 \Rightarrow I_2 \Rightarrow ... \Rightarrow I_q$.) I_1 is called the strong invariant and I_2 is called the weak invariant. During failure-free situations, the code should ensure that the strong invariant is true. Whenever the strong invariant is true, the performance is good. When a processor fails, the strong invariant may be violated, but

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003652139.

0363-8928/94 $3.00 © 1994 IEEE

the code should guarantee that the weak invariant is true. If the strong invariant is false, the performance of the system may be affected; however, the system will operate correctly provided that the weak invariant is true. If there are no further processor failures, then incomplete updates can result in at most a finite number of recovery actions by other processors before the strong invariant is reestablished, either via data structure integrity checks and recovery or incrementally in a manner similar to self-stabilization.

The rest of the paper is organized as follows. Section 2 presents the system model and definitions. Section 3 develops a general algorithm for the case where a new data structure instance is created. Section 4 develops algorithms for several data structures to tolerate processor failures during *in situ* updates. Finally, Section 5 summarizes the paper and outlines some research directions.

2 System model

The global view of the system is shown in Figure 1. There is a set of clients, $C = \{c_1, c_2, ..., c_m\}$, that send requests to a set of servers, $S = \{s_1, s_2, ..., s_n\}$, implementing some object. Any server can process any request and each request is processed completely by one server. The servers access a common data structure such as a tuple space [1] or blackboard. The goal is to tolerate the failure of any number of servers such that the overhead during normal operating situations is as small as possible.

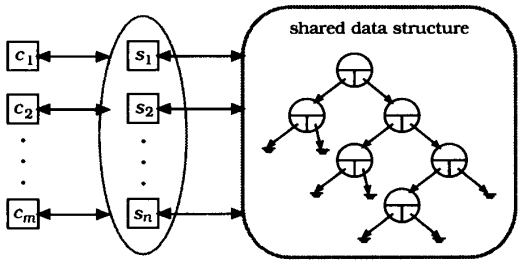

Figure 1: System model.

This model is appropriate for shared memory MIMD and SPMD parallel systems. This includes systems connected via omega networks such as the BBN Butterfly as well as via cross-bar (e.g., Alliant FX/8) and high speed optical networks [23]. We assume that processors fail cleanly (failstop behavior) and that processor failures can be detected. For example, all working processors can be organized in a ring structure with periodic "I am alive" message exchanges between neighbors. Time out can be used to detect failures and reconfigure the ring.

As an example, consider the task of multiplying two matrices, $A := B \times C$. For performance speed-up, the matrices can be partitioned into submatrices resulting in a number of parallel subtasks. These are then processed by the servers in parallel. Another example is a group of servers adding entries to or removing entries from a search table. Efficient ways are known for doing this in parallel [6] [25].

These two examples illustrate two different canonical situations which nevertheless share some common characteristics. In the first case, the system creates a new object in parallel. Hence, the failure of a processor will only mean that some task has to be performed again. In the second case, the failure of a processor can also result in corruption of the shared data structure.

The basic behavior of the system is as follows.

```
client: enqueue request; wait for response;
server: loop
          dequeue request; process request;
          send response to client;
        end loop;
```

The possible effects of a server failure include the following:

1. No effect if it fails before dequeueing a request or after sending the response.
2. Incomplete request if it fails after dequeueing a request and before sending the response.
3. Violation of the data structure invariant if it fails after partially modifying the shared data structure.
4. Lockout if it fails while holding a lock.
5. Indefinite waiting if a process is waiting for it.

The approach we use to tolerate server failures is to use data structures requiring weaker constraints. Problems (2), (4), and (5) are application-independent and we develop a fault-tolerant algorithm to tolerate these failures in Section 3. Problem (3) depends on the application. To overcome this problem, we try to replace the data structure by a multilevel data structure defined as follows [3].

Definition: A multilevel data structure D is a data structure for which there is a sequence of invariants $I_1, I_2, \ldots, I_q$, such that

1. D always satisfies I_q,
2. $I_1 \Rightarrow I_2 \Rightarrow \cdots \Rightarrow I_q$,

3. if D satisfies I_j but not I_{j-1} for $j > 1$ then it can be modified to obtain D' such that (a) D' satisfies I_{j-1}, (b) the output of any operation on D' is identical to the output of that operation on D, and (c) the performance of the program using D' is better than that using D.

Here, we consider $q = 2$ and refer to I_1 as the strong invariant of D and I_2 as the weak invariant of D. The weak invariant must be chosen so that it is possible to write the code for each operation to guarantee that a server failure will never violate it.

As an example, consider a simple array implementation of sets. Let a *status* field, initialized to *empty*, be associated with each entry in the table. It is set to *occupied* when a key is stored at that location and *empty* when that key is removed, making both of these operations atomic. A variable v stores the largest index after which all locations are *empty*. The strong invariant, I_1, asserts that there are no *empty* locations between 1 and v while the weak invariant, I_2, allows such *empty* locations. These can be specified formally:

$$I_2 \equiv \forall i : v + 1 \leq i \leq n :: T(i).status = empty$$

$$I_1 \equiv I_2 \wedge \forall i : 1 \leq i \leq v :: T(i).status = occupied.$$

The strong and weak invariants for more complicated data structures are given in Section 4 along with corresponding fault-tolerant update and recovery algorithms.

3 Data structure creation

Consider an assignment statement,

<identifier>:= <expression>,

that can be decomposed into several subtasks having the following characteristics.

1. The subtasks are independent of each other.
2. They are idempotent, i.e., multiple executions are equivalent to a single execution.

One way of satisfying these conditions is to ensure that each subtask works on a different portion of the data structure, <identifier> does not occur in <expression>, and evaluation of <expression> has no side-effects. This can be done easily for matrix computations where each submatrix can be computed independently.

3.1 Fault-intolerant algorithm

We first present a fault-intolerant algorithm for the problem. In this algorithm, a FIFO queue Q, implemented using a circular array of size L, is used to maintain the requests that are ready to be executed. Two pointers associated with Q are Q_h and Q_t which indicate the starting locations from which elements can be removed and inserted, respectively. The queue is empty when $Q_h = Q_t$ and full when $(Q_t + 1) \bmod L = Q_h$. Each entry Q_i, for i satisfying

$$\{(Q_h \leq Q_t) \wedge (Q_h < i \leq Q_t)\} \vee$$
$$\{(Q_h > Q_t) \wedge [(0 \leq i \leq Q_t) \vee (Q_h < i < L)]\},$$

contains the request ID of the request stored there. A client obtains a ticket, in parallel with other clients, to access a free location in the queue. It then enqueues its task at the location specified by the ticket and waits for a response from a server. Likewise, a server obtains a ticket, in parallel with other servers, to access an occupied location in the queue. It then dequeues the task stored at that location, processes it, and sends a response to the client. In the following, we consider an efficient parallel ticket assignment algorithm. This is made fault-tolerant in Section 3.2.

Let P^R denote the set of servers that have completed their requests and have to remove new requests from Q and P^I denote the clients that have generated new requests which they have to add to Q. These two types of accesses to Q, by processors in P^R and P^I, will be handled at different times. The system operates in a quasi-synchronous mode with a predetermined time quantum t. Processors in P^R and P^I alternately enter a competition phase for accessing the request queue once every t time units. The processors in P^R or P^I request tickets which grant access to a location in the request queue for request retrieval or insertion. A tournament type of computation is used to determine the total number of tickets being requested. In each round, a pair of processors compete for a critical section and the winner performs the operation. The final winning processor obtains the desired number of consecutive tickets on behalf of all requesting servers. The tickets are then distributed by traversing down the tree.

To simplify the presentation, assume that $N = 2^j$ for some $j \geq 1$. Let $C_1, C_2, ..., C_{N-1}$ denote the $N - 1$ critical sections. These critical sections are organized in a tree structure, with $C_{N/2}, ..., C_{N-1}$ as the leaves (at level 1) and C_1 as the root of the tree (at level $\log N$). During the ith stage of the competition phase, processors will compete for critical sections at level i of the tree. The total number of processors

that wish to access Q for fetching requests or inserting requests is determined during the competition phase and consecutive tickets corresponding to the locations in Q are granted for accessing the queue in parallel. There are two storage locations, Lt_i and Rt_i, associated with each critical section C_i. These are used to store the number of tickets requested by processors in the left and right subtrees. Lt_i and Rt_i are initialized to zero at the beginning of each competition phase. The structures of the critical sections and the processors are shown in Figure 2. There are $\log N$ iterations. In each iteration, winning processors compute the total number of tickets required in the entire subtree and then compete for critical sections at the next level. Let P_w be the final winning processor, which is the processor that holds lock C_1. At the end of the competition phase, the number of requests requested by all the processors is T_w.

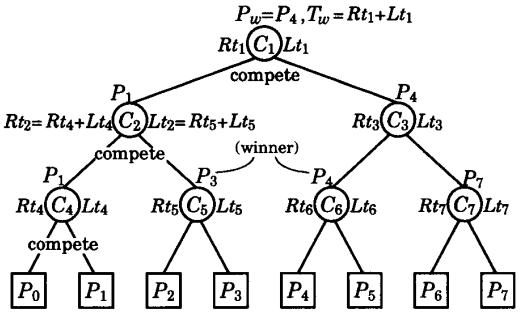

Figure 2: Critical sections and servers.

After computing T_w, the ticket granting phase starts. For request removal, tickets with values $Q_h + 1$ through $Q_h + \min(T_w, K)$ will be granted, where K is the number of requests in Q, $K = (Q_t - Q_h) \bmod L$. Also, the head pointer of Q is adjusted by setting $Q_h \leftarrow (Q_h + \min(T_w, K)) \bmod L$. For request insertion, the tickets issued have values Q_t through $Q_t + \min(T_w, L - 1 - K) - 1$. Q_t is set to $(Q_t + \min(T_w, L - 1 - K) \bmod L)$. These tickets are distributed to all the requesting processors by starting from the top of the tree and proceeding to the leaves. If there are fewer tickets than T_w, then some processors will not get tickets and must try again in the next round. Once the ticket is granted, processors can access the corresponding locations in the queue for request insertion or removal.

3.2 Fault-tolerant algorithm

There is no problem if a processor fails before starting any access procedure. In the following, we consider the weakening of various invariants to handle other failures.

First, a processor requesting a ticket can go down before a ticket is issued. The ticket(s) for the failed processor and all processors waiting for it are discarded. The consequence of this is that a location currently being accessed may have an unremoved request (during the addition operation) or may not have any available requests (during the fetch operation). In these cases, the processor tries again in the next round.

Second, its failure can result in permanent locking of some portion of the data structure. To address this, a weaker lock is used which can be broken if the processor holding it has failed.

Third, some processors may be waiting for the failed processor. A weaker form of waiting is used in which a processor will only wait for a certain amount of time and then stop waiting.

Fourth, a processor can go down during the execution of a request. To ensure that the request is not lost, the request being removed for execution is maintained in the queue as long as its execution is incomplete. The ticket granting algorithm is modified so that T_w tickets are granted irrespective of the length of the queue even for request removal. Hence, processors will have the chance to check whether there are unremoved requests in the queue due to the first type of failures as well as unfinished requests in the queue due to the fourth type of failures.

3.2.1 Recoverable locks

We constrain locks to be acquired and released via special pointer or index variables. These are declared in the usual way except that the key word **lock** is inserted before the type declaration. The records or array elements that they point to must be declared as **lockable**. In the following, we illustrate this using pointers; the case for index variables is similar.

type *node* **is lockable record** ... **end**;
type *lock_ptr_to_node* **is lock access** *node*;
var *p,q,r: lock_ptr_to_node*;

Lock pointers have a parameter, *LOCK*, associated with them as well as a special field, *lock*, associated with the records that they point to. Both of these are 1-bit flags which are used to lock or unlock the record. The compiler must allocate these variables in the same memory unit and must initialize each *LOCK* parameter to 1 and each *lock* field to 0. We assume that all lock pointers for a processor are stored in an array called *lock_pointer_array* which can be accessed by other processors for recovery purposes. The code

for locking, unlocking, and recovery are shown in the following:

```
lock(p):    loop
              swap(p.lock,p'LOCK);
            until p'LOCK = 0;

unlock(p): swap(p.lock,p'LOCK);

recovery:  var r: lock_ptr_to_node;
           for i:=lock_pointer_array'START to
             lock_pointer_array'END loop
             if lock_pointer_array(i) is locked then
               transfer_lock(lock_pointer_array(i),r);
               if r'LOCK = 0 then unlock(r); end if;
             end if;
           end loop;
```

The codes for operations *lock(·)* and *unlock(·)* use a primitive **swap** operation which exchanges the contents of two bits in one cycle. When a processor detects that another processor has failed, it scans through the lock array for that processor and releases all the locks held by it. The recovery code is designed to tolerate the failure of the processor performing the recovery action at any point during the recovery procedure. The global invariant maintained is that a lock is held by only one processor (dead or alive) at any given time. This is achieved in two steps by procedure *transfer_lock(p,q)*:

$q := p$; **swap**(*p'LOCK,q'LOCK*);

Here, first q is made to point to the node that p is pointing to. Then, an atomic **swap** operation is used to transfer the lock from p to q. Busy waiting is not required since no other processor will access p or q as long as this processor is alive.

With recoverable locks, the algorithm sketched above for request insertion into or removal from request queue Q is robust. It can tolerate fail-stop processor failures at any stage of the computation. Also, the algorithm requires $O(\log N)$ time when there is no failure. Both the tournament and ticket granting phases require $\log N$ iterations, and each iteration requires only constant time. The access phase requires only constant time. When there is a failure, a processor may need to retry by participating in the next tournament and ticket granting phases.

4 Data structure updates

In this section, we show how data structures satisfying weaker invariants can be used to tolerate processor failures. We consider three common data structures, namely, single-linked list, double-linked list, binary search tree. The single-linked list data structure does not require a weaker invariant but, because of its simplicity, helps in understanding the other data structures.

4.1 Single-linked linear list

The data structure is shown in Figure 3(a). The list is empty when *head.next* = **nil**. The invariant assertion is $ok(head.next)$ where

$$ok(p) \equiv p = \textbf{nil} \vee [p \neq \textbf{nil} \wedge valid(p.info) \wedge ok(p.next)].$$

Note: *valid(p.info)* is true if *p.info* contains a legitimate value.

The client can issue two operations, $insert(x)$ and $remove(x)$. For $insert(x)$, item x is added to the list. For the linear list, it is simply added at the head of the list. For $remove(x)$, the list is searched for x and it is removed from the list. Since multiple processors can access the list concurrently, locking is used to ensure mutually exclusive access to nodes in the list. However, should a processor fail, the lock can broken as described in Section 3.2.1.

The code for $insert(x)$ is as follows (see Figure 3(b)).

```
new(r); r.info := x; p := head; lock(p);
r.next := p.next; p.next := r; unlock(p);
```

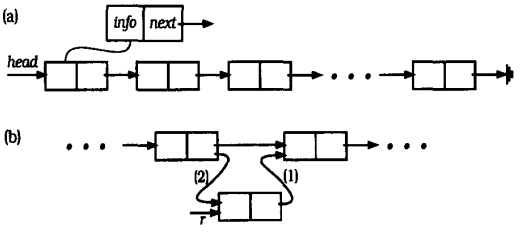

Figure 3: Single linked linear list.

The only critical statement is *p.next* := *r*. (A critical statement is one that modifies the shared data structure.) We make the reasonable assumption that a write operation is atomic. If this is not provided by the underlying hardware, then it can be implemented using the method proposed in [14]. In this case, the only effect of a failure is that the request will not be processed. If so, then another processor will process the request using the request queueing method presented in Section 3. Finally, there is no extra overhead in making this code fault-tolerant.

The code for $remove(x)$ is as follows.

```
p := head; lock(p); q := p.next;
while q ≠ nil and then q.info ≠ x loop
   lock(q); unlock(p); p := q; q := p.next;
end loop;
if q ≠ nil then
   lock(q); p.next := q.next; unlock(p); unlock(q);
end if;
```

The critical statement is $p.next := q.next$ which again is a write operation to a single word which we assume is atomic. There is no overhead in this code. However, we assume that the system has some mechanism for garbage collection. This imposes some overhead even during failure-free periods. Garbage collection is also needed for nodes lost due to failures that occur during *insert* operations.

4.2 Double-linked linear list

The data structure satisfying the strong invariant is shown in Figure 4(a). Each node contains a *status* field in addition to the usual *next* and *prev* pointers and the *info* field. The status field for the header contains -1. For the other nodes, it is 0 if the node is in the list and 1 if it has been removed from the list. The strong invariant is true when the status field has the value 0 for all nonheader nodes and adjacent nodes point to each other. More formally, the strong invariant is $ok\_strong(head.next)$ where

$$ok\_strong(p) \equiv [(p = head) \wedge (p.next.prev = p)]$$
$$\vee\ [(p \neq head) \wedge (p.tag = 0) \wedge valid(p.item) \wedge$$
$$(p.next.prev = p) \wedge ok\_strong(p.next)]$$

The weak invariant (Figure 4(b)) is selected so that it can be maintained when *insert* and *remove* operations are performed. Also, correct traversals can be done in the forward and backward directions even when only the weak invariant is true (though it may be slower).

$$ok\_weak(p) \equiv [(p = head) \wedge (p.next.prev = p)]$$
$$\vee\ [(p \neq head)$$
$$\wedge\{[p.tag = 0 \wedge valid(p.item)] \vee [p.tag = 1]\}$$
$$\wedge\{p.next.prev = p \vee p.next.prev = head$$
$$\vee(p.tag = 1 \wedge reachable(p, p.next.prev))\}$$
$$\wedge ok\_weak(p.next)]$$
$$reachable(p, q) \equiv (p = q) \vee (p \neq q \wedge reachable(p, q.next))$$

The list is empty when all nodes reachable from *head*, other than *head*, have $status = 1$.

The code for $insert(x)$ is shown in the following.

```
p := head;
new(r); r.prev := p; r.info := x; r.status := 0;
lock(p); q := p.next; r.next := q;
p.next := r; q.prev := r; unlock(p);
```

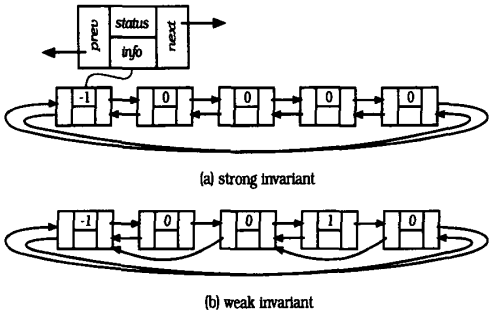

Figure 4: Double linked list data structure.

The critical statements are $p.next:= r$ and $q.prev:= r$. If the processor fails after $p.next:= r$ but before $q.prev:= r$, then the *prev* pointer of q will be in error. This does not affect forward traversal, but does affect backward traversal. However, the error can be detected since $q.prev.next \neq q$, so the traversal continues from $q.prev.next$ and proceeds in the forward direction till it reaches a node r such that $r.next = q$.

The code for $remove(x)$ consists of an initialization step, a traversal step, and a termination step. The initialization and traversal steps are as follows.

```
p.next := head; lock(p); q := p.next;
while q ≠ head and then (q.info ≠ x or else
  q.status ≠ 0) loop
   lock(q);
   if q.status = 0 then
      q.prev := p; unlock(p); p := q;
   else
      q.next.prev := p; p.next := q.next; unlock(q);
   end if;
   q := p.next;
end loop;
```

The overhead inside the loop is the evaluation of $q.status = 0$ in the **if** statement and the recovery action $q.prev := p$ in the **then** portion. This action handles partially completed *insert* actions. The recovery action in the **else** portion handles partially completed *remove* actions. It is not an overhead since it is only performed when the strong invariant is false.

The termination step is as follows.

```
if q = head then unlock(p);
else lock(q); q.status := 1; q.next.prev := p;
   p.next := q.next; unlock(p); unlock(q);
end if;
```

The **then** portion corresponds to the case where x is not in the list. In the **else** portion, the moment

q.status is set to 1, *q* is effectively removed from the list. The next two actions disconnect the node from the list. Thus, there are three critical actions here, each of which consists of an atomic "write" operation, and failure is allowed at any point.

This example illustrates the integration of the recovery action within the normal action of fault-free processors. This can be dangerous since it increases the possibility of subtle coding errors; it is safer to simply restart the operation after performing the recovery action [22]. Other strategies are also possible.

1. The recovery is done during the lock recovery process. This will yield the most efficient code during failure-free situations. However, the lock recovery process becomes application-specific since a detailed knowledge is required about how the locks are used.
2. The recovery is done by a background process.
3. A complete recovery action is initiated whenever a violation of the strong invariant is detected.

4.3 Binary search tree

Each node in the binary search tree has the usual *left* and *right* pointers (Figure 5(a)). However, the way keys are stored in the nodes is different. The algorithm for removing a key from a binary search tree sometimes requires a node other than the node containing the key to be deleted (e.g., see [7]). This occurs whenever the target node has non-empty left and right subtrees. In this case, the information from an appropriate descendant node is moved to the target node and then the descendant is deleted. To ensure that the transfer is done in one *write* instruction without using the **swap** instruction, the key is moved out of the node into a record containing two fields, *info* which stores the key (and other information) and *nodeptr* which points to the node where the key should reside. In turn, each node contains a field, *infoptr*, which points to the information record associated with it. A node, *p*, is considered to be deleted if $p.infoptr.nodeptr \neq p$; this essentially corresponds to the status bit used in the double-linked data structure.

The strong invariant is true when the nodes form a binary search tree and when no node is in a deleted state. More formally, the strong invariant is *strong_ok(head.root)* and is given in the following.

strong_ok(p) $\equiv (p = \textbf{nil}) \vee$
($p \neq \textbf{nil} \wedge p.infoptr \neq \textbf{nil} \wedge p.infoptr.nodeptr = p$
$\wedge$ *greater* $(p.infoptr.info, p.left)$
$\wedge$ *smaller* $(p.infoptr.info, p.right)$
$\wedge$ *strong_ok(p.left)* $\wedge$ *strong_ok(p.right)*)

greater(i,p) $\equiv (p = \textbf{nil}) \vee$
$[p \neq \textbf{nil} \wedge i > p.infoptr.info \wedge$
greater$(i, p.left) \wedge$ *greater*$(i, p.right)]$

smaller(i,p) is similar to *greater(i,p)* with $>$ replaced by $<$.

The weak invariant also requires that the nodes constitute a binary search tree. However, one or more nodes are allowed to be in the deleted state. The data structures satisfying the strong and weak invariants are shown in Figure 5. The weak invariant is *weak_ok(head.root)* and is given below.

weak_ok(p) $\equiv (p = \textbf{nil}) \vee$
$(p \neq \textbf{nil} \wedge p.infoptr \neq \textbf{nil} \wedge$
weak_greater $(p, p.left) \wedge$
smaller $(p.infoptr.info, p.right) \wedge$
weak_ok(p.left) $\wedge$ *weak_ok(p.right)*)

weak_greater(p,q) $\equiv (q = \textbf{nil}) \vee (q \neq \textbf{nil} \wedge$
$(\{p.infoptr.nodeptr \neq q$
$\wedge p.infoptr.info > q.infoptr.info\}$
$\vee \{p.infoptr.nodeptr = q$
$\wedge p.infoptr.info \geq q.infoptr.info\})$
$\wedge$ *weak_greater* $(p, q.left) \wedge$ *weak_greater* $(p, q.right)$)

smaller(i,p) is identical to the corresponding predicate for *strong_ok(p)*. The asymmetry in the definition of *weak_greater* allows the fault-tolerant code to correspond closely to the fault-intolerant version that removes the previous node (in inorder traversal) if the target node has non-null subtrees. The only other difference between the strong and weak invariants is that in the latter case it is not required that the information record pointed to by the node should point back to the node. The tree is empty if all the nodes, *p*, reachable from *head* satisfy

$$p = \textbf{nil} \vee (p \neq \textbf{nil} \wedge p.infoptr.nodeptr \neq p).$$

The code for *insert(x)* consist of three portions, namely, initialization, traversal, and termination parts. The initialization and traversal parts are given in the following.

```
p := head; lock(p); q := p.root;
while q ≠ nil and then (q.infoptr.info ≠ x
   or else q.infoptr.nodeptr ≠ q) loop
   if q.infoptr.nodeptr ≠ q then del(p,q);
   else lock(q); unlock(p); p:= q;
   end if;
   if x < p.infoptr.info then q := p.left;
   else q := p.right;
   end if;
end loop;
```

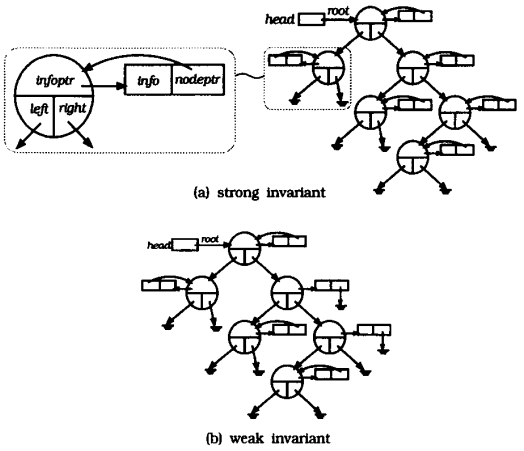

Figure 5: Binary tree data structure.

One overhead is the evaluation of the predicate $q.infoptr.nodeptr \neq q$. If it is true, then a recovery procedure, *del(p,q)*, is invoked before resuming the *insert* operation. The invocation of *del(p,q)* is not an overhead since it is only performed when the strong invariant is false. The code for *del(p,q)* is given later.

The code for the termination step of *insert(x)* is given in the following.

```
if q ≠ nil then unlock(p);
else new(r); r.left := nil; r.right := nil;
    new(r.infoptr); r.infoptr.info := x;
    r.infoptr.nodeptr := r;
    if x < p.infoptr.info then p.left := r;
    else p.right := r;
    end if;
    unlock(p);
end if;
```

The critical operations are *p.right:= r* and *p.left:= r*. Since only one of these operations needs to be performed, the strong invariant will be true irrespective of whether the processor fails or does not fail.

The code for *remove(x)* also consists of three parts, namely, the initialization, traversal, and termination parts. The first two parts are identical to those for the *insert(x)* operation. The termination part is given below.

```
if q = nil then unlock(p);
else q.infoptr.nodeptr := nil; del(p,q); unlock(p);
end if;
```

The assignment to *q.infoptr.nodeptr* essentially deletes x from the binary tree. Actual removal of the node from the tree or transfer of some other key to this node is done by *del(p,q)* which is given in the following.

```
{p is locked ∧ (p.left = q ∨ p.right = q)
 ∧ q is not locked ∧ q.infoptr.nodeptr ≠ q}
if q.left = nil then
    if p.left = q then p.left:= q.right;
    else p.right:= q.right;
    end if;
elsif q.right = nil then
    if p.left = q then p.left:= q.left;
    else p.right:= q.left;
    end if;
else lock(q); r := q.left; lock(r);
    if r.right = nil then
        q.infoptr := r.infoptr; r.infoptr.nodeptr := q;
        q.left := r.left; unlock(r);
    else s := r.right; lock(s);
        while s.right ≠ nil loop
          unlock(r); r:= s;
          s := r.right; lock(s);
        end loop;
        q.infoptr := s.infoptr; s.infoptr.nodeptr := q;
        r.right := s.left; unlock(r); unlock(s);
    end if;
    unlock(q);
end if;
```

If q has an empty subtree, then it is removed from the tree by adjusting the pointer to it from its parent, p; otherwise, the node containing the highest value from the left subtree is located and its content is moved to q. This transfer is done in one step by redirecting its *nodeptr* in the information record to q. Since this node must have an empty right subtree, it is removed from the tree by adjusting the pointer to it from its parent.

4.4 Handling non-idempotent requests

It is important to ensure that updates to the data structure are performed only once in spite of processor failures. This can be achieved efficiently using the request handling method developed in Section 3 and the fault-tolerant codes for updating various data structures presented in the above subsections.

Every task is associated with a task control block (TCB) that contains several fields. The client first creates a TCB, say *tcb*, and stores the name of the operation in *tcb.operation* and assigns values to the input parameters, *tcb.input*. It then initializes to **nil** a pointer, *tcb.nodeptr*, that points to a "detector node", i.e., a node that allows the lock recovery processor to determine whether a critical statement has been executed. For example, the detector node for *insert(x)* for the double-linked linear list is *head*, while for *remove(x)* it is the node preceding the node that is to be

removed. The client then locks *tcb.lock* and waits for a server to unlock the lock. Once this has been done, it retrieves the output from *tcb.output*. This could be error messages, such as "duplicate node" for *insert(x)*, the information in the node being removed for *remove(x)*, or some other information for other operations. It is assumed that the server can compute the output values before executing a critical statement. The server code is modified as follows.

```
read tcb.input and compute tcb.output;
tcb.nodeptr:= detecting node; -- must be locked
<execute first critical statement>
unlock(tcb.lock);
<execute the remaining critical statements>
```

The lock recovery processor for a dead processor checks the TCB, if any, that the dead processor was working on. Let this be *tcb*. It first examines *tcb.nodeptr* to determine whether a critical statement has been executed; the code for doing this depends on *tcb.operation*. If a critical statement has not been executed, then the operation can be restarted; otherwise, the remaining critical statements will be executed by other servers in the process of restoring the strong invariant.

```
if tcb.nodeptr ≠ nil then
    if tcb.lock has not been unlocked then
      -- tcb.nodeptr must already be locked;
      examine tcb.nodeptr using the detection
        code corresponding to tcb.operation;
      if it shows execution of a critical action then
        unlock(tcb.lock);
      else tcb.nodeptr:= nil;
      end if;
    end if;
end if;
```

The recovery procedure allows other servers to quickly determine whether task *tcb* has been completed (if *tcb.lock* is unlocked) or not (if it is locked). In the latter case, *tcb.nodeptr* is reset to **nil**.

5 Summary

We have developed an efficient method of tolerating processor failures for systems where multiple processors operate on a shared data structure. The approach consists of an application-independent fault-tolerant request queue handling method and application-specific data structure update algorithms. The update procedures are written to ensure that processor failures will at most result in violation of the stronger invariants; the weakest invariant should always be guaranteed to remain true. We showed how the strong invariant could be restored as part of the actions of other processors.

In contrast to self-stabilizing systems [9], we do not consider arbitrary state transitions. While self-stabilization is more general, it can suffer from a vulnerable period, following a transient failure, during which the output may not be correct. Also, in practice it is very difficult to construct efficient self-stabilizing algorithms. One effective application of self-stabilization in our approach is for maintaining optimal system configuration, such as the tournament tree shown in Figure 2. This can be done using the spanning tree algorithm given in [2]. Another combination arises from the concept of convergence stairs [10] which uses a sequence of strengthening invariants to prove that a system is self-stabilizing. However, the system output may not be correct unless the strongest invariant is established. In contrast, the multilevel data structure approach uses a strengthening sequence to achieve different performance levels while assuring correct output even when only the weak invariant is true. These two sequences can be concatenated to tolerate transient storage failures and fail-stop processor failures, with the convergence stairs leading to the establishment of the weak invariant for the multilevel data structure.

Robust data structures [8] [15] [18] [19] [21] are also used to detect and correct a certain number of arbitrary storage failures. A uniform approach for simultaneously recovering from storage and processor failures is presented in [20]. It requires an error detection and correction procedure that can either complete the operation or restore the data structure to its state before the operation. The type of correction procedures considered are those that work correctly provided that the data structure is within a certain number of changes from one of the two correct versions. In contrast, our approach can recover the data structure provided it satisfies the weak invariant even if the number of changes to reach the strong invariant is large. This makes it more widely applicable. However, the reliance on status bits makes it less robust to storage failures since changing just one bit can result in the loss of a node. An interesting question is whether the strong/weak invariant approach can be used to develop useful robust data structures that can efficiently tolerate storage and processor failures.

The fact that processor failures can be tolerated efficiently makes this approach useful in several situations. Examples include high performance transaction processing systems as well as scientific and engineering computations. Since failures can occur at any point in

the code, the approach is also attractive for real-time applications. In these cases, processors can immediately abandon non real-time or low priority tasks (i.e., "fail" for these tasks) and switch instantly to handle an urgent real-time task. The detection and recovery actions built into the code will ensure that the abandoned tasks will be completed efficiently by other processors.

6 Acknowledgments

The authors wish to thank David Taylor and the reviewers for their detailed comments and suggestions that have significantly enhanced the quality of the paper.

References

[1] S. Ahuja, N.J. Cerriero, and D. Gelernter, "Linda and friends," *Computer*, Vol. 19, No. 8, Aug. 1986.

[2] A. Arora and M.G. Gouda, "Distributed reset," *Proc. 10th Conf. Foundations of Softw. Tech. and Theo. Comp. Sc.*, LNCS 472, 1990, pp. 316-331, Springer-Verlag.

[3] W.H. Bahaa-El-Din, F.B. Bastani, and J.-E. Teng, "Performance analysis of periodic and concurrent data structure maintenance strategies for network servers," *IEEE Trans. Software Engineering*, Vol. 15, No. 12, Dec. 1989, pp. 1526-1536.

[4] F.B. Bastani, I.L. Yen, and I.R. Chen, "A class of inherently fault-tolerant distributed programs," *IEEE Trans. Softw. Eng.*, Oct. 1988, pp. 1432-1442.

[5] N.S. Bowen and D.K. Pradhan, "Processor- and memory-based checkpoint and rollback recovery," *Computer*, Vo. 26, No. 2, Feb. 1993, pp. 22-31.

[6] K.H. Cheng, "A simultaneous access queue," *Journal of Parallel and Distributed Computing*, Vol. 9, 1990.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[8] I.J. Davis, "Local correction of helix(k) lists," *IEEE Trans. Computers*, , May 1989, pp. 718-724.

[9] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Comm. ACM*, Vol. 17, No. 11, Nov. 1974, pp. 643-644.

[10] M.G. Gouda and N. Multari, "Stabilizing communication protocols," *IEEE Trans. Computers*, Vol. 40, No. 4, April 1991, pp. 448-458.

[11] K.H. Huang and J.A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, vol. 33, No. 6, June 1984, pp. 518-528.

[12] C.B. Jones, "Tentative steps towards a development for interfering programs," *ACM Trans. Prog. Langs. and Sys.*, Vol. 5, No. 4, 1983, pp. 596-619.

[13] S. Katz and K. Perry, "Self-stabilizing extensions for message-passing systems," *Proc. 9th ACM Symp. Principles of Distributed Computing*, 1990.

[14] B.W. Lampson, "Atomic transactions," in B.W. Lampson, M. Paul, and H.J. Siegert (Eds.), *Distributed Systems - Architecture and Implementation*, Springer-Verlag, New York, 1981, pp. 246-265.

[15] C.-C.J. Li, P.P. Chen, and W.K. Fuchs, "Local concurrent error detection and correction in data structures using virtual backpointers," *IEEE Trans. on Computers*, Nov. 1989, pp. 1481-1492.

[16] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

[17] M. Schneider, "Self-stabilization," *ACM Computing Surveys*, Vol. 25, No. 1, March 1993, pp. 45-67.

[18] D.J. Taylor, D.E. Morgan, and J.P. Black, "Redundancy in data structures: Improving software fault tolerance," *IEEE Trans. Software Engineering*, Vol. SE-6, No. 6, Nov. 1980, pp. 585-594.

[19] D.J. Taylor, D.E. Morgan, and J.P. Black, "Redundancy in data structures: Some theoretical results," *IEEE Trans. Software Engineering*, Vol. SE-6, No. 6, Nov. 1980, p. 595-602.

[20] D.J. Taylor and C.-J.H. Seger, "Robust storage structures for crash recovery," *IEEE Trans. Computers*, Vol. C-35, No. 4, April 1986, pp. 288-295.

[21] D.J. Taylor and J.P. Black, "A locally correctable B-tree implementation," *Computer J.*, Vol. 29, June 1986, pp. 269-276.

[22] D.J. Taylor, Private communication, 1994.

[23] R.J. Vetter and D.H.C. Du, "Distributed computing with high-speed optical networks," *Computer*, Vol. 26, No. 2, Feb. 1993, pp. 8-18.

[24] B. Vinnakota and N.K. Jha, "Synthesis of algorithm-based fault-tolerant systems from dependence graphs," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 8, Aug. 1993, pp. 864-874.

[25] I.-L. Yen and F.B. Bastani, "Robust coordination in distributed multi-server systems," *Proc. IEEE Workshop on Advances in Parallel and Distributed Systems*, Princeton, NJ, Oct. 1993, pp. 133-138.

[26] I.-L. Yen, E.L. Leiss, and F.B. Bastani, "Exploiting redundancy to speed up parallel systems," *IEEE Parallel & Distributed Technology: Systems & Applications* Vol. 1, No. 3, Aug. 1993, pp. 51-60.