Beyond Fail-Stop: Wait-Free Serializability and Resiliency in the Presence of Slow-Down Failures*

Dennis Shasha
John Turek

May 25, 1994

Abstract

Historically, database researchers have dealt with two kinds of process failures: fail-stop failures and malicious failures. Under the fail-stop assumption, processes fail by halting. Such failures are easily detectable. Under the malicious (or Byzantine) failure assumption, processes fail by behaving unpredictably, perhaps as adversaries. Such failures are not necessarily detectable. When system designers discuss fault tolerance, they typically restrict themselves to the problem of handling fail-stop failures only. This paper proposes an intermediate failure model and presents a practical algorithm for handling transactions under this model. The new failure model allows processes to fail by either slowing down or stopping; slow processes may later speed up, continue to proceed slowly, or, (eventually) stop. We call such failures *slow-down* failures. The model does not assume the ability to distinguish among these possibilities, say, by using a timeout mechanism, nor does it assume that it is possible to kill a slow process. Our algorithm, instead, allows for a new process to be dispatched to do the job that had been assigned to a slow process. The problem is that several processes may end up doing the same task and interfere with one another. Our algorithm controls such interference while guaranteeing both serializability and resiliency.

1 Introduction

The primary tool that is used in building database management systems is the transaction. Typical systems try to guarantee that any concurrent execution of transactions will have the same effect as a sequential execution of those transactions that commit during the execution; aborted transactions will have no effect.

Whether these guarantees can be achieved depends very much on the failure assumptions. Realizable systems, to date, assume *fail-stop* failures where processors fail by halting. Some memory may disappear, but other memory called *stable storage* will survive failures.

For some systems, these assumptions model hardware failures quite well. The stable storage assumption can be guaranteed by using disk mirroring (or disk arrays) and checksums. As for processor failures, many systems vendors guarantee that any single hardware failure in a processor

*This Research was partially supported by the National Science Foundation under grant IRI-89-01699 and by the Office of Naval Research under grant N00014-90-J-1110.

will cause the processor to stop and announce failure within a few nanoseconds. Moreover, when a failed processor is repaired, it performs special recovery activity – intuitively, it "knows" that it has failed.

When processors don't announce failures timeouts can be used to detect failures. The problem is that using timeouts in this way assumes that inactivity over some time interval implies a fail-stop failure. Even when this assumption is justified (e.g., when all failures are fail-stop and processors execute at predictable speeds) timeouts have the undesirable property that the timeout threshold may be too long for important real-time applications. We believe that network congestion and certain kinds of software interactions (e.g. cycle stealing activity by a high priority job) can be better modeled by processes[1] that simply slow down and may speed up later; when the processes speed up, they will not perform any special recovery activity – they do not "know" they have failed. So, we explicitly reject the assumption that the system can distinguish between a slow process, that may eventually speed up, and a stopped process, that will never execute another instruction. All we can determine is that the process is currently slow. We call this more general class of failures *slow-down* failures.

This paper presents an algorithm that tolerates slow-down failures by dispatching new processes to complete the work left by unfinished processes. The new processes can be created dynamically, as soon as the system detects that a problem may have occurred. Briefly, the algorithm uses two phase locking and a form of no undo/no redo recovery. Since several processes are working on the same transaction, it is possible that a sleeping process, p, wakes up after another process, p', has completed the transaction that t was to do. In this case, t must be prevented from causing updates. We accomplish this by timestamping the data items and applying techniques developed by Herlihy [Her89].

This paper assumes a reliable, stable, shared memory on which the primitive *compare&swap* is atomic. The basic idea also carries over to memories that suffer from slow-down failures; however, we do not address this extension here.

The next section briefly reviews related work. In Section 3 we describe our model and objectives. In Section 4 we present an algorithm, that correctly processes transactions in the presence of slow downs, and prove its correctness. Finally, in Section 5 we conclude the paper giving a brief description of some optimizations that are possible.

2 Related Work

The literature on achieving fault tolerance in database systems is extensive and its main methods are well described in [BHG87]. That work assumes that processor failures are fail-stop. The inspiration

[1] A process will be our term for a thread of execution. However, many processes share the same address space.

for our work, however, comes from recent advances in the theoretical parallel computation literature along with Herlihy's work on wait-free algorithms.

Kedem et al. [KPS90] studied the problem of achieving efficient emulation of PRAMs (parallel random access memories) on a PRAM with fail-stop failures. At each time step they use a strategy, similar to no redo/no undo, combined with a certification technique to guarantee resiliency. Martel et al. [Mar90] extend this result to include asynchronous PRAMs. The importance of these works is that they show that one does not need to limit oneself to being resilient to a predetermined number of faults. Both algorithms cause the system to degrade gracefully; as the number of faults increase, system performance decreases. In fact, both algorithms will continue to execute the computation at hand as long as at least one processor is still functional. Our algorithm has a similar flavor to Martel's in that both algorithms make use of faster processes to do the work of slower ones and that both require the use of a specialized hardware primitive.

Herlihy [Her88] proved that special hardware primitives are necessary to transform any sequential computation into a wait-free (or even non-blocking) concurrent[2] implementation. In a more recent paper [Her89], he gives various examples of the use of a specific primitive, the *compare&swap*, to create wait-free implementations of some well known data structures finally bringing wait-freedom into the realm of practicality. *Compare&swap*, shown in Figure 1, atomically com-

```
compare&swap(m, v, v')
  begin /* Atomic Action */
      if (m = v) then
      begin
          m = v';
          return(true)
      end
      else return(false);
  end; /* Atomic Action */
```

Figure 1: Compare&Swap

pares the value of a memory location m with a value v, and replaces m with another value v', if and only if m contains v. We follow Herlihy's use of the *compare&swap* in the implementation of our algorithm.

The idea of replicating transactions in order to achieve fault tolerance is not new, see for example [NS89,Bir85,DLA88,Svo84]. We focus on the more recent work by Ng and Shi [NS89]. In their algorithm they immediately create k replicas of the transaction in order to survive $k-1$ failures. Upon completion, the surviving replicas synchronize, selecting the replica with the highest

[2] Note that, in this context, concurrency implies asynchrony.

priority to commit. Their algorithm only handles fail-stop failures and generates process and data replicas regardless of need. Our algorithm can tolerate slow-down failures and replicates processes and data only when a failure occurs. However, their algorithm can tolerate both processor and memory failures while ours assumes that memory is stable.

3 Model

The computational environment that we consider is depicted in Figure 2. There are a fixed number

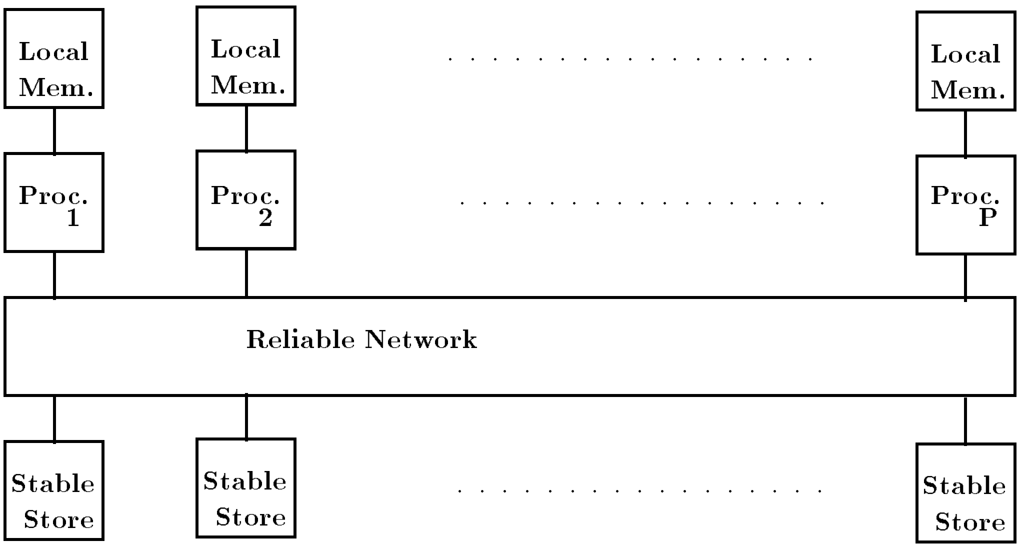


Figure 2: Computing Environment

of asynchronous processors connected to one or more reliable memory modules that we treat as a single global memory space called stable storage. The connection is through a reliable, though possibly slow, network. It is assumed that messages arrive in the same order as they were sent.

Each processor can support several threads of control; we call each such thread a process. Processes can fail either by slowing down temporarily or by stopping. We do not assume that we can distinguish between the two kinds of failures (say, by using a timeout mechanism [BHG87]), so we call either of them a *slow-down* failure. Also, we do not assume that we can kill a slow process. While the assumption that we can kill processes would significantly simplify our algorithms it is not as simple a matter as it might first appear. For example, a process may be killed after having issued several requests that cause updates. The existence of delays due to slow-down failures means that at any given time the precise state of the system is impossible to determine. Related work [And90] assumes that it is possible to kill a process and ignore any slow request that it might have sent. In that work, the assumption is justified because there is an explicit data server process that can serialize requests and knows what to ignore.

A *task* is defined by a code fragment and a task identifier, denoted task id. The code fragment refers to the execution of a transaction beginning with a *transaction start* command and ending with a *transaction commit* or a *transaction abort* command. The identifier is unique to each task and we assume it is generated by the application, before the task is accepted by the system.

An *action* is an execution of a task. Because actions can suffer from slow-down failures, several actions may be created for any given task; these actions execute concurrently. The decision of when an action should be created to perform the same task as another action may be the responsibility of the operating system, some application-dependent dispatcher, or some time-stamping convention to which the processes adhere. These issues affect the performance of the system, but will not be addressed in this paper.

A task may be aborted either because it is (or appears to be) in a deadlock state or because another task was unable to wait for this task to terminate.[3] In either case, the application receives an *aborted* terminating condition and can decide to either create a new task (to do the work of the aborted task) or to forgo execution of the task.

To summarize, a *task* is the code of a transaction. An *action* is an execution of a task. A *process* performs actions one-at-a-time on behalf of many tasks. Because of slow-down failures, several processes may perform actions concurrently on behalf of a single task. This interaction can be seen in Figure 3.

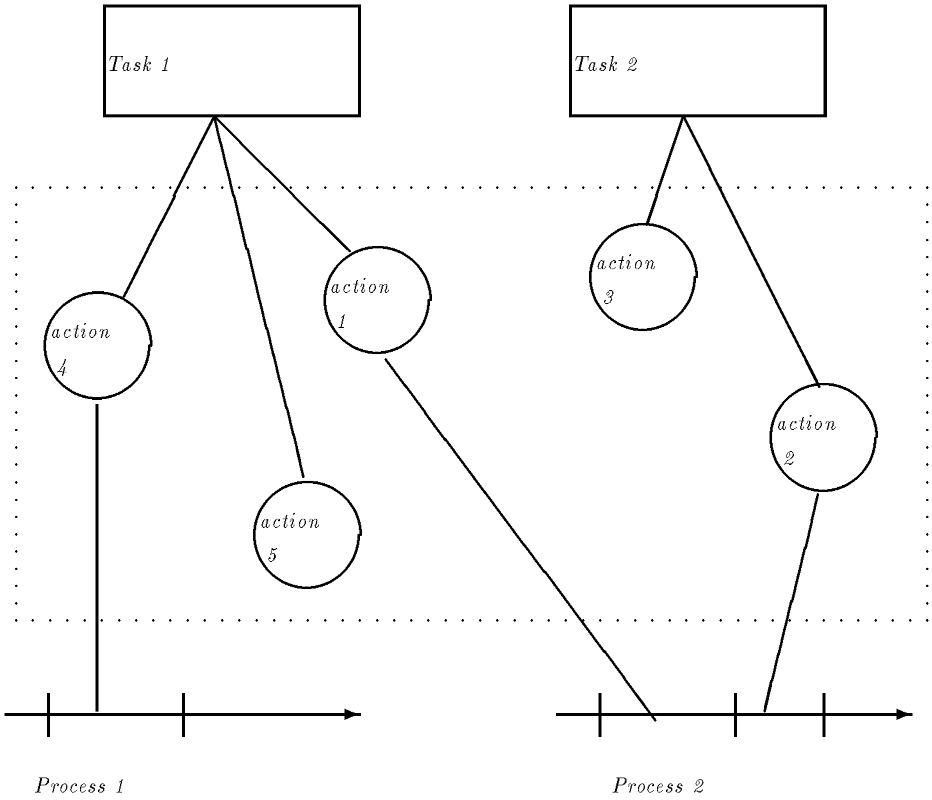
3.1 Correctness Criteria

Although the idea of having several actions do the same task is attractive in principle (reminiscent of resending messages over a network that can lose messages), it introduces several problems. For example, all the actions working on the same task must share locks but must not read the dirty pages written by one another. Also, an action a for some task t that continues to execute after another action a' has *committed* t should not update anything. If a did so, it might overwrite data that other tasks have since updated. Finally, because actions may slow down at arbitrary times, no failed action should stop the system. We summarize these considerations in our correctness criteria:

1. If a task commits, exactly one action executing that task commits.

2. The concurrent execution of tasks is correct. Following Hadzilacos [Had88], this entails the following three conditions.

[3] This is important for real-time applications where high priority jobs can not afford to be delayed by lower priority jobs.



Actions 1, 4, and 5 execute on behalf of task 1. Actions 3 and 2 execute on behalf of task 2. Process 1 executes action 4. Process 2 executes actions 1 and 2. Other processes executing actions 3 and 5 are not shown.

Figure 3: Tasks, Actions, and Processes

(a) *Commit Serializability*: Committing actions execute (one-copy) serializably.

(b) *Recoverability*: Any task (or action) which has not been committed can be aborted without affecting the semantics of committed tasks.

(c) *Resiliency*: A consistent database state can be recovered from the information in stable storage at any time.

3. The system should be wait-free. In other words, if a process gets killed, or becomes slow, it should not indefinitely block the progress of any other process.

Item 1 deserves additional attention. We do not assume that the executed task is idempotent. For example, if the effect of a task is to add 1 to x, then, if care is not taken, the net result of having two actions commit for the task will be different from having just one action commit. This problem is further exacerbated if the execution flow of the two actions is different. This can occur for several reasons; for example, the execution may depend on the state of the rest of the system (such as user input from a keyboard) or be non-deterministic.

4 Algorithm

Our algorithm uses a generalized two phase locking concurrency control algorithm combined with a generalized no undo/no redo recovery algorithm. After discussing states of task and the data state, We present the algorithm at a high level (i.e., using powerful constructs). We show how to implement those constructs later.

4.1 Task States

A task is in one of three mutually exclusive states as shown in Figure 4. Since different actions may execute on behalf of the same task, the task state is a function of the states of its various actions.

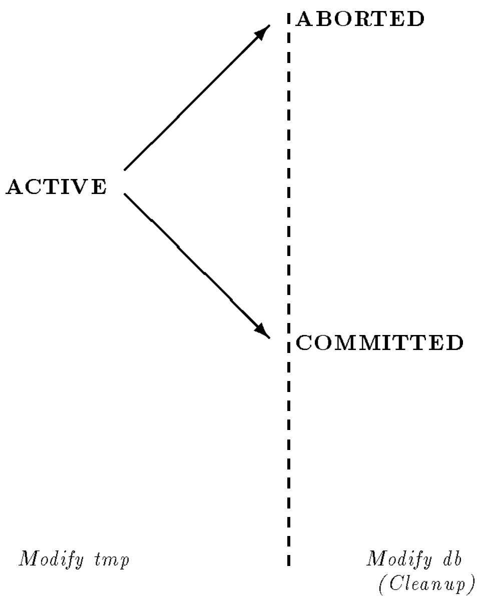


Figure 4: States of a Task

Active — Start state.

Aborted — Some action has set the task state to aborted. (The action may belong to this task or another, presumably higher priority, task.)

Committed Some action belonging to this task has set the task state to committed. (In our implementation, this means setting a pointer to a list of after-images.)

A race condition can exist between two actions where one attempts to set the state to *aborted* and the other attempts to set the state to *committed*. As we show in Section 4.6, the first action to set the state wins the race. Thus, Aborted and Committed are final states.

4.2 Data State

There may be several instances of any object in the database. Each of these either will have been written by an action of a *committed* task (we will call such an instance a *committed* instance) or will be associated with some active action. Each instance of an object has a version number associated with it. At any time, the *current version* of an object is the *committed* instance with the largest version number. Our algorithm ensures that no two committed instances of an object can have the same version number. The *data state* at any time is defined to be the set of current versions of objects in the database.

4.3 High Level Algorithm

A task starts in the active state. All actions for a task share locks with one another. When an action *aborts* a task, the action need not release locks for that task. Here is the algorithm for a single action a for task t:

1. *active phase:*

```
Loop
    Execute the code associated with the task,
    For each object x needed by the action do:
        Obtain the lock for x, abort if unable to obtain lock.
        Read the current version of ¿x into a temporary instance,
            x.tmp, that only this action can access.
        Increment the version number of x.tmp by 1.
    End for
    Continue processing.
End loop

Link all the temporary instances into an after-image list.
```

2. *commit phase:*

```
Atomically, commit task t and associate the after image list of a with task t,
    provided no other action for t has performed a commit.
    If successful, objects on the after-image list are now committed.
```

3. *Cleanup Phase* (optional; the alternative, *scavenging*, is discussed below):

Release the locks.

Atomically is taken, in this context, to mean that the atomic operation either occurs in its entirety or not at all and that no other operation can read or modify intermediate results of the atomic one. The action can abort either the task on whose behalf it executes or, depending on system requirements, the task holding the lock.

4.4 Correctness of High Level Algorithm

In subsection 4.5 we show how to implement the difficult operations of the high level algorithm. In the subsection following, we prove that the implementation is correct. This subsection *assumes* that the high level operations can be implemented and verifies the algorithm under that assumption.

Lemma 4.1 *If a task commits, exactly one action executing that task commits.*

Proof: Because of the provision associated with the commit operation, at most one action will commit for a committed task. If no action commits, then there is no way for the task to commit. ■

Lemma 4.2 Commit Serializability*: Committing actions execute (one-copy) serializably.*

Proof: Define the history of the computation on a single object to be the temporal order of reads and writes on any version of that object where we only consider committing actions. Because of write and read locks, two distinct operations on the same version of an object are ordered if one of them is a write.

Every read to an object x made by an action is to the current version of x at the time of the read. Every committing write creates a new current version. Therefore, each read will be to the value written by the last preceding write. This is the same behavior as would occur in a single copy execution. Therefore, we can treat the execution as a single-copy execution and apply the standard two phase locking theorem [BHG87]. ■

Lemma 4.3 Recoverability*: Any task (or action) which has not been committed can be aborted without affecting the semantics of committed tasks.*

Proof: The state of the database is changed only by a *commit*. Non-committing tasks or actions never reach the commit phase, so do not change the state. ■

Lemma 4.4 Resiliency*: A consistent database state can be recovered from the information in stable storage at any time.*

Proof: A consistent state is exactly the set of current versions. They are all in stable storage. ■

Lemma 4.5 *The system should be wait-free. In other words, if a process gets killed, or becomes slow, it should not indefinitely block the progress of any other process.*

Proof: If a process p is blocked by an action a' of process p', p does not depend on the progress of p' but rather of the task, say t', that p' is executing. Because many actions may execute t' in our scheme, another process p'' will eventually issue an action to execute t', unblocking p. (Which other process does this is a pragmatic decision that is outside the scope of this discussion.) ■

Since these lemmas correspond exactly to our correctness conditions, we have proven our main theorem.

Theorem 4.1 *The high level algorithm is correct assuming that each of its operations is implemented correctly.*

4.5 Implementing the Constructs of the Algorithm

In order to implement our algorithm, we need to support the fundamental operations:

1. Obtain a lock on a data item x.
2. Read the current version of a data item x.
3. Commit or abort a task by at most one action.
4. Atomically change either a task or lock state.

We discuss the above operations in order of presentation. For concreteness, we focus on a specific implementation though other implementations are possible.

Locks are held by tasks; all the actions executing a given task will share the lock on an object. Suppose that an action, a, for a task, t, attempts to acquire a lock on an object x. There are three possibilities: no task holds the lock on x, t already holds the lock (because another action for t previously acquired the lock), or the lock is held by some other task, t'. If the first case is true, then a can atomically lock x for t. If t already holds the lock, then it is as if a had already obtained the lock. In both cases, we say that the lock was obtained *directly*.

If the lock is held by another task t', then there are three possibilities. The task, t', has either *committed*, *aborted*, or is still *active*. If t' is still *active*, then it can still make progress (even if all the actions currently performing t' have failed, a new action will eventually be dispatched). In this case, a can wait or abort t or abort t' (what it does depends on implementation decisions and whether t' is deadlocked). If, on the other hand, t' has terminated (either by *committing* or *aborting*), then, as far as concurrency control is concerned, $t^{'}$ no longer needs to hold the lock. In this case a will reset the lock to its unlocked state. We call this *scavenging.*

Scavenging must be done carefully to ensure that operation 2 is supported. Locks serve two purposes in our algorithm; first, they ensure that tasks obey two phase locking; second, they serve as a pointer to the current version of an object. Ideally, the current version of an object x would always be in a well-known location say $x.db$. The difficulty is that the commit operation does not put it there but rather puts it on an after-image list.[4] So the current version must be copied from the after-image list of $t^{'}$ to $x.db$.

Normally, the first committing action of t' will perform this copying, however it may suffer a slow-down failure after committing. So we allow actions for other tasks, here action a from task t, to do the copying. Whichever action performs the copying does essentially the same sequence of steps. In both cases we call the process scavenging:

1. Atomically, write the instance of x on the after-image list of $t^{'}$ to $x.db$ provided the instance on the after-image list has a larger version number than $x.db$. [5]

2. Release the lock held by t'. That is, atomically reset the lock to null provided it is still held by t'.

3. Mark the instance on the after-image list as no longer needed (a service for the garbage collector).

At this point, unless a task other than the one on whose behalf action a executes does so first, a will be able to obtain the lock on x directly, so it tries again.

The procedure for obtaining the lock can be seen in succinct form in Figure 5. x represents the object to be locked and t the task on whose behalf the calling action, a, is acting. An instance of x on the after-image list of some task, $t^{'}$, is denoted $x.after(t^{'})$. The lock on x is denoted as $x.lock$. Once *obtain_lock* completes, the current version of x is in $x.db$. *Scavenge* can be seen in Figure 6.

As will be proved, the *obtain_lock* procedure maintains the invariant that the current version of x can either be found in $x.db$ or, if x is locked by a committed task t, in the after-image list of t. In order to establish the invariant we assume that when object x is first accessed, its current version is in $x.db$.

The *obtain_lock* and *scavenge* procedures require the implementation of 3 distinct atomic operations; this brings us to item 4 on our list of constructs to implement. We address this issue in the next subsection.

[4] For recovery afficionados, the high level algorithm is no undo/no redo, but the implementation is redo/no undo.

[5] Another possibility is to write this value to a database cache instance. In the event of a volatile memory crash, the latest committed instance of each item can be obtained from the lists associated with the committed tasks.

```
obtain_lock(x, t)
   do forever
   begin
       Atomically [ if x.lock is free or owned by t, lock x ]

       If lock attempt is successful, then exit /* lock obtained directly */
       if t' is active then /* lock is owned by another task, t' */
         wait or abort /* application dependent issue */
       else /* t' has terminated */
         scavenge(x, t')
       end if
   end
```

Figure 5: Obtaining a Lock

4.6 Implementing the Atomic Actions

One of our objectives in this paper is to use only basic hardware primitives, yet achieve resiliency in an environment with slow-down failures. Herlihy's work can be used to argue that one such primitive is necessary and sufficient – *compare&swap*. In this section we show how to implement each of the three atomic actions described in Figure 5 and also show how to *commit* a task atomically.

We first show how to atomically lock an item. We assume that the lock on an object is held in stable storage in the location defined by $x.lock$. If task t' holds the lock on x, then, the value of $x.lock$ is t' or $x.lock = t'$ for short. If no task holds the lock on x, then, $x.lock = Null$. If task t wants to lock x, then it can do so if and only if $x.lock = Null$. In other words, we need to create a mechanism whereby we can atomically determine if $x.lock = Null$ and replace it with t if true. This is easily achieved using *compare&swap*. The procedure *lock* shown in Figure 7 takes as arguments the object, x, and the task id, t, and returns *true* if, after the procedure, t holds the lock on x.

Releasing a lock held by a task, t, is similar to the procedure for locking the object. The difference is that here we need to set $x.lock = Null$ if and only if $x.lock = t$. We show the procedure in Figure 8.

Consider the problem of replacing $x.db$ with $x.after(t')$ if $x.after(t')$ has a higher version number. We address this problem by assuming that the object, x, along with its version number, can be modified by a single *compare&swap*. This is not unreasonable since x could be comprised of the version number and a pointer to the actual object; we discuss the representation of objects further in Section 5. The idea is then to read $x.db$ and $x.after(t')$ into local memory and then

```
scavenge(x, t')
  begin
     if t' is committed then
        Atomically | if x.after(t') has a higher version number
                   | than x.db, then x.db := x.after(t')

        Atomically | release lock of t'; i.e., if x.lock = t' then unlock
                   | x

        mark x.after(t') as unneeded
     else if t' is Aborted then
        Atomically | release lock of t'; i.e., if x.lock = t' then unlock
                   | x

  end if
end
```

Figure 6: Scavenging a Lock

```
lock(x, t)
   begin
      compare&swap(x.lock, Null, t); /* if x.lock = Null then x.lock = t */
      if (x.lock = t) return(true)
      else return(false)
   end
```

Figure 7: Locking an Object

use *compare&swap* to replace $x.db$ if $x.after(t')$ has a higher version number. The routine *replace*, shown in Figure 9, shows one way to do this. In the figure, the procedure $get\_version(x)$ returns the tag associated with the object, x.

A word about *replace* is in order. One might imagine that two processes might try to replace $x.db$ with different versions of x both having a higher version number than $x.db$. That cannot occur as our proof of correctness will show. Also, the restriction that $x.db$ is read into local memory is required to guarantee that the version of $x.db$ being replaced is the one that was first read; i.e., a process may go to sleep sometime after extracting the version number of $x.db$ and before overwritting $x.db$.

Finally, we have to show how to change the state of the task in an atomic fashion. For the sake of concreteness, we assume that the state of a task, t, can be found in $t.status$. Now, changing

```
release_lock(x, t)
  begin
    compare&swap(x.lock, t, Null); /* if x.lock = t then x.lock = Null */
    if (x.lock = Null) return(true)
    else return(false)
  end
```

Figure 8: Releasing a Lock

```
replace(x, t)
  begin
    old_x = x.db;
    /* if version of x.db is less than version of x.after(t') then */
    if get_version(old_x) <get_version(x.after(t')) then
      /* if x.db = x.old then x.db = x.after(t') */
      compare&swap(x.db, old_x, x.after(t'));
    end
```

Figure 9: Replacing $x.db$

the state of the task can be done by executing the routine *new_state*, shown in Figure 10, which

```
new_state(t, A, B)
  begin
    compare&swap(t.status, A, B); /* if t.status = A then t.status = B */
    if (t.status = B) return(true)
    else return(false)
  end
```

Figure 10: Changing the Status of a Task

atomically changes the state of t to B if and only if t is in state A. In the case where we are changing the state from *active* to *committed*, we also need to associate the after-image list with t. In order to do this, we define special values of $t.status$ for *active* and *aborted*. All other values of $t.status$ indicate that the task has committed and that $t.status$ is a pointer to the location of the after-image list. We describe possible implementations of the after-image list in Section 5.

4.7 Proof of the Constructs

We have already shown that our algorithm satisfies the correctness criteria as long as we can implement the lower level operations. In this section we show that the lower level constructs we have given achieve their stated objectives.

Lemma 4.6 *The routines* lock, release_lock, replace, *and* new_state *all behave atomically.*

Proof: By inspection of each of the routines and the assumed atomicity of the *compare&swap*. ■

Lemma 4.7 *If task t commits, exactly one action executing on behalf of task t will change the state of t.*

Proof: By inspection of the *new_state* routine, we see that each execution on behalf of a transaction will atomically create a list. The next execution will see the pointer to that list and will fail to commit. ■

Lemma 4.8 *The algorithm maintains the invariant that the current version of x can either be found in $x.db$ or, if x is locked by some task t that has committed, in the after-image list of t.*

Proof: An instance of an object, x, in stable storage changes only when a task commits, thereby creating a new committed instance of x, or when an action scavenges x, thereby changing the value of $x.db$. We must show that both of these operations maintain the invariant. We do so inductively.

The invariant holds when the first action acquires a lock on x, because the initial version of x is in location $x.db$. Assume that the invariant holds when some action a for task t acquires a lock on x. Between that time and the time t commits or aborts, no other task's action can write x. So, the invariant still holds just before t aborts or commits. Suppose task t' is the next to acquire the lock after t aborts or commits.

If t aborts then $x.db$ will still be the current version of x. When t' invokes scavenge, the lock that t had will be released without changing $x.db$. This will preserve the invariant.

If t commits, then exactly one action executing on behalf of t will commit by lemma 4.7. By inspection of the high level algorithm, the action a that commits will have created a temporary version of x whose version number is one greater than that of $x.db$ when a was still active. Since a holds the lock through its commit point, no other action will modify $x.db$ or create any other committed version of x while a holds the lock. Therefore, immediately after a commits, the version number of x will be one greater in the after-image list of t than in $x.db$ and will be greater than the version number of any other committed instance of x. Since t holds a lock on x at that moment, the invariant holds.

When *scavenge* is invoked, it will copy the current version of x to $x.db$ and remove the lock of t on x, maintaining the invariant.

Operations on objects follow a strict order. A task first obtains a lock on x. It then either aborts, without creating a new committed version of x, or commits, creating a new committed version of x. Finally the lock on the object is scavenged before x can be accessed by another task. This alternating sequence of events guarantees that the invariant is maintained for all objects throughout the existence of the database. ■

Corollary 4.1 *If task t has not terminated (i.e., committed or aborted), then when action a for t completes execution of* obtain_lock(x, t) *successfully the current version of x can be found in $x.db$.*

Proof: By lemma 4.8 the current version of object x will either be in $x.db$ or in the after-image list of the task holding the lock on x assuming that the task has committed. *Obtain_lock* completes successfully if and only if task t has obtained the lock. However, the algorithm guarantees that once task t has obtained the lock on x it will not release the lock until t has terminated. By assumption of this corollary, t cannot have terminated and the current version of x must therefore be in $x.db$. ■

These lemmas can be used to show that we can implement the lower level constructs in our algorithm thereby satisfying the needs of our high level algorithm.

Theorem 4.2 *Using* compare&swap *we can implement the three fundamental operations required to run our high level algorithm.*

Proof: We proceed in reverse order of the stated conditions:

1. *Atomically change either a task or lock state.* Atomicity follows from Lemma 4.6.

2. *Only one action commits or aborts a task.* Follows from Lemma 4.7.

3. *Read the current version of a data item x.* By Corollary 4.1 we can find the current version of x in $x.db$ unless the task on which the action is acting has terminated. However, if the task has terminated then by Lemma 4.7 the execution of the current action will have no effect on the state of the database and can be regarded as an empty execution.

4. *Obtain a lock on item x.* If no task holds a lock on x, then *obtain_lock* will succeed getting a lock on x for task t. If the task, t', holding a lock on x has terminated then *obtain_lock* will scavenge the lock for t and return successfully. Otherwise, either there is a deadlock and the current task, t, will be aborted (i.e. we don't need the lock on x), or t' will eventually terminate and *obtain_lock* will succeed.

■

5 Efficiency Considerations

We have tried to avoid implementation details until now, because these details do not affect the correctness of our algorithm. However, they can have significant performance consequences. In this section we explore some implementation alternatives.

5.1 Object Representation

Our technique depends on being able to modify an object with a single atomic action. If the data object occupies more space than can be realistically handled by the *compare&swap* then the object can be replaced by pointers to the object. This implies that a new copy of the object must be created each time it is modified. Although this is not a problem for small objects, it can have serious performance implications for larger objects. We follow Herlihy [Her89] in using persistent data structures [DSST86] as an economical alternative to copying entire objects.

For ease of exposition, we refer to *blocks* of memory. The size of a block will be chosen as a function of latency and transmission time in the network and in stable storage. An object, x, consists of two parts, $x.version$ and $x.data$, where the combined pair can be handled by a single *compare&swap*. If the contents of the object are too large to fit into $x.data$, then $x.data$ contains a pointer to a block containing the actual data. When the object changes, the entire block is copied and $x.data$ is changed to point to the new block.

The problem is that the contents of x may be too large to fit into one block of memory. In this case, we represent the object as a tree of blocks. The leaves of the tree represent the data and the internal nodes represent pointers to blocks. Modifying an object now consists of replacing the appropriate leaf(ves) of the tree as well as the nodes on the path from the leaf to the root. See Figure 11. We refer to objects fitting in $x.data$ as *word level* objects, objects fitting in a block as *block level* objects, and larger objects as *complex* objects.

This technique also minimizes the overhead associated with maintaining multiple versions of an object. This creates concurrency opportunities that need to be explored further.

5.2 Read Locks

Using read (also known as shared) locks is a well known method of increasing the amount of concurrency available in a transaction processing system. Our algorithm, as presented so far, assumes that all locks are write locks (and therefore exclusive). In this section, we describe how to incorporate read locks into our algorithm.

Because of scavenging, we must remove read locks associated with tasks and therefore must know which task each read lock belongs to. Therefore, we will have a linked list of locks associated with each object. That linked list will have length one if a task holds a write lock for the given

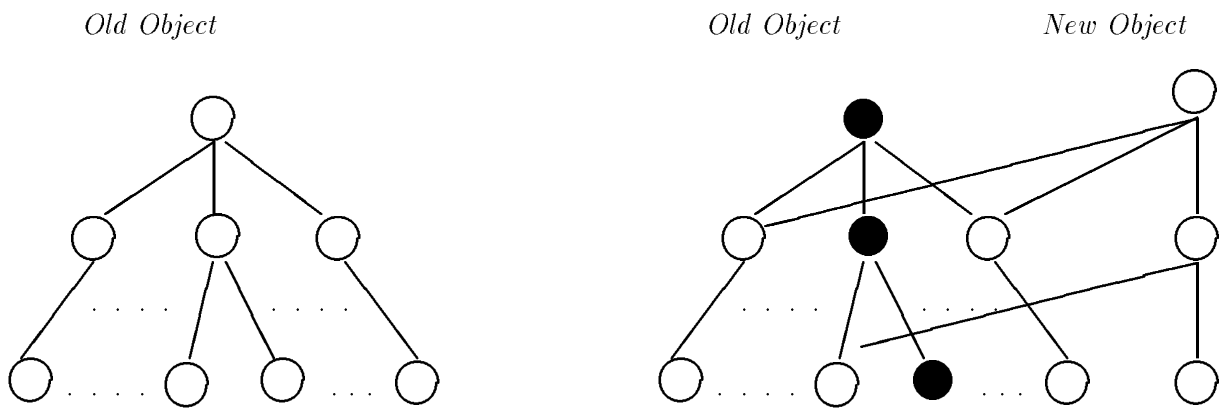


Figure 11: Modifying a Large Object

object. So, to obtain a (read or write) lock on object x, an action, a, on behalf of task t will get an exclusive lock on the linked list associated with x and then will attempt to obtain the lock.

The inherent difficulty of implementing this approach, as stated, is that we now need to obtain an exclusive lock on the linked list and release the lock before task t terminates (otherwise we have an exclusive lock on the list for the duration of t and have not gained anything). Even if we treat this lock in a special fashion (i.e., we do not require the 2-phase locking constraint to be maintained for this lock) action a could still fail after having obtained the exclusive lock on the list but before releasing it. We now need to control interaction between the actions. When we were dealing with exclusive locks only the lock could be viewed as a word level object that could be manipulated with a single *compare&swap*. With read locks, the lock has become either a page level or complex object that cannot be manipulated atomically.

One solution to this problem is to create a *subtask* to deal with the lock list. The purpose of this subtask is to perform the requested insert or delete operation on the lock list. Subtasks, like their parents, are performed by actions; if one action fails, then another will be created to perform the work of the failed action. Since at most one lock (on the lock list) must be obtained by the subtask, no deadlock can occur; therefore, subtasks never need to be *aborted* and will always terminate successfully. The parent task waits for its subtask to complete before continuing.

Since the lock list is a well-defined object (having operations insert, read, and delete), one could directly apply the techniques in [Her89] to make the lock list wait-free. This would avoid the additional overhead of creating a subtask.

5.3 Garbage Collection

Throughout the life of a database, many tasks will be processed. Our algorithm requires that space be created for each of these tasks and does not address the issue of reclaiming this space once the task has terminated. Furthermore, page level and complex objects require the creation of additional space that also needs to be reclaimed. In this section we focus on the problem of garbage collection.

Assuming that we have only word level objects, space is created by each task for the task status word and the task id. Space is created by each action for the after-image list. We assume that the object is stored directly in the after-image list and is copied, via a *compare&swap*, directly into the $x.db$ version. Hence, we only need to garbage collect the after-image list, the task status word, and the task id.

Each process is assigned a fixed amount of process space in stable memory. The process space is large enough to hold at least one full after-image list. Each process is responsible for reclaiming its own process space. If a process fails permanently, then its process space is also lost. When a process attempts to commit an after-image list and fails, it can immediately reclaim the space used by that list. When a process runs out of space (when creating a new after-image list) it can reclaim any committed after-image list, l, it created by:

1. Scavenging all the objects on l.
2. Change the task id associated with l to null and the task status to null.
3. Reclaim l.

This, then, reclaims both the after-image list, l, as well as the task space (i.e., the task space has been set to null and can be reused). We still have to deal with errant processes that are reading the reclaimed data; i.e., we need to guarantee that an action accessing l at the same time as it is reclaimed will not read garbage. We achieve this result by modifying the way in which an action accesses the after-image list. Recall that the task status word points to the after-image list:

1. Find the task status word associated with task, t. If t cannot be found in the list of task ids and their respective status words, then the space associated with that task has been reclaimed (therefore, all the locks have been scavenged) and we can stop.
2. Search for the object in the after-image list.
3. Check the task id of t. If it has changed (i.e., it no longer contains the task id of t), then the after image-list has been reclaimed and the object read is no longer valid. If it has not changed, then the object retrieved from the after image-list is valid and can be used.

Hence, an after-image list is considered valid only if the task id of of the task for which the list was committed is still valid after the list has been read. We check the validity of any object retrieved from the after-image list by checking the validity of the after-image list after the object has been retrieved.

In order to extend our algorithm to deal with page level and complex objects, we can still use the same technique as above to deal with the after-image list and the task space; however, we add a global free list that is used to manage the blocks with which page level and complex objects are built. We extend the after-image list to maintain a reference to the old object as well as the new object. In this way, when a process is reclaiming an after image list it can also reclaim the space no longer needed by the new version of the objects on the after-image list; this avoids competition during the reclamation. Processes will discover that they are errant when they attempt to read the root of a complex object after reading portions of that objects.

Reclamation must add all freed-up space to a free list structure in the global space. This can be implemented by using subtasks in the spirit of Section 5.2.

5.4 After-Image List

The after-image list plays a critical part in our algorithm. Depending on the system configuration (e.g. number of processes, contention for data objects, etc.) one may want to use a variety of different data-structures. Some examples are:

1. A linked list or a binary search tree.
2. A hash table. If the application is real-time, then the hash table can be constructed in such a way as to guarantee that high priority tasks spend minimal time scavenging any locks that they might require.
3. A wait-free linked list implemented using the techniques in [Her89]. HAs the linked list is scanned for the object in question, all objects encountered in the search are scavenged and removed from the after-image list (some modifications to the garbage collecting policy described above would be necessary). This amortizes the cost of scavenging locks across all the processes and leads to improved overall system performance.

6 Conclusion

We have presented an algorithm that guarantees the correct serializable execution of tasks in an environment where slow-down failures may occur. While we have focused on an approach using 2-phase locking, the algorithm could be modified to work with other locking strategies. The algorithm uses three techniques that may have wider applicability.

- The use of several dynamically created actions to perform a single task. This may prove useful in real-time transaction systems even when slow processes may speed up eventually.
- The dual use of locks on stable storage as pointers to after-image lists as well as to ensure serializability.
- The ability to scavenge locks from terminated (committed or aborted) tasks. This is critical to our ability to handle slow processes.

Further work is needed to extend the slow-down model to replicated failure-prone memories. For example, consider replicated file servers where timeout is not a reliable indication of a stop failure.

ACKNOWLEDGEMENTS
The authors would like to thank Brian Anderson, Zvi Kedem, Krishna Palem, and Bill Weihl for helpful discussion.

References

[And90] Anderson, B. Persistent Linda: Extending Linda to Include DataBase Facilities. Manuscript: To appear as N.Y.U. Ph.D. thesis, 1990.

[BHG87] Bernstein, P., V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[Bir85] Birman, K. P. Replication and Fault-Tolerance in the ISIS System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 79–86, 1985.

[DLA88] Dasgupta, P., R. LeBlanc, and W. F. Appelbe. The Clouds Distributed Operating System: Functional Description, Implementation details, and Related Work. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9, 1988.

[DSST86] Driscoll, J., N. Sarnak, D. Sleator, and R. Tarjan. Making Data Structures Persistent. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 109–121, 1986.

[Had88] Hadzilacos, V. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, January 1988.

[Her88] Herlihy, M. Impossibility and Universality Results for Wait-Free Synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

[Her89] Herlihy, M. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceeding of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1989.

[KPS90] Kedem, Z., K. Palem, and P. Spirakis. Efficient Robust Parallel Computations. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 134–148, May 1990.

[Mar90] Martel et. al. Efficient PRAM Emulation by Asynchronous Prams. Manuscript: Submitted to FOCS 1990, 1990.

[NS89] Ng, P. and S. Shi. Replicated Transactions. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, pages 474–480, 1989.

[Svo84] Svobodova, L. Resilient distributed computing. *IEEE Transactions on Software Engineering*, 10(3):257–267, May 1984.