How to share an object: A fast timing-based solution

Rajeev Alur Gadi Taubenfeld
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974.

Abstract

We consider the problem of transforming a given sequential implementation of a data structure into a wait-free concurrent implementation. Given the code for different operations of an object that is designed to work under the assumption that only a single process accesses it, we want to construct an implementation that works correctly in a concurrent environment where it may be accessed by many processes.

We assume a shared memory model with atomic registers. It is well known that using atomic registers it is impossible to construct concurrent implementations of even very simple objects such as test-and-set bits. However, we show that the knowledge about relative speeds of processes *can* be used for such implementations. We assume that there is a known upper bound on the time taken by the slowest process to execute a statement involving an access to the shared memory. This timing assumption is very powerful and enables us to construct fast wait-free implementations of data structures such as queues, stacks and synchronization primitives such as test-and-set, compare-and-swap, fetch-and-add, etc.

Our transformation works only when the given sequential implementation is bounded, that is, there is a known upper bound on the number of steps required to complete any of the operations it supports. In the absence of contention, it guarantees that there is only a small overhead in the cost of executing the concurrent operations over the sequential ones, namely, only a constant number of accesses to the shared memory.

1 Introduction

An *object* is a data structure that can be accessed via a fixed set of operations. A *sequential* implementation of an object consists of the code for all its operations, which behaves correctly when all the operations are executed one after the other in a sequential fashion. A *concurrent* implementation gives the code that behaves correctly even when executed by many processes concurrently. A concurrent implementation is usually required to be *wait-free*, that is, it should guarantee that any operation by a process will always be completed in a finite number of steps regardless of the behavior of other processes (such as abnormal termination). Since writing a concurrent implementation is usually much trickier than writing a sequential one, our goal is to obtain a general method that transforms a given sequential implementation into a wait-free concurrent implementation. For efficiency purposes, we are interested in minimizing the difference between the costs of executing an operation in the concurrent implementation and the sequential one. In this paper, we will focus on the *contention-free* complexity of a concurrent implementation, namely, the cost of an operation when only one process is accessing the object. Lamport has pointed out that in well designed systems contention for using an object is rare – most of the time only one process is interested in accessing the object [Lam87]. Thus, we want an implementation that works correctly even when many processes access the object, but when only one process is accessing the object the overhead in executing an operation is small. We will say that a concurrent implementation is *fast* when the difference between the cost of an operation in the sequential implementation and its contention-free cost in the concurrent implementation is small.

One way to obtain a concurrent implementation from the sequential code is to enforce sequentiality in accessing the object using a mutual exclusion algorithm. In this solution, in order to access the object, a process participates in a mutual exclusion algorithm that protects the object, and accesses the object only in its critical section. However, typically mutual exclusion algorithms are not wait-free, and a failure of a process in its critical section will block any further access to the object by other processes. Also, for this solution to be fast, the mutual-exclusion algorithm used needs to be of low contention-free complexity. We will present a solution by designing a fast wait-free algorithm for mutual exclusion.

The model

For process communication, we use the shared memory model with *atomic registers*. Thus, in one atomic step a process can either read or write a shared register, but cannot do both. It is well known that using atomic registers it is impossible to construct concurrent implementations of even very simple objects such as queues and test-and-set bits [LA87]. However, we will use timing assumptions, and show that the knowledge about the relative speeds of processes *can* be used to obtain such implementations.

We assume that there is a known upper bound, denoted by Δ, on the time taken by the slowest process to execute a single step. A single step involves at most one access to the shared memory along with a minimal local computation such as assignments to local registers, change of control location in the code, comparisons etc. Observe that our timing assumption does not imply lock-step execution as in the synchronous model, and the time it takes for different processes to execute a single step may be different. The bound Δ can be used explicitly in the following way: a process can execute a statement $delay(\Delta)$; this statement is similar to a *skip* statement, but takes at least Δ time units to finish execution.

The cost of an implementation is measured by the number of accesses to the shared memory, and the amount of explicit delay introduced by executing the *delay* statements. A concurrent implementation is called *fast* if the difference between the sequential cost and the contention-free concurrent cost is a constant number of accesses and no explicit delay. Notice that the costs of a single access to memory and an execution of $delay(\Delta)$ are different, because the value of Δ needs to account for the slowest process, whereas the time taken to execute a memory access needs not. Hence, we require that no delay statements are used in absence of contention.

Processes are subject to *crash failures* in which a process at an arbitrary time ceases to participate further in the algorithm. As long as a process has not failed, it must follow its program and satisfy all the timing assumptions. Thus, wait freedom means that any operation by a process must be completed in a finite number of steps even if all other processes crash.

Overview

The main result of the paper is a method for transforming a sequential implementation into a fast wait-free concurrent one. Our transformation works only for *bounded implementations.* A sequential implementation is called *bounded* if there exists a known upper bound on the number of steps required to complete any of the operations. Thus, in a bounded implementation, the time complexity of various operations does not depend on the input or the current state of the object. The natural implementations of queues, stacks, test-and-set, fetch-and-add, consensus (for one process), sticky-bit, swap, compare-and-swap, are all bounded. On the other hand, consider a data structure *set* that is implemented as a linked list with the membership operation implemented by a linear search through the list. Such an implementation is not bounded because the number of steps needed to execute the membership operation depends on the numbers of elements in the set.

We reach our goal in several steps. First, using only atomic registers we design a fast timing-based implementation of a restricted type of test-and-set bits; the subsequent algorithms employ these bits as primitives. In the second step, we construct a fast starvation-free mutual exclusion algorithm using test-and-set bits. This algorithm is interesting in its own right. It makes no timing assumptions and regardless of the level of contention, the maximum number of steps of the process that enters its critical section in its entry code and exit code, since the *last* time a process exited its critical section, is a constant (assuming *test-and-set* takes one step). The previous best known starvation-free algorithm has time complexity $O(\log n)$, where n is the total number of processes. The third step is to modify the starvation-free algorithm, maintaining the property of being fast, so that it also becomes wait-free. This requires the additional assumption that there is a bound on the time a process spends in its critical section.

Finally we use the fast wait-free mutual exclusion algorithm to transform a given sequential bounded implementation into a concurrent one. When each sequential operation involves only writes that are robust to failure, that is, a process failure immediately after a write does not leave the object in an inconsistent state, the transformation is easy. In this case the wait-free mutual exclusion algorithm is fine-tuned so that the bound on the time a process is allowed to be in its critical section is long enough to enable it to finish an operation. When the sequential operations contain writes that are not robust to failure, a process may fail leaving the object in an inconsistent state. In this case, when some other process detects such a failure it needs to complete the preceding unfinished operation. In Section 5 we explain how this can be done for any bounded implementation.

Related work

The idea of transforming a given sequential implementation into a wait-free concurrent one, has drawn the attention of many researchers. Both general transformation methods and specific concurrent implementations of various objects have been proposed [Her91, Plo89, Her90]. However, our model, that employs atomic registers with timing assumptions as primitives, is quite different from the models used in these previous works.

The importance of contention-free complexity was stressed originally by Lamport [Lam87], where he presents a *fast* mutual exclusion algorithm in which the number of steps by a process before entering its critical section in the absence of contention is constant. Since then, various algorithms that are sensitive to the level of contention have been proposed for mutual exclusion [AT92, CS93, MT93].

Relatively less work has been done on designing algorithms in the timing-based model used in this paper. In such a model, the first deadlock-free mutual exclusion algorithm, is due to Fischer [Lam87]. Lamport also gives a fast timing-based deadlock-free mutual exclusion algorithm, which works correctly when some bound is assumed on the time needed to execute the critical section [Lam87]. In [AT92], we have presented a fast timing-based algorithm for mutual exclusion. Lynch and Shavit have presented an algorithm for mutual exclusion where only the property of deadlock freedom depends on the timing assumptions; their algorithm is not fast [LS92]. None of these previous algorithms is starvation-free, and hence, cannot be used directly to obtain wait-free concurrent implementations.

2 Implementing test-and-set bits

In this section we provide a fast implementation of a restricted type of test-and-set bits from atomic registers using timing assumptions. This implementation will play a central role in the implementations of more complex objects in the remaining sections.

2.1 Single-use test-and-set

We first consider test-and-set bits that are meant to be used only once. The shared object is a bit that is initially *false*, and it is accessible to the processes sharing it through the *test-and-set* operation. This operation atomically reads the register, sets it to *true*, and returns the value read. Such an object is called a *single-use test-and-set bit*. When many processes execute the test-and-set operation on such an object, the first operation returns *false* and all others return *true*, and hence, we need some mechanism of choosing the winner.

```
function single-use-test-and-set($t$):
$t.x$, $t.y$ : shared registers, initially $t.y$ is 0;
$t.z$ : shared bit, initially false;
        $t.x := i$;
        if $t.y \neq 0$ then return(true) fi;
        $t.y := i$;
        if $t.x \neq i$ then  delay($3\cdot\Delta$);
                              if $t.y \neq i$ then
                                 return(true) fi fi;
cs:     if $t.z$ then return(true)
        else $t.z :=$ true; return(false) fi
end-function
```

Figure 1: Fast timing-based implementation of a single-use test-and-set bit from atomic registers (process i's program).

Our solution is inspired by the algorithms for mutual exclusion of [Lam87] and [AT92], and is shown in Figure 1. A single-use test-and-set bit t is implemented using two atomic registers x and y, and an atomic bit z. We assume that each process has a unique identifier, and when process i wants to execute the test-and-set operation on the bit t, it executes the code of Figure 1. In our construction, the bit z models the actual shared bit, and is accessed only in the critical section of the code, namely, the *if* statement labeled *cs*. The algorithm ensures that no two processes are in their critical sections simultaneously. Also, if some process i returns *true* without reading z, then some other process j, that starts before i finishes, enters its critical section (assuming no failure). These two properties together ensure the existence of a consistent serialization. Let us see why no two processes can be in their critical sections simultaneously. We will say that a process i enters its critical section along path α if it finds $x = i$, and along path β if it finds $y = i$ after the delay. It should be obvious that at most one process can enter along path α. The delay statement has two roles. First, after some process executes the delay, the value of y will stay unchanged, and hence at most one process i can find $y = i$ after the delay, and enter along path β. Secondly, suppose that process i enters along path α and process j enters along path β. Since j finds $y = j$ after the delay, process i must have executed $y := i$ before j starts its delay. Since process i takes at least 3 steps while j executes its delay statement, i terminates when j finishes the delay.

In absence of contention, a process enters the critical section along path α, executes only 7 steps. In presence of contention, the test-and-set operation involves at most 8 steps and an explicit delay of $3{\cdot}\Delta$.

2.2 Multi-use test-and-set

The (multi-use) test-and-set bit is similar to a single-use test-and-set bit, except that it has an additional *reset* operation. As before, the bit is initially *false*. The test-and-set operation is as before, and the reset operation assigns the value *false*.

We will modify the code of Figure 1 to implement a weaker form of this object. Let us define a *corrupted test-and-set bit* to be a bit with test-and-set and reset operations such that the test-and-set operation always returns *true*, and the reset operation does nothing. Thus, a corrupted test-and-set bit is just like the (read-only) constant value *true*. A *corruptible test-and-set bit* is again a shared bit accessible through test-and-set and reset operations; it behaves like a correct test-and-set bit as long as there are no process failures, but if a process accessing the bit crashes then the bit *may* start behaving like a corrupted test-and-set bit. In other words, a failure of a process accessing the bit may cause all the subsequent test-and-set operations to return *true*.

Figure 2 shows the proposed construction of a corruptible test-and-set bit from atomic registers and delays. The construction is similar to the one of Figure 1, and we will point out the differences. As before, the bit z models the actual shared bit, and the reset operation corresponds simply to setting the bit z to *false*. In Figure 1, once the register y gets a nonzero value, it is never reset to 0, and all the remaining processes return *true* without even testing z. This mechanism is now unacceptable because of the reset operation. In the new code, the process that enters the critical section resets y to 0 before returning from the test-and-set operation. The algorithm ensures that no two processes are in the critical section simultaneously.

```
function corruptible-test-and-set(t):
t.x, t.y : shared registers, initially t.y is 0;
t.z : shared bit, initially false;
     t.x := i;
     if t.y ≠ 0 then delay(Δ);
                     if t.y ≠ 0 then
                        delay(9·Δ);
                        return(true) fi fi;
     t.y := i;
     if t.x ≠ i then delay(4·Δ);
                     if t.y ≠ i then
                        delay(5·Δ);
                        return(true) fi fi;
cs:  if t.z then t.y := 0; return(true)
     else t.z := true; t.y := 0; return(false) fi
end-function
procedure reset(t)
     t.z := false
end-procedure
```

Figure 2: Fast timing-based implementation of a corruptible test-and-set bit from atomic registers (process i's program).

To prove correctness, we need to show that every run involving multiple invocations of *test-and-set* and *reset* operations has a consistent serialization. With each operation we associate a *commit* point. Consider a *test-and-set* operation by a process that enters the critical section. If the process finds z set, then the commit point coincides with this read. If the process finds z to be *false* and assigns *true* to z, then the commit point coincides with this assignment. The commit point of a *reset* operations coincides with the assignment of *false* to z. To begin with, let us ignore all the *test-and-set* operations that terminate without testing z, that is, without entering the critical section. Then, it is easy to prove that the ordering of all other operations specified by the ordering of their respective commit points gives a consistent serialization. Now consider a *test-and-set* by some process i which returns *true* without testing z. The deadlock freedom of the underlying mutual exclusion algorithm ensures that there is some other process j whose execution of *test-and-set* overlaps with that of process i, and process j enters the critical section. The long delays introduced just before process i terminates ensure that this invocation overlaps with the commit point associated with process j's operation. The commit point for process i's operation is immediately after process j's commit point, and this leads to a consistent serialization. There is one remaining case: it may happen that process j is already committed before process i starts, but process j has not yet reset y to 0. In this case, process i may find $y \neq 0$. The statement $\mathit{delay}(\Delta)$ following the test ensures that process j finishes the assignment $y := 0$ in its critical section. Now process i tests y again, and if it finds $y \neq 0$ again, then there must be another process k that enters the critical section. In this case, process i's execution overlaps with the commit point of process k.

If a process in the critical section fails just before resetting y, then all the subsequent test-and-set operations will return *true*, and thus, this failure will change the bit into a corrupted test-and-set bit.

In absence of contention, a process will enter its critical section along path α and test z, and thus, the implementation is fast. The worst-case time-complexity of the implementation can be computed easily by counting the steps. The properties of the construction are as follows:

Proposition 1 *Properties of algorithm of Figure 2:*

- *The algorithm is a correct implementation of a corruptible test-and-set bit.*
- *A process finishes test-and-set operation within time* $17{\cdot}\Delta$ *irrespective of the failures of other processes.*
- *The contention-free time complexity is 8 steps and no delay statements.*

3 Starvation-free mutual exclusion

We now present a new and simple solution to the mutual exclusion problem [Dij65] (see also [Ray86]). The solution uses test-and-set bits as the basic synchronization primitive. The algorithm we present is fast even when there is contention – its worst-case time complexity is a constant. That is, regardless of the level of contention, the maximum number of steps of the process that enters its critical section in its entry code and exit code, since the last time some process exited its critical section, is a constant (assuming *test-and-set* takes constant time). This should be contrasted with the fact that using atomic registers only, it is impossible to design a deadlock-free mutual exclusion algorithm, even for two processes, with bounded (worst-case) time complexity [AT92].

```
turn: shared register;
lock: shared (corruptible) test-and-set bit,
      initially false;
waiting[0..(n − 1)]: shared array of bits,
      initially false;
lturn, key: local registers ;

   waiting[i] := true;
   key := true;
   while (waiting[i] and key) do
         key := test-and-set(lock) od;

   critical section;

   waiting[i] := false;
   if turn = i then
         lturn := (turn + 1) mod n
   else lturn := turn fi;
   if waiting[lturn] then
         turn := lturn; waiting[lturn] := false
   else turn := (lturn + 1) mod n; reset(lock) fi
```

Figure 3: Fast starvation-free mutual exclusion using test-and-set bits – process i's program.

The algorithm is based on a starvation-free algorithm by Burns [Bur78]. In his algorithm, the winning process, even in the absence of contention, executes $O(n)$ steps, where n is the total number of processes. As far as we know, the time complexity of the best previously-known (deterministic) starvation-free solution is $O(\log n)$.

The algorithm of Figure 3 employs a test-and-set bit *lock*, an array *waiting* of atomic bits, and a shared atomic register *turn*. Process i, when it wants to enter its critical section, first sets $\mathit{waiting}[i]$ to *true*, and then repeatedly performs *test-and-set* operation on the bit *lock*. It can decide to enter its critical section in two ways. If *test-and-set* returns *false*, then the process has the lock, and can enter the critical section. On the other hand, the last winner, that is, the last process to enter the critical section, may grant permission to process i by resetting $\mathit{waiting}[i]$ to *false*. When some process exits its critical section, it checks if the process *turn* wants to enter critical section, that is, if $\mathit{waiting}[\mathit{turn}]$ is set. If so, it grants *turn* permission to enter the critical section; otherwise, it increments *turn* and resets *lock*. The winning process executes only a constant number of steps in both its entry code and exit code. Also, the algorithm guarantees that while a process is waiting to enter its critical section, all the remaining processes can enter the critical section at most $n-1$ times altogether. The properties of the algorithm are summarized in the following claim:

Proposition 2 *Properties of algorithm of Figure 3:*

- *No two processes are in their critical sections simultaneously even if processes fail.*
- *In absence of process failures, if a process is trying to enter its critical section, then it eventually enters its critical section.*
- *Assuming test-and-set takes constant number of steps, a process entering its critical section takes constant number of steps in its entry code and exit code, since the last time some process exited its critical section.*

Note that the algorithm makes no use of timing assumptions. We can replace the test-and-set bit by atomic registers with timing assumptions according to the construction of Section 2. The resulting algorithm is a timing-based starvation-free solution from atomic registers. In absence of contention, it provides fast access with constant time complexity.

```
$turn$: shared atomic register;
$count[0..(n-1)]$: shared array of registers;
$lock[0..(n-1)]$: shared array of corruptible
        test-and-set bits, initially $false$;
$waiting[0..(n-1), 0..(n-1)]$: shared array of bits,
        initially $false$;
$lturn, lcount, j, \ell$: local registers, $\ell$ is initially 0;
$K$: constant;

start1: $waiting[i,\ell] := true$;
start2: $lcount := count[\ell]$;
        for $j = 1$ to $K$ do
            if not $test\text{-}and\text{-}set(lock[\ell])$ then
              goto cs fi;
            if not $waiting[i,\ell]$ then
              goto cs fi;
            $delay(\Delta)$ od;
        if $count[\ell] = lcount$ then
            $\ell := \ell + 1$; goto start1
        else goto start2 fi;
cs:     critical section;
        $lcount := (count[\ell] + 1) \bmod n$;
        $count[\ell] := lcount$;
        $waiting[i,\ell] := false$;
        if $turn = i$ then
            $lturn := (turn + 1) \bmod n$
        else $lturn := turn$ fi;
        if $waiting[lturn,\ell]$ then
            $turn := lturn$; $waiting[lturn] := false$
        else $turn := (lturn + 1) \bmod n$;
            $reset(lock[\ell])$ fi
```

Figure 4: Fast wait-free mutual exclusion using test-and-set bits – process i's program.

4 Wait-free mutual exclusion

Now we modify the algorithm of Section 3 so as to make it robust to process failures also. The property of *wait freedom* requires that a failure of a process should not block the progress of other non-faulty processes; that is, if a process is trying to enter its critical section, then it should eventually enter its critical section, provided this process itself does not fail. To ensure wait freedom, it is essential that a process is able to detect the failure of some other process. We use timing assumptions for this purpose. We assume that there is an upper bound on the amount of time a process is allowed to spend in its critical section, and this bound is known to all the processes. Recall that in our model only crash failures are allowed, and thus, all timing assumptions are satisfied as long as a process participates in the algorithm.

Let us see how the solution of Figure 3 can be made wait-free. We introduce a new register called *count*, and this register is incremented each time a process leaves its critical section. Recall that, in absence of failures, once a process leaves its critical section, the next one enters the critical section within a constant number of steps. Each process spends only a bounded amount of time in the critical section. This implies that the value of *count* should change within a bounded period. If a process records the value of *count* initially, and finds *count* unchanged after a sufficient delay, it can conclude that some process has failed. Observe that, once a process i starts trying, there can be at most $n-1$ entries to the critical section before i wins, and hence, *count* can get incremented at most $n-1$ times while i is waiting to enter. Consequently, it suffices to have *count* to act as a modulo n counter. Once a failure is detected by a process, the process can execute the same algorithm, but with a fresh set of registers. For each process failure, we may need a new copy, and hence, n such copies are required. (A finer analysis reveals that it is sufficient to have only one copy of the register *turn*.)

The solution is shown in Figure 4. The register ℓ denotes the number of the copy currently being used. Initially, ℓ is 1, and when a failure is detected, it is incremented. Notice that this register should stay the same between different invocations of the mutual exclusion algorithm by the same process; that is, though ℓ is accessible only by process i, it is a (temporally) global register.

The waiting for entering the critical section is mainly inside the **for** loop. As before, process i can enter its critical section if it obtains *false* from *test-and-set* operation, or if the previous winner grants it permission by resetting its bit in the array *waiting*. Executing the loop K times ensures a delay of $K \cdot \Delta$ between the successive sampling of the register *count*. Let $K_{cs} \cdot \Delta$ be the upper bound on the time that a process is allowed to spend in its critical section. Suppose the time taken to execute *test-and-set* is $K_{ts} \cdot \Delta$. A careful counting shows that it suffices to have $K = K_{cs} + K_{ts} + 12$. If we assume that *test-and-set* is atomic, then choose K to be $K_{cs} + 13$.

If a process accessing *lock* fails, thereby corrupting it, then this failure will be eventually detected by everyone. We point out that a process will have to execute the **for** loop K times only when some process fails, a rare occasion. On the other hand, in absence of contention, our wait-free solution is fast.

Proposition 3 *Properties of algorithm of Figure 4:*

- *No two processes are in their critical sections simultaneously even if processes fail.*
- *If a process is trying to enter its critical section, then it eventually enters its critical section even if other processes fail.*
- *In absence of contention, a process needs to execute 4 steps (including a test-and-set operation) in its entry code, 8 steps in its exit code, and no delay statements.*

Since *lock* can be an array of corruptible test-and-set bits, we can use the implementation of Section 2. In that case, the value of K should be $K_{cs} + 29$, and contention-free time complexity is 19 steps.

5 Sharing a sequential object

Consider a data structure that can be accessed through a fixed set of operations. We are given the code of each operation such that the code implements the operation correctly using atomic registers under the assumption that the object is accessed only sequentially, that is, by only one process. We also assume that the given code is bounded, that is, the number of steps required to execute the given sequential code of an operation is bounded, and this bound is known a priori. As an example, consider the object *stack* with three operations *push*, *pop*, and *is-empty*. It is possible to implement the stack as an array, where all these three operations take only a small number of steps, and this number is independent of the current state of the stack, or the value to be pushed. On the other hand, consider a data structure *set* that supports the operation *member* that tests whether its argument is a member of the set. Suppose we implement the set by a linked list of its elements. The membership operation will involve search, and the number of steps it takes will depend on the size of the set at the time of the invocation. Consequently, this implementation of *set* is not bounded. We can get a bounded implementation of a set (which supports a membership operation) using a hash table.

Typically, the given sequential code for any operation will involve several read and write operations to the shared registers modeling the object. A write operation is said to be *failure robust*, if a process failure immediately after the write operation does not leave the object in an inconsistent state. That is, in case of a process failure immediately after a failure-robust write, we may either assume that the operation was successfully completed, or assume that the operation was never executed. As an example, consider the object *stack* that is implemented as an array A and an integer c that keeps the count of its elements. The stack consists of the elements $A[0]$ through $A[c-1]$. Suppose the operation *push* first writes to the location $A[c]$ and then increments c. Here, both writes are failure robust; if a process fails before the second write, it is safe to assume that the process failed before executing *push*. Observe that if the operation *push* first increments c, and then writes the element into the array, the first write is not failure robust.

If all the write operations in the given sequential code are failure robust, then we can use the code of Figure 4 directly to transform the given sequential bounded implementation into a concurrent implementation. To access the object, process i simply executes the code of Figure 4 and in the critical section executes the given sequential code of the desired operation. The fact that the object is bounded gives us the desired value of K_{cs}. (Recall that $K_{cs} \cdot \Delta$ is the upper bound on the time that a process is allowed to spend in its critical section.) Each such access to the shared object finishes within a finite time in spite of the failures of other processes. In absence of contention, in addition to the number of steps of the sequential code, the process needs to execute 19 extra steps. Thus, the overhead is constant in absence of contention, and the implementation is fast.

It is possible to write bounded sequential code with only failure robust writes for various synchronization primitives such as test-and-set, fetch-and-add, and data structures such as queue, stack. This code can then be transformed as above to allow sharing.

Now we illustrate how to use the construction of Figure 4 even when the code contains writes that are not failure robust. Suppose the number of writes in a sequential operation accessing the object is bounded by a constant, say k. We introduce additional atomic registers $x_1, \ldots x_k$ and $y_1, \ldots y_k$, and an atomic bit b. We modify the code for each operation as follows. The bit b is initially *false*. At the i-th write, instead of modifying the object, the process writes the name of the register (or the location) in x_i, and the value it wants to write in y_i. At the end of the code, the process sets the bit b to *true*, and then actually updates the object; that is, for $i = 1, \ldots k$, writes the value y_i to the location specified by x_i. After this update, it resets the bit b to *false*. Consider what happens when a process fails in the middle of an operation. If b is *false* then the object is in a consistent state. If b is *true* then the object may be only partially updated, but all the information needed to finish the incomplete operation is available. Hence, any process that wants to update the object first tests b, and if b is *true*, it writes the value y_i to the location specified by x_i for $i = 1, \ldots k$, resets b to *false*, and then starts its own operation.

As an example, consider an array A that supports the operation *swap*(i, j) that swaps the values in the locations $A[i]$ and $A[j]$. Clearly, the operation has to update both the registers, and no matter in what order these two writes occur, the first one cannot be failure robust. The code of Figure 5 shows the above strategy. We can now use this code in the critical section of Figure 4 with $K_{cs} = 18$ (assuming each statement involves one access to a shared location).

```
procedure swap(i, j):
$x_1, x_2, y_1, y_2$: shared registers;
$b$ : shared bit, initially false;
    if $b$ then
        $A[x_1] := y_1$; $A[x_2] := y_2$; $b := false$ fi;
    $x_1 := i$; $y_1 := A[j]$;
    $x_2 := j$; $y_2 := A[i]$;
    $b := true$;
    $A[i] := y_1$; $A[j] := y_2$;
    $b := false$
end-procedure
```

Figure 5: Code for swapping.

References

[AT92] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992. Full version available from {*alur,gadi*}*@research.att.com*.

[Bur78] J.E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2):42–47, 1978.

[CS93] M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 183–194, August 1993.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[Her90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. 2nd ACM Symp. on Principles and Practice of Parallel Programming*, pages 197–206, 1990.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[LA87] M. C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Comp. Systems*, 5(1):1–11, 1987.

[LS92] N. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proc. of the 13th IEEE Real-Time Systems Symp.*, pages 2–11, December 1992.

[MT93] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[Plo89] S. A. Plotikin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Dist. Computing*, pages 159–175, August 1989.

[Ray86] M. Raynal. *Algorithms for mutual exclusion.* The MIT Press, 1986.