Wait-Free Synchronization in Quantum-Based Multiprogrammed Systems* (Extended Abstract)

James H. Anderson, Rohit Jain, and David Ott

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract. We consider wait-free synchronization in multiprogrammed uniprocessor and multiprocessor systems in which the processes bound to each processor are scheduled for execution using a scheduling quantum. We show that, in such systems, any object with consensus number P in Herlihy's wait-free hierarchy is universal for any number of processes executing on P processors, provided the scheduling quantum is of a certain size. We give an asymptotically tight characterization of how large the scheduling quantum must be for this result to hold.

1 Introduction

This paper is concerned with wait-free synchronization in multiprogrammed systems. In such systems, several processes may be bound to the same processor. In related previous work, Ramamurthy, Moir, and Anderson considered wait-free synchronization in multiprogrammed systems in which processes on the same processor are scheduled by priority [4]. For such systems, Ramamurthy et al. showed that any object with consensus number P in Herlihy's wait-free hierarchy [2] is universal for any number of processes executing on P processors, i.e., universality is a function of the number of *processors* in a system, not the number of *processes*. An object has *consensus number* C iff it can be used to solve C-process consensus, but not $(C+1)$-process consensus, in an asynchronous system in a wait-free manner. An object is *universal* in a system if it can be used to implement any other object in that system in a wait-free manner.

In this paper, we establish similar results for multiprogrammed systems in which quantum-based scheduling is used. Under quantum-based scheduling, each processor is allocated to its assigned processes in discrete time units called *quanta*. When a processor is allocated to some process, that process is guaranteed to execute without preemption for Q time units, where Q is the length of the quantum, or until it terminates, whichever comes first. In this paper, we show that quantum-based systems are similar to priority-based systems with regard

* Work supported by NSF grants CCR 9510156 and CCR 9732916, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAH04-95-1-0323. The first author was also supported by an Alfred P. Sloan Research Fellowship.

to universality. In particular, we show that any object with consensus number P in Herlihy's wait-free hierarchy is universal in a quantum-based system for any number of processes executing on P processors, *provided* the scheduling quantum is of a certain size. We give an asymptotically tight characterization of how large the scheduling quantum must be for this result to hold.

Our results are summarized in Table 1. This table gives conditions under which an object with consensus number C is universal in a P-processor quantum-based system. In this table, T_{max} (T_{min}) denotes the maximum (minimum) time required to perform any atomic operation, Q is the length of the scheduling quantum, and c is a constant that follows from the algorithms we present. Obviously, if $C < P$, then universal algorithms are impossible [2]. If $P \leq C \leq 2P$, then the smallest value of Q that suffices is a value proportional to $(2P + 1 - C)T_{max}$. If $2P \leq C < \infty$, then the smallest value of Q that suffices is a value proportional to $2T_{max}$. If $C = \infty$, then Q (obviously) can be any value [2].

An important special case of our main result is that reads and writes are universal in quantum-based uniprocessor systems ($P = 1$). In this case, the scheduling quantum must be large enough to encompass the execution of eight high-level language instructions (see Theorem 1). In any practical system, the scheduling quantum would be much larger than this. Thus, in practice, Herlihy's wait-free hierarchy collapses in multithreaded uniprocessor applications in which quantum-based scheduling is used.

It is important to note that the results of this paper do *not* follow from the previous results of Ramamurthy et al. concerning priority-based systems, because priority-based and quantum-based execution models are fundamentally incomparable. In a priority-based system, if a process p is preempted during an object invocation by another process q that invokes the same object, then p "knows" that q's invocation must be completed by the time p resumes execution. This is because q has higher priority and will not relinquish the processor until it completes. Thus, operations of higher priority processes "automatically" appear to be atomic to lower priority processes executing on the same processor. This is the fundamental insight behind the results of Ramamurthy et al.

In contrast, in a quantum-based system, if a process is ever preempted while accessing some object, then there are no guarantees that the process preempting it will complete any pending object invocation before relinquishing the processor. On the other hand, if a process can ever detect that it has "crossed" a quantum boundary, then it can be sure that the next few instructions it executes will be performed without preemption. Several of the algorithms presented in this paper employ such a detection mechanism. This kind of detection mechanism would be ill-suited for use in a priority-based system, because a process in such a system can never be "sure" that it won't be preempted by a higher-priority process.

Our quantum-based execution model is based on two key assumptions:

(i) If a process is preempted during an object invocation, then the first such preemption may happen at any point in time after the invocation begins.

(ii) When a process resumes execution after having been preempted, it cannot be preempted again until after Q time units have elapsed.

consensus number C	universal if:	not universal if:
P	$Q \geq c(P+1)T_{max}$	$Q \leq PT_{min}$
$P+1$	$Q \geq cPT_{max}$	$Q \leq (P-1)T_{min}$
$\vdots$	$\vdots$	$\vdots$
n, where $P \leq n < 2P$	$Q \geq c(2P+1-n)T_{max}$	$Q \leq (2P-n)T_{min}$
$\vdots$	$\vdots$	$\vdots$
$2P-2$	$Q \geq c3T_{max}$	$Q \leq 2T_{min}$
$2P-1$	$Q \geq c2T_{max}$	$Q \leq T_{min}$
$2P$	$Q \geq c2T_{max}$	$Q \leq T_{min}$
$2P+1$	$Q \geq c2T_{max}$	$Q \leq T_{min}$
$\vdots$	$\vdots$	$\vdots$
∞	$Q \geq 0$	—

Table 1. Conditions under which an object with consensus number C is universal for any number of processes in a P-processor quantum-based system.

Note in particular that we do not assume that each object invocation starts at the beginning of a quantum. This is because the objects we implement might be used in other algorithms, in which case it might be impossible to ensure that each invocation of an object begins execution at a quantum boundary. We also make no assumptions regarding how the next process to run is selected on a processor. Indeed, the process currently running on a processor may be allocated several quanta in succession before the processor is allocated to a different process — in fact, the processor may *never* be allocated to another process.

These assumptions rule out certain trivial solutions to the problems we address. For example, the above-mentioned result about the universality of reads and writes in quantum-based uniprocessors is obtained by presenting a wait-free implementation of a consensus object that uses only reads and writes. It may seem that such an implementation is trivial to obtain: simply define the quantum to be large enough so that any consensus invocation fits within a single quantum! However, our model precludes such a solution, because we do not assume that each object invocation starts at the beginning of a quantum.

It is similarly fruitless to implement a uniprocessor consensus object by requiring each process to repeatedly test some condition (i.e., by busy waiting) until a quantum boundary has been crossed, and to then perform the consensus invocation (safely) within the new quantum. This is because, for a process to detect that a quantum boundary has been crossed, it must eventually be preempted by another process. Such a preemption may never occur. Even if we were able to assume that such a preemption would eventually occur (e.g., because round-robin scheduling was being used), the proposed solution should still be rejected. This is because it forces operations (on consensus objects, in this case) to be performed at the rate the processor is switched between processes. This defeats one of the main attractions of the quantum-based execution model:

in most quantum-based systems, the scheduling quantum is large enough to allow many operations to be performed safely inside a single quantum without any fear of interferences, i.e., the rate at which operations potentially could be performed is *much* higher than the rate of process switches.

The remainder of this paper is organized as follows. In Sect. 2, we present definitions and notation that will be used in the remainder of the paper. Then, in Sect. 3, we present our results for quantum-based uniprocessor systems. We begin by presenting a wait-free, constant-time implementation of a consensus object that uses only reads and writes and a quantum of constant size. This implementation proves that reads and writes are universal in quantum-based uniprocessor systems [2]. Object implementations of practical interest are usually based on synchronization primitives such as *compare-and-swap* (`C&S`), not consensus objects. We show that, given a quantum of constant size and using only reads and writes, `C&S` can be implemented in a quantum-based uniprocessor system in constant time. We also show that any read-modify-write primitive can be implemented in constant time as well. In Sect. 4, we present our results for quantum-based multiprocessor systems. Our goal in this section is to establish universality results for objects with consensus number C, where $C \geq P$. We do this by showing how to use such objects to implement a wait-free consensus object for any number of processes running on P processors. As C varies, the quantum required for this consensus implementation to work correctly is as given in Table 1. In the full paper, we prove that our characterization of the required quantum is asymptotically tight [1]. This proof is omitted here due to space limitations. We end the paper with concluding remarks in Sect. 5.

2 Definitions and Notation

A *quantum-based system* consists of a set a processes and a set of processors. In most ways, quantum-based systems are similar to shared-memory concurrent systems as defined elsewhere. For brevity, we focus here on the important differences. In a quantum-based system, each process is assigned to a distinct processor. Associated with any quantum-based system is a *scheduling quantum* (or *quantum* for short), which is a nonnegative integer value. In an actual quantum-based system, the quantum would be given in time units. In this paper, we find it convenient to more abstractly view a quantum as specifying a statement count. This allows us to avoid having to incorporate time explicitly into our model. Informally, when a processor is allocated to some process, that process is guaranteed to execute without preemption for at least Q atomic statements, where Q is the value of the quantum, or until it terminates.

Our programming notation should be self explanatory; as an example of this notation, see Fig. 1. In this and subsequent figures, each numbered statement is assumed to be atomic. When considering a given object implementation, we consider only statement executions that arise when processes perform operations on the given object, i.e., we abstract away from the other activities of these processes outside of object accesses. For "long-lived" objects that may be invoked

repeatedly, we assume that when a process completes some operation on the object, that process's program counter is updated to point to the first statement of some nondeterministically-selected operation of the object.

We define a program's semantics by a set of histories. A *history* of a program is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state and $t_i \xrightarrow{s_i} t_{i+1}$ denotes that state t_{i+1} is reached from state t_i via the execution of statement s_i; unless stated otherwise, a history is assumed to be a maximal such sequence. Consider the history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots t_i \xrightarrow{s_i} t_{i+1} \cdots t_j \xrightarrow{s_j} t_{j+1} \cdots$, where s_i and s_j are successive statement executions by some process p. We say that p *is preempted before* s_j in this history iff some other process on p's processor executes a statement between states t_{i+1} and t_j. A history $h = t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ is *well-formed* iff it satisfies the following condition: for any statement execution s_j in h by any process p, if p is preempted before s_j, then no process on p's processor other than p executes a statement after state t_{j+1} until either (i) p executes at least Q statements or (ii) p's object invocation that includes s_j terminates. We henceforth assume all histories are well-formed. We define program properties using invariants and stable assertions. An assertion is *stable* in a history iff it holds from some state onward. An assertion is an *invariant* in a history iff it is stable and initially true.

Notational Conventions: The number of processes and processors in the system are denoted N and P, respectively. Processors are labeled from 1 to P. M denotes the maximum number of processes on any processor. Q denotes the value of the quantum, and C will be used to refer to a given object's consensus number (see Sect. 1). Unless stated otherwise, p, q, and r are assumed to be universally quantified over process identifiers. The predicate $running(p)$ holds at a state iff process p is the currently-running process on its processor at that state. The predicate $p@s$ holds iff statement s is the next statement to be executed by process p. We use $p@S$ as shorthand for $(\exists s : s \in S :: p@s)$, $p.s$ to denote statement number s of process p, $p.v$ to denote p's local variable v, and $pr(p)$ to denote process p's processor. *valtype* denotes an arbitrary type. □

3 Uniprocessor Systems

In this section, we present constant-time implementations of a consensus object (Sect. 3.1), a `C&S` object (Sect. 3.2), and a read-modify-write object (Sect. 3.3) for quantum-based uniprocessor systems. Each of these implementations uses only reads and writes and requires a quantum of constant size.

3.1 Consensus

Our uniprocessor consensus algorithm is shown in Fig. 1. Here Q is assumed to be eight atomic statements. In lines 1-14 of the algorithm, the worst-case execution sequence consists of nine atomic statements (in particular, lines 1-8 and 14). Thus, with $Q = 8$, a process can be preempted at most once while executing within lines 1-14. Note that each process p both begins and ends this

```
shared variable  Dec1, Dec2: valtype ∪ ⊥ initially ⊥;
                 Run: 1..N

procedure decide(in: valtype) returns valtype
private variable val: valtype ∪ ⊥        /* local to process p, the invoking process */

1:      Run := p;
2:      if Dec2 = ⊥ then
3:          Dec1 := in;
4:          if Run ≠ p then
5:              val := Dec2;             /* statements 5-9 execute without preemption */
6:              if val = ⊥ then
7:                  Dec1 := in;
8:                  Dec2 := in
                else
9:                  Dec1 := val
                fi
            else
10:             Dec2 := in;
11:             if Run ≠ p then
12:                 val := Dec1;         /* statements 12-13 execute without preemption */
13:                 Dec2 := val
                fi
            fi
        fi;
14:     Run := p;
15:     return Dec2
```

Fig. 1. Uniprocessor consensus using reads and writes.

code sequence by assigning $Run := p$. Thus, if any process p is preempted while executing within lines 1-14 by another process q that also executes within lines 1-14, then $Run \neq p$ holds by the time p resumes execution. This is because q's execution of lines 1-14 can itself be preempted at most once, and thus if q executes any of these statements within a quantum, then it must execute $q.1$ or $q.14$ or both within that quantum. This handshaking mechanism is typical of those employed in the algorithms in this paper to enable a process to detect if it has been preempted.

Having explained the manner in which preemptions are detected, we can now describe the rest of the algorithm. Two shared "decision" variables are employed, $Dec1$ and $Dec2$. Both are initially $\perp$, and it is assumed that no process's input value is $\perp$. All processes return the value assigned to $Dec2$. Before returning, each process attempts to assign its input value to both $Dec1$ and $Dec2$ in sequence.

To understand how the algorithm works, suppose that a process p is preempted just before executing the assignment to $Dec1$ at line 3. If, while p is preempted, another process q executes within lines 1-14, then p will detect this when it resumes execution and then execute lines 5-9. When p resumes execu-

tion, it will immediately perform the assignment at line 3. Note that p may be assigning $Dec1$ here very "late", i.e., well after other processes have reached a decision and terminated. However, the algorithm ensures that p's late assignment does not cause some process to return an erroneous value. To see this, note that because p has already been preempted once during lines 1-14, it executes lines 5-9 without preemption. Thus, it can safely deal with its late assignment without any fear of interferences due to further preemptions. The late assignment is dealt with as follows. If a decision has not yet been reached, then p assigns its own input value to both $Dec1$ and $Dec2$ (lines 7 and 8). Otherwise, p "undoes" its late assignment to $Dec1$ by copying to $Dec1$ the current value within $Dec2$ (line 9). The need to "undo" late assignments is the main reason why the algorithm uses two decision variables — to restore the value of one variable, another variable is needed.

A process p potentially could also be preempted just before performing the assignment to $Dec2$ at line 10. If, while p is preempted, another process q executes within lines 1-14, then p will detect this when it resumes execution and then execute lines 12 and 13 without preemption. These lines "undo" p's potentially "late" assignment to $Dec2$ by copying to $Dec2$ the current value of $Dec1$.

The correctness of this algorithm follows from the following three lemmas, which due to space limitations are stated here without proof.

Lemma 1: $p@15 \Rightarrow (\exists q :: Dec2 = q.in)$ *is an invariant.* □

Lemma 2: $Dec2 \neq \bot \Rightarrow ((Dec1 = Dec2) \vee (\exists p :: running(p) \wedge p@\{4..9, 11..13\}))$ *is an invariant.* □

Lemma 3: $(\forall v : v \neq \bot :: (Dec1 = v \wedge Dec2 = v) \vee (\exists p :: running(p) \wedge p@\{4..9\} \wedge Dec2 = v) \vee (\exists p :: running(p) \wedge p@\{11..13\} \wedge Dec1 = v))$ *is stable.* □

Theorem 1: *In a quantum-based uniprocessor system with $Q \geq 8$, consensus can be implemented in constant time using only reads and writes.* □

3.2 Compare-and-Swap

Our uniprocessor `C&S` implementation is shown in Fig. 2. The most important shared variables in the implementation are $X1$ and $X2$. Each of these variables has three fields, *val*, *proc*, and *alt*. $X2.val$ gives the "current" value of the implemented object at all times. It can be seen by inspecting lines 25-48 that the way in which $X1$ and $X2$ are assigned is very reminiscent of the way $Dec1$ and $Dec2$ were assigned in our consensus algorithm. After each is assigned, a check is made to see if a preemption has occurred, in which case the assignment is undone if necessary. Assuming Q is defined to be large enough so that each `C&S` invocation is preempted at most once, this "undo code" cannot be preempted.

In our consensus algorithm, undoing a late assignment was relatively simple because a consensus object is accessed only once by each process. For the sake of comparison, consider what happens when a process p detects that $Run \neq p$ at line 11 in Fig. 1. For $Run \neq p$ to be detected, p must have been preempted

```
type
    X-type = record val: valtype;  proc: 1..N;  alt: 0..1 end /* stored in one word */

shared variable
    Seen1, Seen2: array [1..N, 0..1] boolean;
    Run: 1..N;
    X1, X2: X-type  initially (v, (1, 0)), where v is object's initial value

procedure C&S(old,new: valtype)
                        returns boolean
private variable
    /* p denotes the invoking process */
    v: X-type;
    b: boolean

1:   if old = new then
2:     return X2.val = old
     fi;
3:   Run := p;
4:   v := X2;
5:   Seen1[v.proc, v.alt] := true;
6:   if Run ≠ p then
       /* lines 7-11 nonpreemptable */
7:     b := Seen2[v.proc, v.alt];
8:     Seen1[v.proc, v.alt] := b;
9:     v := X2;
10:    Seen1[v.proc, v.alt] := true;
11:    Seen2[v.proc, v.alt] := true
     else
12:    Seen2[v.proc, v.alt] := true;
13:    if Run ≠ p then
         /* lines 14-18 nonpreemptable */
14:      b := Seen1[v.proc, v.alt];
15:      Seen2[v.proc, v.alt] := b;
16:      v := X2;
17:      Seen1[v.proc, v.alt] := true;
18:      Seen2[v.proc, v.alt] := true
       fi
     fi;
19:  if v.val ≠ old then
20:    Run := p;
21:    return false
     fi;
22:  alt := 1 − alt;
23:  Seen1[p, alt] := false;
24:  Seen2[p, alt] := false;
25:  X1 :=(new, p, alt);
26:  if Run ≠ p then
       /* lines 27-35 nonpreemptable */
27:    v := X2;
28:    X1 := v;
29:    Seen1[v.proc, v.alt] := true;
30:    Seen2[v.proc, v.alt] := true;
31:    if ¬Seen2[p, alt] then
32:      if X2.val = old then
33:        X1 :=(new, p, alt);
34:        X2 :=(new, p, alt);
35:        Seen2[p, alt] := true
        fi
      fi
     else
36:    X2 :=(new, p, alt);
37:    if Run ≠ p then
         /* lines 38-46 nonpreemptable */
38:      v := X1;
39:      X2 := v;
40:      Seen1[v.proc, v.alt] := true;
41:      Seen2[v.proc, v.alt] := true;
42:      if ¬Seen2[p, alt] then
43:        if X2.val = old then
44:          X1 :=(new, p, alt);
45:          X2 :=(new, p, alt);
46:          Seen2[p, alt] := true
          fi
        fi
       else
47:      Seen2[p, alt] := true
       fi
     fi;
48:  Run := p;
49:  return Seen2[p, alt]
```

Fig. 2. Uniprocessor C&S implementation. For simplicity, the object being accessed is left implicit. To be precise, the object's address should be passed as a parameter to the C&S procedure. The object can be read by reading $X2.val$.

either before or after its assignment to $Dec2$ at line 10. If it was preempted after assigning $Dec2$, then it assigned both $Dec1$ and $Dec2$ (lines 3 and 10) without preemption, in which case copying the value of $Dec1$ to $Dec2$ in lines 12 and 13 has no effect. (The value copied from $Dec1$ must equal that assigned to $Dec1$ by p: another process q can alter $Dec1$ only if it executes lines 2 and 3 without preemption and detects $Dec2 = \perp$ at line 2.) On the other hand, if p was preempted before assigning $Dec2$, then this is a potentially late assignment that may have overwritten a previously agreed upon decision value. In this case, copying the value of $Dec1$ to $Dec2$ in lines 12 and 13 undoes the overwrite.

Undoing a late assignment in our `C&S` implementation is much more complicated, because each process can perform repeated `C&S` operations. Some of the subtleties involved can be seen by considering what happens when a process p detects that $Run \neq p$ upon executing line 37 in Fig. 2 (the counterpart to the situation considered above for our consensus algorithm). By our handshaking mechanism, this implies that p was preempted either before or after its assignment to $X2$ at line 36. The question is: Should this assignment to $X2$ be undone? Given that $X2.val$ defines the current state of the implemented object, the answer to this question depends on whether the value assigned to $X2$ by p has been "seen" by another process. If p's value has been seen, then its assignment to $X2$ *cannot* be undone. There is no way for p to infer that its value has been seen by inspecting the value of $X2$ (or $X1$, for that matter), so an additional mechanism is needed. In our implementation, the *Seen* flags provide the needed mechanism. There are two pairs of such flags for each process p, $Seen1[p, 0]/Seen2[p, 0]$ and $Seen1[p, 1]/Seen2[p, 1]$. p alternates between these pairs from one `C&S` to the next. The current pair is given by p's *alt* variable. If p detects that $Seen2[p, alt]$ holds, then it knows that a value it has assigned to $X1$ or $X2$ has been seen by another process. Two pairs of *Seen* flags are needed to distinguish values assigned by p in consecutive `C&S` operations. Two *Seen* flags per pair are needed to be able to undo late assignments to these flags after preemptions. The *proc* and *alt* fields in $X1$ and $X2$ allow a process to detect which *Seen* flags to use.

Given the above description of the shared variables that are used, it is possible to understand the basic structure of the code. Trivial `C&S` operations for which $old = new$ are handled in lines 1 and 2. Lines 3-49 are executed to perform a nontrivial `C&S`. In lines 3-18, the current value of the implemented object is read and the process that wrote that value is informed that its value has been seen by updating that process's *Seen* flags. The code sequence that is used here is similar to that employed in our consensus algorithm to update $Dec1$ and $Dec2$. In lines 19-21, a check is made to see if the current value of the object matches the specified old value. If they do match, then lines 22-49 are executed to attempt to update the object to hold the specified new value. In lines 23 and 24, the invoking process's *Seen* flags are initialized. The rest of the algorithm is as described above. First $X1$ is written, and a preemption check is performed. If a preemption occurs, then the assignment to $X1$ is undone if necessary. After writing $X1$, $X2$ is written. Once again, a preemption check is performed and the assignment is undone if necessary. Note that a process may update another

```
procedure RMW(Addr: pointer to valtype; f: function) returns valtype
private variable old, new: valtype
1:      old := *Addr;
2:      new := f(old);
3:      if C&S(Addr, old, new) = false then
4:          old := *Addr;   /* statements 4 and 5 execute without preemption */
5:          *Addr := f(old)
        fi;
6:      return old
```

Fig. 3. Uniprocessor read-modify-write implementation.

process's *Seen* flags in lines 29 and 30 and in lines 40 and 41. Because these lines are each within a code fragment that is executed without preemption, the *Seen* flags can be updated here without resorting to a more complicated code sequence like in lines 5-18.

The formal correctness proof of our `C&S` implementation is not hard, but it is quite long, so due to space limitations, we defer it to the full paper. The correctness proof hinges upon the assumption that a process can be preempted at most once while executing within lines 3-48. A `C&S` invocation completes after at most 26 atomic statement executions. The worst case occurs when the following statements are executed in order: 1, 3-6, 12, 13, 19, 22-26, 36-46, 48, 49. Thus, lines 3-48 give rise to at most 24 statement executions. It follows that if $Q \geq 23$, then our preemption requirement is met. This gives us the following theorem.

Theorem 2: *In a quantum-based uniprocessor system with $Q \geq 23$, any object that is accessed only by means of read and* `C&S` *operations can be implemented in constant time using only reads and writes.* □

3.3 Other Read-Modify-Write Operations

A RMW operation on a variable X is characterized by specifying a function f. Informally, such an operation is equivalent to the atomic code fragment $\langle x := X;\ X := f(x);$ **return** $x \rangle$. Example RMW operations include fetch-and-increment (`F&I`), fetch-and-store, and test-and-set. RMW operations can be implemented on a uniprocessor as shown in Fig. 3. If the `C&S` at line 3 succeeds, then the RMW operation atomically takes effect when the `C&S` is performed. If the `C&S` fails, then the invoking process must have been preempted between lines 1 and 3. Provided Q is defined to be large enough so that lines 1-5 can be preempted at most once, lines 4 and 5 execute without preemption. If we implement `C&S` as in Fig. 2, then the `C&S` invocation consists of at most 26 statement executions. Thus, a value of $Q \geq 26 + 3$ suffices. This gives us the following theorem.

Theorem 3: *In a quantum-based uniprocessor system with $Q \geq 29$, any object accessed only by means of read, write, and read-modify-writes can be implemented in constant time using only reads and writes.* □

4 Multiprocessor Systems

In this section, we show that wait-free consensus can be implemented for any number of processes in a P-processor quantum-based system using C-consensus objects (i.e., objects that implement consensus for C processes), where $C \geq P$, provided Q is as specified in Table 1. For simplicity, we assume here that $C \leq 2P$, because for larger values of C, the implementation we give for $C = 2P$ can be applied to obtain the results of Table 1. The consensus implementation we present to establish the results in this table is shown in Fig. 4. In addition to C-consensus objects, a number of uniprocessor `C&S` and `F&I` objects are used in the implementation. Recall from Sect. 3 that these uniprocessor objects can be implemented in constant time using only reads and writes. We use "*local*-`C&S`" and "*local*-`F&I`" in Fig. 4 to emphasize that these are uniprocessor objects.

In our implementation, processes choose a decision value by participating in a series of "consensus levels". There are L consensus levels, as illustrated in Fig. 5. L is a function of M and P, as described below. Each consensus level consists of a C-consensus object, where $C = P + K$, $0 \leq K \leq P$. Also associated with each consensus level l is a collection of shared variables $Outval[l, i]$, where $1 \leq i \leq P$. $Outval[l, i]$ is used by processes on processor i to record the decision value from level l (see line 17 in Fig. 4). When a process assigns a value to $Outval[l, i]$, we say that it "publishes" the result of level l. The requirement that at most C processes can access a C-consensus object is enforced by defining $P + K$ "ports" per consensus level. Processors 1 through K have two ports per object, and processors $K + 1$ through P have one port. A process can access a C-consensus object only by first claiming one of the ports allocated to its processor. A process claims a port by executing lines 4-12. One can think of all ports across all consensus levels as being numbered in sequence, starting at 1. Ports can be claimed on each processor i by simply performing a `F&I` operation on a counter $Port[i]$ that is local to processor i, with one special case as an exception. This special case arises when a process p executing without preemption on a processor $i \leq K$ (which has two ports per level) accesses the first of processor i's ports at some level l. In this case, if p simply increments $Port[i]$, then it is then positioned to access the second port of level l, which is pointless because a decision has already been reached at that level. To correct this, $Port[i]$ is updated in such a case using a `C&S` operation in line 8.

The consensus levels are organized into a sequence of blocks as shown in Fig. 5. The significance of these blocks is explained below. Each process attempts to participate in each consensus level in the order depicted in Fig. 5, skipping over levels for which a decision value has already been published. When a process accesses some level, the input value it uses is either the output of the highest-numbered consensus level for which there is a published value on its processor, or its own input value, if no previously-published value exists (see lines 2, 13, and 14). So that an input value for a level can be determined in constant time, a counter $Lastpub[i]$ is used on each processor i to point to the highest-numbered level that has a published value on processor i. Due to preemptions, $Lastpub[i]$ may need to be incremented to skip over an arbitrary number of levels. It is

```
constant L = (M(P − K)^2 + 1)(KM + 1)
          /* total number of consensus levels for C = P + K, where 0 ≤ K ≤ P */

shared variable
    Lastpub: array[1..P] of 0..L  initially 0;
        /* latest level on a processor for which there is a published consensus value */
    Outval: array[1..L, 1..P] of valtype;
                    /* Outval[l, i] is the consensus value for level l on processor i */
    Port: array[1..P] of 1..2L + M  initially 1 /* next available port on processor */

procedure decide(val : valtype) returns valtype
private variable                                                    /* local to process p */
    input, output: valtype;                                  /* input/output value for a level */
    level, last_level: 0..L + M;                          /* current (last) level accessed by p */
    numports: 1..2;      /* number of ports per consensus object on processor pr(p) */
    port, newport: 1..2L + M;                                          /* port numbers */
    publevel: 0..L /* last level for which there is published value on processor pr(p) */

1:  if pr(p) ≤ K then numports := 2 else numports := 1 fi;
2:  input, last_level, level := val, 0, 0;
3:  while level ≤ L do
4:      port := Port[pr(p)];                                  /* determine port and level */
5:      level := ((port − 1) div numports) + 1;
6:      if last_level = level then          /* if level didn't change, make correction */
7:          newport := port + numports;
8:          if local-C&S(&Port[pr(p)], port, newport + 1) then port := newport
9:          else port :=local-F&I(Port[pr(p)])
10:         fi
        else
11:         port :=local-F&I(Port[pr(p)])
        fi;
12:     level := ((port − 1) div numports) + 1;
13:     publevel := Lastpub[pr(p)];                 /* determine input for next level */
14:     if publevel ≠ 0 then input := Outval[publevel, pr(p)] fi;
15:     if level ≤ L then    /* necessary because F&I may overshoot the last level */
16:         output := C-consensus(level, input);  /* invoke the C-consensus object */
17:         Outval[level, pr(p)] := output;                  /* publish the result */
18:         local-C&S(&Lastpub[pr(p)], publevel, level)
        fi
19:     last_level := level;
    od;
20: publevel := Lastpub[pr(p)];
21: return(Outval[publevel, pr(p)])
```

Fig. 4. Multiprocessor consensus implementation.

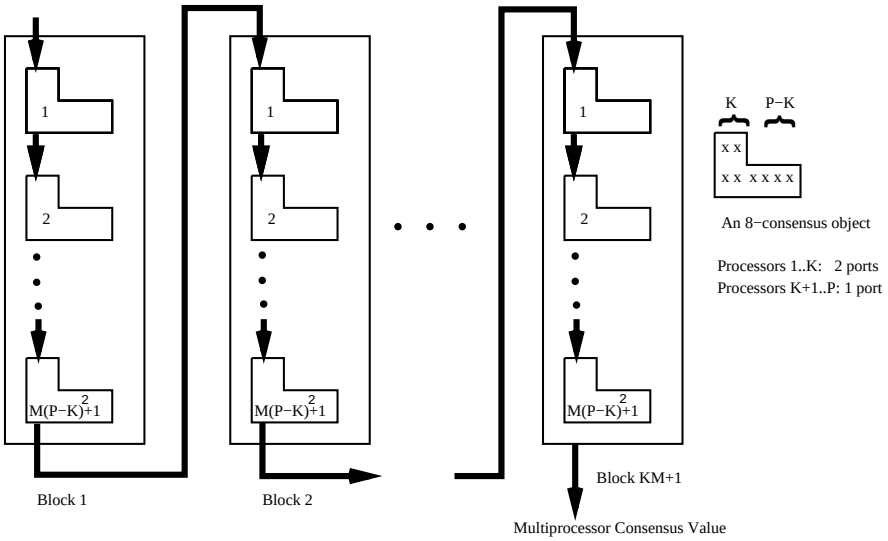

Fig. 5. Organization of the consensus levels in the implementation in Fig. 4.

therefore updated using a `C&S` operation instead of `F&I` (see line 18). Each process on processor i completes by returning the output value from level $Lastpub[i]$.

When a process p attempts to determine an input value for a level, there may be a number of previous levels that are inaccessible to p (because all available ports have been claimed) yet no decision value has been published. This can happen for a previous level l only if the process(es) on p's processor that accessed level l were preempted before publishing an output value for that level. We say that the preempted process(es) at level l *cause an access failure at level* l.

Obviously, there is a correlation between the number of access failures that can occur on a processor and the number of preemptions that can occur on that processor. The latter in turn depends on the size of the scheduling quantum Q. We show below that with a suitable choice of Q, the number of levels for which an access failure occurs on each processor is limited to a fraction of all the levels. Using a pigeon-hole argument, it is possible to show that, in any history, there exists some level for which no process on any processor experiences an access failure. We call such a level a *deciding level*. A simple inductive argument shows that at all levels below a deciding level l, the output value of level l is used by *every* process on *every* processor when accessing any level below l, even if access failures occur when accessing levels lower than l.

We now state some lemmas about the number of access failures that may occur in a history. We first consider the two extremes $C = 2P$ and $C = P$ and then the general case (proofs of Lemmas 4 and 5 can be found in [1]).

Lemma 4: *Suppose that $C = 2P$ and that Q is large enough to ensure that each process can be preempted at most once while accessing any two consensus levels in succession (the two levels don't have to be consecutive). If processes p and q on processor i cause an access failure at level l, then at least one of p and q does not cause an access failure at any level less than l.* □

Corollary 1: *If C and Q are as defined in Lemma 4, and if there are at most M processes per processor, then there can be at most M access failures on any processor. Furthermore, there exists a deciding level among any $MP+1$ levels.* □

Lemma 5: *Suppose that $C = P$ and that Q is large enough to ensure that each process can be preempted at most once while accessing any $P+1$ consensus levels in succession (these levels don't have to be consecutive). If there are at most M processes on any processor, then there exists a deciding level within any consecutive $MP^2 + 1$ consensus levels.* □

Lemma 6: *Suppose that $C = P + K$, where $0 \leq K \leq P$, and that Q is large enough to ensure that each process can be preempted at most once while accessing any $P - K + 1$ consensus levels in succession (these levels don't have to be consecutive). If there are at most M processes on any processor, then there exists a deciding level in any consecutive $(M \cdot (P-K)^2 + 1) \cdot (KM + 1)$ levels.*

Proof: For processors $K + 1$ through P, we know from Lemma 5 that there exists a deciding level among any consecutive $M \cdot (P-K)^2 + 1$ levels. If we have $KM + 1$ groups of $M \cdot (P-K)^2 + 1)$ levels each, then processors $K + 1$ through P have at least one deciding level in each group. Also, by Corollary 1, processors 1 through K can experience at most KM access failures in total. Thus, there exists at least one level that is a deciding level for the whole system. □

The proof of Lemma 6 reveals the insight as to why we group the levels into $KM + 1$ blocks as depicted in Fig. 5. It is easy to see that each consensus level is accessed in constant time (recall that our uniprocessor `C&S` and `F&I` algorithms take constant time). Thus, letting c denote the worst-case number of statement executions per level, we have the following (it can be shown that $c = 96$ suffices).

Theorem 4: *In a P-processor, quantum-based system, consensus can be implemented in a wait-free manner in polynomial space and time for any number of processes using read/write registers and C-consensus objects if $C \geq P$ and $Q \geq \max(2c,\ c(2P + 1 - C))$.* □

In the full paper, we prove the following theorem, showing that the quantum used in the implementation above is asymptotically tight [1].

Theorem 5: *In a P-processor, quantum-based system, consensus cannot be implemented in a wait-free manner for any number of processes using read/write registers and C-consensus objects if $C \geq P$ and $Q \leq \max(1,\ 2P - C)$.* □

If we were to add time to our model, then we could easily incorporate the T_{max} and T_{min} terms given in Table 1 in the bounds on Q given above.

```
shared variable P: array[1..3] of valtype ∪ ⊥ initially ⊥
procedure decide(val: valtype) returns valtype
private variable v, w: valtype
1: v := val;
2: for i := 1 to 3 do
3:     w := P[i];
4:     if w ≠ ⊥ then
5:         v := w
       else
6:         P[i] := v
       fi
   od;
7: return P[3]
```

Fig. 6. Moir and Ramamurthy's uniprocessor consensus algorithm.

5 Concluding Remarks

Our work was partially inspired by a read/write consensus algorithm for quantum-based uniprocessor systems due to Moir and Ramamurthy [3]. Actually, their goal was to design wait-free algorithms for multiprocessor systems in which the processor-to-memory bus is allocated to processors using quantum-based scheduling. Their consensus algorithm, which is shown in Fig. 6, is also correct in a quantum-based uniprocessor system. This algorithm is correct if $Q = 8$ (this requires first replacing the **for** loop by straight-line code), just like our uniprocessor consensus algorithm. However, their algorithm requires fewer references to shared memory. The algorithm employs three shared variables, $P[1]$, $P[2]$, and $P[3]$. The idea is to attempt to copy a value from $P[1]$ to $P[2]$, and then to $P[3]$. We found our mechanism of detecting preemptions to be much easier to employ when implementing other objects.

References

1. J. Anderson, R. Jain, and D. Ott. Wait-free synchronization in quantum-based multiprogrammed systems, May 1998. Available at `http://www.cs.unc.edu/~anderson/papers.html`.
2. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
3. M. Moir and S. Ramamurthy. Private communication. 1998.
4. S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal support. *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pp. 233–242. 1996.

This article was processed using the LaTeX macro package with LLNCS style