Real-Time Object Sharing with Minimal System Support*

(Extended Abstract)

Srikanth Ramamurthy, Mark Moir, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

We show that, for a large class of hard real-time systems, any object with consensus number P in Herlihy's wait-free hierarchy is universal for any number of tasks executing on P processors. These results exploit characteristics of priority-based schedulers common to most hard real-time systems. An important special case of this result is that, for hard real-time applications on uniprocessors, reads and writes are universal. Thus, Herlihy's hierarchy collapses for such applications.

1 Introduction

This paper is concerned with implementations of shared objects in hard real-time systems. A *hard real-time system* is comprised of a collection of tasks that execute on one or more processors. A *task* is a sequential program that may be invoked repeatedly in response to an external stimulus or timer. Each invocation of a task must complete execution by a specified *deadline*. Tasks are prioritized, usually with several tasks multiprogrammed on a single processor. On a given processor, a task may execute only if there are no higher-priority tasks on that processor that are ready for execution.

Traditional approaches to implementing shared objects in hard real-time systems rely on lock-based synchronization protocols. The main problem when using such protocols is that of *priority inversion*, i.e., the situation in which a given task waits for another task of lower priority to unlock an object. Mechanisms such as the *priority ceiling protocol* (PCP) [13, 14] are used to solve this problem. The PCP requires the kernel to dynamically adjust task priorities to ensure that a task within a critical section executes at a priority that is sufficiently high to bound the duration of any priority inversion. The PCP provides a general framework for real-time synchronization, but at the price of added operating system overhead.

Wait-free and lock-free shared object implementations offer an attractive alternative to lock-based protocols such as the PCP. In such object implementations, progress is guaranteed in the face of process[1] delays. Specifically, if several processes attempt to access a *lock-free* (*wait-free*) object concurrently, and if some proper subset of these processes stop taking steps, then *one* (*each*) of the remaining processes must complete its access in a finite number of its own steps.

From a real-time perspective, lock-free and wait-free object implementations are of interest because they avoid priority inversions with no kernel support. However, such implementations have not gained acceptance among real-time practitioners. For the case of wait-free implementations, this lack of acceptance may partially stem from the fact that wait-free algorithms can require space and time overhead that is somewhat high for some objects. However, on multiprocessors, this overhead is probably far outweighed by the scheduling overhead required of lock-based protocols (although, to our knowledge, no one has verified this experimentally). On uniprocessors, Anderson, Ramamurthy, and Jeffay have shown that such overhead can be avoided altogether by using lock-free object implementations, rather than wait-free ones [5]. On the surface, it is not immediately apparent that lock-free shared objects can be employed if tasks must adhere to strict timing constraints. In particular, lock-free implementations permit concurrent operations to interfere with each other, and repeated interferences can cause a given operation to take an arbitrarily long time to complete. The main contribution of [5] is to show that such interferences can be bounded on uniprocessors by judicious scheduling.

Another possible reason for the lack of acceptance of wait-free and lock-free object implementations among real-time practitioners is that many published implementations require strong synchronization primitives

*Work supported, in part, by NSF contract CCR 9216421, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAH04-95-1-0323. The second author was also supported by a UNC Alumni Fellowship.

[1] We use the term "task" when referring to real-time systems, and "process" when referring to general systems.

such as compare-and-swap (C&S). If such primitives are not available, then they must be "simulated", and the most obvious way to do so is by using lock-based protocols. However, this brings us back to problems associated with locking — problems that wait-free and lock-free objects are designed to eliminate.

The fact that many published wait-free and lock-free object implementations are based on strong synchronization primitives is no accident. Herlihy has shown that such strong primitives are, in general, necessary for these implementations [6]. Herlihy's results are based upon a categorization of synchronization objects by "consensus number". An object has *consensus number* N if it can be used to solve N-process consensus, but not $(N+1)$-process consensus, in a wait-free (or lock-free) manner. Herlihy's results show that synchronization objects with unbounded consensus numbers are necessary in general-purpose wait-free or lock-free object implementations. Such objects are called *universal* because they can be used to implement any other object. Common universal primitives include C&S and load-linked/store-conditional (LL/SC).

In this paper, we call into question the applicability of Herlihy's results to hard real-time systems. Specifically, we show that primitives that are weaker than indicated by Herlihy's hierarchy suffice for wait-free and lock-free implementations in such systems. The basis for our results is the realization that certain task interleavings cannot occur within hard real-time systems. For example, if a task T_i accesses an object in the time interval $[t, t']$, and if another task T_j on the same processor accesses that object in the interval $[u, u']$, then it is not possible to have $t < u < t' < u'$, because the higher-priority task must finish its operation before relinquishing the processor. Requiring an object implementation to correctly deal with this interleaving is pointless, because it cannot arise in practice. The distinction between traditional asynchronous systems, to which Herlihy's work is directed, and hard real-time systems is illustrated in Figure 1.

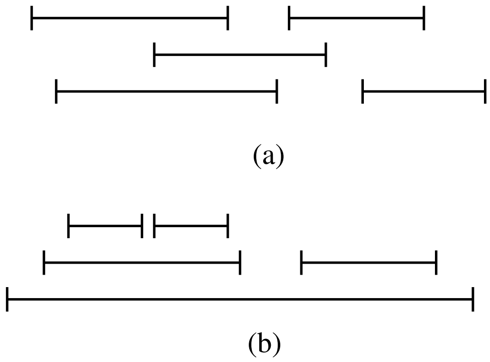

Figure 1: Line segments denote operations on shared objects with time running from left to right. Each level corresponds to operations by a different task. (a) Interleaved operations in a asynchronous system. Operations may overlap arbitrarily. (b) Interleaved operations in a uniprocessor real-time system. Two operations overlap only if one is contained within the other.

The real-time task model we assume is characterized by two key requirements: (i) on a given processor, a task T_i may preempt another task T_j only if T_i has higher priority than T_j; (ii) a task's priority can change over time, but not during any object access. We primarily consider systems in which a high-priority task can arbitrarily preempt any low-priority task executing on the same processor. However, we do relax this assumption by allowing short nonpreemptable code fragments when considering object implementations for multiprocessors.

Requirement (i) is fundamental to all priority-driven scheduling policies. Requirement (ii) holds for most common policies, including rate-monotonic (RM) [11], deadline-monotonic (DM) [10], and earliest-deadline-first (EDF) [11] scheduling. Under RM and DM scheduling, task priorities are static, i.e., do not change over time. Under EDF scheduling, each invocation of a task is assigned a priority based upon its current deadline, and this priority does not change during that invocation (and hence during any object access). Requirement (ii) also holds for variations of RM, DM, and EDF scheduling in which tasks are broken into phases that are allowed to have distinct priorities [7]. The only common scheduling policy that we know of that violates requirement (ii) is least-laxity-first (LLF) scheduling [12]. Under LLF scheduling, the priority of a task invocation *can* change during its execution.

Observe that requirements (i) and (ii) expressly preclude the use of locking within a processor. In particular, if a high-priority task T_i attempts to lock an object that is already locked by a low-priority task T_j, then to ensure progress, T_j must be allowed to resume execution. To ensure that requirement (i) is met, T_j's priority must be raised to exceed T_i's before T_j is resumed. However, raising T_j's priority during its object access violates requirement (ii). An object implementation in this task model is *lock-free* (*wait-free*) iff, for any set of tasks with overlapping operation executions, some (each) task in that set completes its operation in a bounded number of its own steps.

The main contribution of this paper is to show that, for the above execution model, any object with consensus number P in Herlihy's hierarchy is universal in a P-processor hard real-time system. That is, consensus number is a function of the number of processors, rather than tasks. An important special case of this result is that, for hard real-time applications on uniprocessors, reads and writes are universal. Thus, Herlihy's hierarchy collapses for such applications. This implies that any object can be implemented in uniprocessor-based hard real-time applications with no kernel support for object sharing, and no strong synchronization primitives. It is worth noting that such applications potentially constitute one of the largest client bases for work on lock-free and wait-free synchronization.

The results mentioned above are obtained by showing how to solve wait-free consensus within our real-time task model. Although consensus protocols can be used directly to implement wait-free and lock-free objects [6, 8], implementations of practical interest are usually based on universal primitives such as `C&S` or `LL/SC`. We show that such primitives can be implemented efficiently on real-time uniprocessors by considering implementations of `C&S`; using `C&S`, `LL/SC` can be implemented in constant time [4]. We give two N-task implementations of `C&S`. The first uses only reads and writes, and has $O(N)$ space and time complexity. The second is based upon a memory-to-memory move instruction (`Move`) and has $O(1)$ time complexity and $O(N)$ space complexity. Although `Move` is uncommon on multiprocessors, it is widely available on uniprocessors.

For a P-processor hard real-time system, our results imply that P-consensus synchronization primitives are universal. However, we know of no commonly-available synchronization primitive that has a consensus number that is greater than two, but not unbounded. Thus, for $P > 2$, primitives with unbounded consensus number are still required in practice. This requirement can be obviated by adopting a timing model that provides a bound on the execution time of any statement. This conclusion follows from work of Alur, Attiya, and Taubenfeld [1, 3], who have shown that consensus objects (and hence any object) can be implemented in a wait-free manner using only reads and writes in such a model. However, as mentioned previously, most practical wait-free and lock-free object implementations are based on universal primitives, not consensus objects. We show that such primitives can be efficiently implemented from reads and writes in hard real-time multiprocessor systems that guarantee bounds on statement execution times.[2]

It is important to note that we consider assumptions about statement execution times only where Herlihy's hierarchy (as it applies to real-time systems) implies that we must (specifically, within the context of real-time multiprocessors that do not provide universal primitives). We have chosen to avoid such assumptions where possible for several reasons. First, accurately estimating time bounds for statements can be difficult. Second, in an *actual* real-time system, where processors are multiprogrammed, a task may be preempted while one of its statements is enabled for execution. As a result, either noninterruptible code fragments must be used, or the assumed time bound must be inflated to account for the period during which the task is preempted. The use of noninterruptible code fragments can lead to priority inversion, the bane of real-time synchronization. Moreover, in many real-time systems, user-level tasks are not allowed to disable interrupts. Inflating time bounds is just as problematic because the execution times of most timing-based algorithms depend on these bounds. To summarize, we feel that assumptions concerning execution times should be made only as a last resort. Our main results do not require such assumptions. They merely require a priority-based task execution model.

The remainder of this paper is organized as follows. In Section 2, we present definitions and notation that will be used in the remainder of the paper. We then present our results for uniprocessors in Section 3, and for multiprocessors in Section 4. We end the paper with concluding remarks in Section 5.

[2] A wait-free mutual exclusion algorithm is presented in [2] that would also seemingly imply this result. However, this algorithm is susceptible to priority inversion in systems that are multiprogrammed, and only guarantees that each task "eventually" reaches its critical section. Thus, it could not be easily applied within a hard real-time system.

2 Definitions

A *task invocation* may have one or more distinct computational phases whose priorities may be different. Successive invocations of a given task execute in the order that they are invoked, i.e., steps of two different invocations of a given task cannot be interleaved.

Notational Conventions: The number of tasks and the number of processors in the system are denoted N and P, respectively. Processors are labeled from 1 to P. $M[i]$ denotes the number of tasks on processor i, and M denotes the maximum number of tasks on any processor. In static-priority scheduling schemes, the tasks on processor i are labeled from 1 to $M[i]$, with higher-priority tasks having higher labels.

Unless stated otherwise, p, q, and r are assumed to be universally quantified over task identifiers. The predicate $p@s$ holds iff statement s is the next statement to be executed by task p. We use $p@S$ as shorthand for $(\exists s : s \in S :: p@s)$, $p.s$ to denote statement s of task p, $p.v$ to denote p's local variable v, and $pr(p)$ to denote the processor on which task p runs. Unless stated otherwise, labeled program fragments are assumed to be atomic. Each such fragment accesses at most one shared variable. If a task p is not accessing a shared object, we consider it to be executing within a *remainder section*, which is denoted by statement $p.0$. The shared variable $L[p]$ contains the priority level of task p. We use $X.Pr()$ to denote an operation performed on object instance X using procedure $Pr()$. $\Box$

In a real-time system, each processor always executes the highest-priority task that is available for execution on that processor (i.e., that has an invocation that has not completed execution). Abstractly, a task's priority is determined by the "environment", and can change dynamically over time. The behavior of the environment is contingent on the underlying scheduling scheme. We model the environment as a separate task called *env*. Note that *env* is an abstraction that models the environment, and not necessarily a true task. The behavior of *env* is formalized by the following axioms.

Axiom 1: *The environment does not modify program variables or program counters.* □

Axiom 2: $(\forall a :: L[p] = a \ \ unless \ \ p@0)$ □

Axiom 3: $p \neq q \ \wedge \ pr(p) = pr(q) \Rightarrow L[p] \neq L[q]$ □

Axiom 2 states that the environment does not change the priority of a task that is accessing a shared object, i.e., for all statements s of the environment task *env*, $\{L[p] = a \ \wedge \ \neg p@0\} \ env.s \ \{L[p] = a \ \vee \ p@0\}$ holds. Axiom 3 states that tasks on the same processor have unique priorities.

Correctness proofs in the full paper take these axioms and the priority-based scheduler into account by using proof rules that are slightly different to those used for fully asynchronous systems.

3 Uniprocessor Systems

In this section, we show that, in hard real-time uniprocessors, reads and writes can be used to solve consensus for any number of tasks. Using consensus objects, any shared object can be implemented in a wait-free manner [6, 8]. However, such implementations usually entail high overhead. More practical wait-free and lock-free implementations are based on primitives such as `C&S` and `LL/SC`. To enable such implementations to be used in real-time uniprocessor systems, we present two implementations of `Read` and `C&S`. (`LL/SC` can be implemented using `Read` and `C&S` in constant time [4].) These implementations use `Read/Write` and `Move`, respectively.

The main difficulty in each of these implementations stems from the fact that, once a task p is enabled to modify a shared variable, it is committed to do so, even if it is first preempted by a higher-priority task that completes an entire operation before p executes again. Thus, there is a risk that p might "corrupt" the values written by completed operations of higher-priority tasks. In each of the implementations in this section, we use a different technique for dealing with this problem. In the first, we ensure that, if task p is committed to writing a variable *Prp* ("propose"), then before it does so, the first (and correct) value written to *Prp* has already been copied to another variable *Fin* ("final"),

```
shared var Fin, Prp : valtype ∪ ⊥
private var v : valtype
initially Fin = ⊥ ∧ Prp = ⊥

procedure local-decide(val : valtype) returns valtype
1 : if Fin = ⊥ then
2 :     if Prp = ⊥ then
3 :         Prp := val
        fi;
4 :     if Fin = ⊥ then
5 :         v := Prp;
6 :         Fin := v
        fi
    fi;
7 : return Fin
```

Figure 2: Consensus using Reads and Writes.

which p cannot subsequently modify. In the second implementation, a task q ensures that the same value is written to all of $2N-1$ shared variables before relinquishing the processor to lower-priority tasks. Thus, even if all $N-1$ of the other tasks subsequently modify one of these variables, the value written by q is still in a majority of the $2N-1$ variables. Finally, in the last implementation of this section, a low-priority task p could be committed to performing a `Move` operation to copy the value of *Prp* to *Fin*. To ensure that preempted tasks do not overwrite the value written to *Fin* by a task q, q copies the value of *Fin* to *Prp* before relinquishing the processor. This ensures that lower-priority tasks copy the same value to *Fin* as q does.

3.1 Universality of Reads and Writes

In Figure 2, procedure *local-decide* solves consensus on a real-time uniprocessor using reads and writes. Objects implemented using this procedure are later referred to as *local-cons-type*. Tasks are at statement 0 (not shown in the figure) when they are not executing *local-decide*. (Recall that statement 0 denotes the remainder section.)

Procedure *local-decide* uses two shared variables, *Prp* and *Fin*. Each task that does not detect the input of another task in *Prp* or *Fin* writes its own value into *Prp*. Having ensured that some value has been proposed (lines 1 to 3), a task copies the proposed value to *Fin*, if necessary (lines 4 to 6); it is easy to see that no task returns before some task's input value is written into *Fin*, and that all tasks return a value read from *Fin*. The following lemma implies that all tasks return the same value, and therefore yields the theorem below.

Lemma 1: *The first value written into Prp is the only value written into Fin.*

Proof: Let p be the first task to write its value $p.val$ into *Prp*. Recall that no higher-priority task is executing when $p.3$ is executed. Thus, if task q executes between $p.3$ and $p.7$, then $q.1$ occurs after $p.3$, which implies that

```
shared var Buf : array [1..2N − 1] of 1..N;
           Val : array [1..N] of valtype;
           Rv : array [1..N] of valtype ∪ ⊥;
           Pm : array [1..N] of boolean; V : valtype
initially (∀k :: Buf[k] = 1) ∧ Val[1] = initial value
private var i : integer; w, maj : 1..N; current : valtype;
            count : array [1..N] of 0..2N − 1

procedure Read() returns valtype
1 :  Pm[p], count := false, (0, ..., 0);
     for i := 1 to 2N − 1 do
2 :     w, count[w] := Buf[i], count[w] + 1
     od;
     maj := select q : (∀r :: count[q] ≥ count[r]);
3 :  current := Val[maj];
4 :  if Pm[p] then
5 :     current := V
     fi; return current
```

Figure 3: Implementation of C&S using Reads and Writes.

```
procedure C&S(old, new) returns boolean
6 :  Pm[p] := false;
7 :  Rv[p], count := ⊥, (0, ..., 0);
     for i := 1 to 2N − 1 do
8 :     w := Buf[i]; count[w] := count[w] + 1
     od;
     maj := select q : (∀r :: count[q] ≥ count[r]);
9 :  current := Val[maj];
10 : if ¬Pm[p] ∧ current = old then
        if old = new then return true fi;
        if maj ≠ p then
11 :        Rv[maj], i := current, 1;
12 :        while ¬Pm[p] ∧ i ≤ 2N − 1 do
13 :           Buf[i], i := maj, i + 1
            od
        fi;
14 :    Val[p], i := new, 1;
15 :    while ¬Pm[p] ∧ i ≤ 2N − 1 do
16 :       Buf[i], i := p, i + 1
        od;
17 :    if i > 2N − 1 ∨ Rv[p] = new then
18 :       V := new;
19 :       for i := 1 to N do Pm[i] := true od;
20 :       return true
        fi
21 : fi; return false
```

Figure 3: (continued)

q does not modify *Prp*. Therefore, $Prp = p.val$ holds continuously between $p.3$ and $p.7$.

Now, consider how some task q can modify *Fin* after $p.3$. If q starts execution after $p.7$, then $q.1$ reads $Fin \neq \perp$, so q does not write *Fin*. If q starts execution before $p.3$ is executed, and $q \neq p$, then by the assumption that $p.3$ writes *Prp* first, $q@\{1..3\}$ holds when $p.3$ is executed. This implies that q has lower priority than p, and therefore does not execute again until after p completes execution. Thus, the test at $q.4$ fails and q does not write *Fin*. Therefore, if task q writes *Fin* after $p.3$, then either $p = q$ or q starts execution between $p.3$ and $p.7$. The latter implies that q has higher priority than p, which implies that q completes execution before $p.7$. In either case, because $Prp = p.val$ holds continuously between $p.3$ and $p.7$, $q.5$ establishes $q.v = p.val$, so $q.6$ writes $p.val$ to *Fin*. $\Box$

Theorem 1: *Consensus can be implemented with constant time and space using reads and writes on a hard real-time uniprocessor system.*

3.2 C&S using Reads and Writes

Our read/write implementation of Read and C&S for N tasks is given in Figure 3. Values of the implemented shared variable are stored in the *Val* array, which contains one element $Val[p]$ for each task p. The following definition states that the current value CV of the implemented shared variable is determined by the task identifier that is stored in a majority of the locations $Buf[1]$ through $Buf[2N-1]$. (It can be shown that such a majority always exists and is unique.)

$$CV \equiv (Val[p] :: p \textit{ is a majority in } Buf)$$

In order to perform a C&S operation, a task p first attempts to determine CV. This is achieved by clearing p's "preempted" flag $Pm[p]$ (line 6), and then reading the entire *Buf* array, counting the task identifiers read (line 8). Then, p attempts to find a majority *maj* among the identifiers read. (Because p does not read *Buf* atomically, it is possible that p does not find a majority, but in any case, p chooses *some* valid task identifier.) p then reads $Val[maj]$ (line 9). It can be shown that if $Pm[p]$ is set between $p.6$ and $p.10$, then the value of CV changed in this interval. In this case, $p.old \neq CV$ holds either before or after the change of CV, so p's C&S can fail at that point. Also, if $Pm[p]$ is *not* set between $p.6$ and $p.10$, then it can be shown that no task modifies *Buf* in this interval. In this case, p correctly determines the majority *maj* and therefore correctly determines CV at line 9. If $p.current$ — the value determined for CV — differs from $p.old$ then p fails immediately. Also, if $p.current = p.old$ and $p.old = p.new$, then p can succeed, because its successful C&S would not modify the implemented shared variable. This leaves C&S operations that determine that $CV = p.old$, and that must attempt to change CV from $p.old$ to $p.new$. Such operations can succeed by first writing $Val[p] := p.new$ (line 14), and by then establishing p as the majority in *Buf* (lines 15 and 16).

A Read operation determines CV (lines 1 to 3) the same way as a C&S operation does (lines 6 to 9). If the Read operation does not determine CV directly, then it can be shown that the value read from V (line 5) is the value of CV at some point during the `Read`.

Several subtle difficulties arise in this implementation. First, it is important to ensure that the majority is not changed by "late" writes to *Buf*. This could potentially occur if a task is preempted while writing *Buf*, and continues writing an "old" value when it resumes running. This possibility is avoided by ensuring that, when a new majority q is established, q is written into all $2N-1$ locations of *Buf* before the next majority is established, and by also ensuring that each of at most $N-1$ lower-priority tasks can write only one "late" value. (Note that this ensures that at least N elements of *Buf* still contain q.) The latter is achieved by having successful `C&S` operations set the *Pm* flags of all tasks (line 19), and having each task check its *Pm* flag before proceeding with each write (lines 12 and 15). Thus, each task can perform at most one "late" write before detecting the overlapping operation and failing. In order to ensure that all $2N-1$ locations of *Buf* are written whenever a new majority is established (the task that established the new majority may have been preempted before finishing its writes to *Buf*), each task writes the majority read to all elements of *Buf* before attempting to change the majority to its own identifier (lines 12 and 13).

As explained above, if p detects that $Pm[p]$ is true before establishing itself as a majority, then p can fail, because it can be shown that an overlapping `C&S` changed the value of CV. However, it is possible for a task p to succeed in making its own identifier the majority, and to be subsequently preempted and therefore fail to detect that it achieved a majority. In this case, the fact that p achieved a majority is "communicated" to p via $Rv[p]$ (line 11). Then, when p executes line 17, it detects that it did achieve a majority, and hence that it succeeded. The correctness proof for this implementation is quite long, and is deferred to the full paper.

Theorem 2: *`Read` and `C&S` can be implemented on a hard real-time uniprocessor system with $O(N)$ time and space complexity.* □

3.3 `C&S` using `Move`

We now present a constant-time implementation of a shared variable X that allows `Read` and `C&S` operations using `Move`. Although `Move` is rare in multiprocessors, it is widely available on uniprocessors. For example, uniprocessor systems based on Intel's 80x86 and Pentium line of processors support the `Move` instruction.

In the implementation shown in Figure 4, a shared variable *Fin* stores the current value of X, along with the identifier of the task that wrote it. The `Read` operation is simply implemented by a read of *Fin.val*. If $old = new$ (line 1), then a `C&S`(old, new) operation by task p succeeds or fails immediately, depending on the value of *Fin.val*. Also, if p detects that $Fin.val \neq old$, its `C&S` can fail immediately (lines 2 and 7). Otherwise, the operation writes its new value into *Prp* (line 8) and, if it does not detect an overlapping successful `C&S` operation (line 9), it attempts to `Move` its new value from *Prp* into *Fin* (line 10), thereby succeeding. It is possible for a task p to successfully `Move` its new value to *Fin* and then get preempted before executing $p.11$. In this case, p cannot detect that it succeeded. To solve this problem, before a task q modifies *Fin*, it first "informs" the task that previously modified *Fin* that it succeeded (lines 5 and 6). The preempted task p can therefore detect by reading $Rv[p]$ (line 12) that it succeeded. To ensure that a low-priority task does not modify *Fin* "late", thereby "corrupting" the value written by a previous `C&S` operation, each `C&S` operation ensures that, if it modifies *Prp* or *Fin*, then $Prp = Fin$ holds before it relinquishes the processor. This ensures that the "late" `Move` operation has no effect on *Fin*. The proof sketch below captures the essence of the proof given in the full paper.

```
type objtype = record val : valtype ; tid : 1..N end
shared var Rv : array [1..N] of valtype ∪ ⊥;
           Fin, Prp : objtype; Run : 1..N
private var oldf : objtype
procedure C&S(old, new : valtype) returns boolean
1 :  if old = new then return Fin.val = old fi;
2 :  if Fin.val ≠ old then return false fi;
3 :  Run := p;
4 :  Rv[p] := ⊥;
5 :  oldf := Fin;
6 :  Rv[oldf.tid] := oldf.val;
7 :  if Fin.val = old then
8 :     Prp := (new, p);
9 :     if Run = p then
10 :       Fin := Prp;
11 :       if Run = p then return true fi;
12 :       if Rv[p] = new then return true fi
        fi;
13 :    Prp := Fin
14 : fi; return false
procedure Read() returns valtype
15 : return Fin.val
```

Figure 4: Implementation of `C&S/Read` using `Move`.

Lemma 2: *If $p@\{4..13\} \wedge Run = p$ holds, then no other task has modified Fin or Prp since p executed $p.3$.*

Lemma 3: *If $p@\{4..13\} \wedge Run \neq p$ holds, then a `C&S` operation changes the value of Fin during p's operation.*

Lemma 4: *If $p.10$ modifies Fin, then $Fin.val = p.old \wedge Prp = (p.new, p)$ holds before $p.10$ and either $Run = p$ holds at $p.11$ or $Rv[p] = p.new$ holds at $p.12$.*

Theorem 3: *`Read` and `C&S` can be implemented on a hard real-time uniprocessor system with constant time* *and* O(N) *space complexity using* `Move`.

Proof: *Fin.val* contains the implemented value. Thus, read operations are correctly linearized. `C&S` operations that modify *Fin* (i.e., change its value) succeed and are linearized to the point at which they update *Fin*. Lemma 4 implies that such operations return *true* and are correctly linearized. It remains to consider `C&S` operations that do not modify *Fin*. It is easy to see that a `C&S` operation is correctly linearized if it returns from line 1 or 2, or from line 14 as a result of failing the test at line 7. By Lemmas 3 and 4, if a `C&S` operation by task p returns from line 14 as a result of failing the test at line 9 or line 11, then it can be correctly linearized before or after the overlapping successful `C&S`, depending on $p.old$. (Note that if p fails the test at $p.12$, then it also fails the test at $p.11$) □

4 Multiprocessor Constructions

In this section, we explore the implications of priority-based scheduling policies and of timing assumptions on the problem of synchronizing tasks in an asynchronous, shared-memory multiprocessor.

In Section 4.1, we present a multiprocessor counterpart to the results of Section 3.1. In particular, we show that any object that can solve consensus for P asynchronous tasks is universal in a P-processor hard real-time system, regardless of the number of tasks. As in the uniprocessor case, this stands in contrast to Herlihy's results for fully-asynchronous systems, in which an object with consensus number P cannot solve consensus for more than P tasks. In this paper, we prove this result only for static-priority real-time systems. However, as explained later, in the full paper we extend this result to apply to dynamic-priority systems.

In Section 4.2, we present two constructions — one for test-and-set (`T&S`)[3] and one for `C&S` — that use `Read` and `Write` operations and delays. Our constructions are based on the assumption that an upper bound Δ is known on the execution time for a short code fragment, and that tasks have the ability to delay execution for at least Δ time units. Such guarantees can be provided by ensuring that a task executing within this code cannot be preempted by other tasks, and by taking the overhead of interrupts into account.

4.1 Universality of P-Consensus

In this subsection, we show that any object that solves consensus for any P tasks is universal in a P-processor, static-priority, real-time system. We achieve this result by showing that, given an object that can solve consensus for all the tasks on processors 1 through i and at most one task on each of processors $i+1$ through P, we can exploit the priority-based scheduling on processor $i+1$ in order to solve consensus for this set of tasks *and* all tasks on processor $i+1$. Using this, we inductively show that the tasks of each processor can be "added" to the set of tasks for which we can solve consensus, until all tasks in the system are included. We use the following definition to formalize this induction.

Definition: *An object T* implements i-decide in a P-processor real-time system *if T solves consensus for any set of tasks S satisfying* $|\{p \in S :: pr(p) > i\}| \leq P - i$. □

We prove that any object with consensus number P is universal in a P-processor real-time system in three steps. First, we show that such an object implements 0-decide. We then inductively show that, given an object that implements 0-decide, we can implement P-decide. Finally, we show that any object that implements P-decide can be used to implement consensus for any number of tasks in a P-processor real-time system. The first and third steps are quite straightforward, so we dispense with them first in the following two lemmas. Both lemmas are easily proved using the fact that all tasks run on processors 1 through P. In particular, the definition of an object with consensus number P solving 0-decide requires that that object solves consensus for any set of at most P tasks, which it does by definition; and an object that solves P-decide solves consensus for any set of tasks that does not contain tasks that run on processors greater than P. Any set of tasks in a P-processor system satisfies this condition.

Lemma 5: *Any object with consensus number P implements 0-decide in a P-processor real-time system.* □

Lemma 6: *In a P-processor hard real-time system, any object that implements P-decide is universal.* □

We now return to the second step of the proof — showing that P-decide can be implemented using 0-decide. We obtain this result inductively by showing that i-decide objects can be used to implement $(i+1)$-decide objects. In particular, the algorithm presented in Figure 5 implements $(i+1)$-decide using i-decide objects. Due to space limitations, we defer the formal proof of this algorithm to the full paper, and give a brief and informal explanation of it here.

Procedure $(i+1)$-*decide* uses an array A of $M[i+1]$ i-decide objects. (Recall that $M[i+1]$ is the number of tasks executing on processor $i+1$.) As explained below, for each $A[j]$, the set S_j of tasks that accesses $A[j]$ can be shown to satisfy $|\{p \in S_j :: pr(p) > i\}| \leq P - i$. Therefore, because each $A[j]$ is assumed to implement

[3]We assume that X.`T&S` returns *true* if it succeeds in changing X from *false* to *true*, and returns *false* otherwise.

```
shared var A : array [1..M[i+1]] of i-decide-type;
           X : local-cons-type (refer to Figure 2)

procedure (i+1)-decide(val : valtype) returns valtype
var v : valtype
1 : if pr(p) = i+1 then
2 :     if X.Prp ≠ ⊥ then
3 :         v := X.local-decide(val); return v
        fi;
4 :     v := A[p].i-decide(val);
5 :     v := X.local-decide(v)
    else
        v := val;
6 :     for j := 1 to M[i+1] do
7 :         v := A[j].i-decide(v)
        od
    fi; return v
```

Figure 5: Inductive step for proof of Theorem 4.

i-decide, each $A[j]$ solves consensus for the tasks that access it.

Procedure $(i+1)$-*decide* also uses a local consensus object X, which is accessed only by tasks on processor $i+1$. X is assumed to be implemented using the code in Figure 2. This allows a task to detect whether the value for X has been determined (line 2 in Figure 5).

Suppose the $(i+1)$-*decide* procedure is accessed by a set of tasks S satisfying $|\{p \in S :: pr(p) > i+1\}| \leq P - i - 1$. As illustrated in Figure 6, each task p on processor $i+1$ accesses $A[p]$ and tasks on other processors access all $A[j]$'s in order. To see that each S_j satisfies $|\{p \in S_j :: pr(p) > i\}| \leq P - i$, first observe that a task p accesses $A[j]$ only if $pr(p) \neq i+1$ (line 7) or if $pr(p) = i+1$ and $p = j$ (line 4). Thus, for any task p, $p \in S_j$ and $pr(p) > i$ implies either that $pr(p) = i+1$ and $p = j$ holds or that $pr(p) > i+1$ holds. By the definition of S, at most $P-i-1$ tasks in S satisfy the second condition. Also, at most one task on processor $i+1$ accesses $A[j]$.

We now explain why procedure $(i+1)$-*decide* solves consensus for S. First, observe that each task returns the value decided by X or by $A[M[i+1]]$. It is easy to see that both values are the input value of some task. It remains to show that the two values are equal.

As explained in Section 3, the first value written to $X.Prp$ is the value decided by X. Observe that no task calls $X.local\text{-}decide$ at line 3 before the first time $X.Prp$ is written. Thus, the first value written to $X.Prp$ is as the result of a call by some task p to $X.local\text{-}decide$ at line 5, and the value written is the value returned by $A[p]$. In the full paper, we prove the following lemma, which implies that this value is the same as that returned by $A[M[i+1]]$. The proof hinges upon q's priority being higher than p's (because $q > p$). This implies that either q reads $X.Prp \neq \perp$, and hence does not access $A[q]$, or some task writes $X.Prp$ before p does.

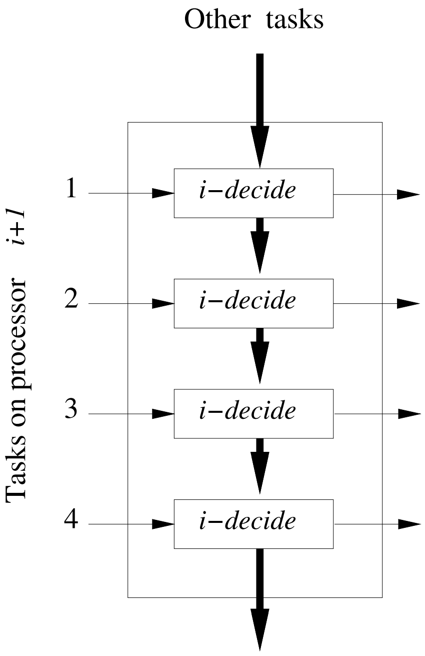

Figure 6: Arrangement of tasks executing procedure $(i+1)$-*decide*. Thick arrows represent sets of tasks; thin arrows represent single tasks.

Lemma 7: *If task p writes $X.Prp$ first, then no task q on processor $i+1$ such that $q > p$ accesses $A[q]$.* □

Lemma 7 implies that the value decided by $A[M[i+1]]$ is the same as that decided by $A[p]$, because, in statements 7 and 8, tasks on processors other than $i+1$ "propagate" the same value from $A[p]$ through each $A[j]$ for $j > p$. This concludes the proof sketch for the algorithm in Figure 5. A straightforward induction using this algorithm together with Lemmas 5 and 6 yields the following result. (Recall that M is the maximum number of tasks on any processor.)

Theorem 4: *In a P-processor, static-priority, hard real-time system, any object with consensus number P can solve consensus for any number of tasks with time complexity $O(M^{P-1})$ and space complexity $O(M^P)$.* □

In the full paper, we show that with minimal time and space overhead, this result can be extended to hold even in dynamic-priority systems. This is achieved by having tasks choose "names" in advance in such a way that the order of names reflects the order of the priorities of tasks that concurrently access P-decide. These names, instead of task identifiers, are then used to determine which task accesses which $A[j]$ at line 5. The names are chosen using a simple renaming algorithm, which uses an array of M local consensus objects for each processor.

While the algorithm in Figure 5 does not yield a practical consensus implementation in general, it is interesting to note that if $P = 2$, then Theorem 4 implies that, given any object with consensus number 2 (e.g., `T&S`),

shared var $Z : \mathit{valtype} \cup \perp$
private var $v : \mathit{valtype} \cup \perp$

procedure T&S() **returns boolean**

1 : $v := Z$;
2 : **if** $v = \perp$ **then** $Z := p$ **fi**;
3 : $\mathit{delay}(\Delta_{1..2})$;
4 : **return** $(v = \perp \wedge Z = p)$

procedure Reset()

5 : $Z := \perp$

Figure 7: Implementing T&S and Reset with delays.

we can solve consensus for any number of tasks in $O(M)$ time with $O(M^2)$ space. Results of Jayanti and Toueg imply that, given universal consensus objects, C&S can be implemented with time complexity $O(N^2)$ and space complexity $O(N^3)$ [8]. Their construction uses $O(N^3)$ consensus objects, and performs at most $O(N)$ consensus object accesses per operation. Thus, Theorem 4 and the results of [8] yield the following corollary.

Corollary 1: *On a two-processor, N-task, hard real-time system that provides an object with consensus number two,* C&S *can be implemented with time complexity $O(N^2)$ and space complexity $O(N^3M^2)$.* □

4.2 Delay-Based Implementations

Many multiprocessors provide objects, such as T&S, that have consensus number two. Corollary 1 above implies that existing universal constructions can be used in such systems to implement any shared object for any number of tasks running on two processors. For more than two processors, however, the situation is not as encouraging because few existing multiprocessors provide objects that have consensus number greater than two, unless they provide universal primitives, in which case existing universal constructions can be applied directly. Thus, on most multiprocessing systems with three or more processors that do not provide universal primitives, Herlihy's results about fully-asynchronous systems imply that general object constructions are not possible [6]. We now show that, by making certain timing assumptions, T&S and C&S can be efficiently implemented in such systems. While these results have been previously established in principle [1, 2, 3], our implementations are significantly more efficient and are better suited to the requirements of real-time systems.

Our delay-based implementation of T&S and Reset,[4] which is based on a delay-based mutual exclusion algorithm due to Fischer [9], is given in Figure 7. A delay-based implementation of T&S and Reset has been presented previously by Alur and Taubenfeld [2]. Their implementation is designed to be fast under low contention. This comes at the expense of somewhat high time complexity in the worst case. In hard real-time systems, optimizing time complexity for the case of low contention is of questionable importance because in such systems, tasks must be scheduled assuming worst-case time bounds. In other words, low time complexity in the absence of contention is of little benefit if time complexity under contention is high. The running time of the T&S and Reset implementation presented in [2] is bounded by 17Δ, where Δ is an upper bound on the time taken by any task to take a step. In comparison, the worst-case running time of our implementation is 5Δ (assuming each line in Figure 7 to be a step). In our implementation, we assume that $\Delta_{1..2}$ is an upper bound on the time taken by any task to execute lines 1 and 2. In practice, this requires that statements 1 and 2 are implemented as a nonpreemptible code fragment. We further assume that executing $\mathit{delay}(\Delta_{1..2})$ always takes at least $\Delta_{1..2}$ time units. We defer a correctness proof of this algorithm to the full paper.

[4] We actually implement a slightly restricted version of T&S and Reset in which a task may invoke the Reset operation only if it is has previously performed a successful T&S operation, and has not performed a Reset operation since.

shared var $X : \mathit{valtype}$; Flag : T&S-*type*
local var ret : **boolean**
procedure C&S($\mathit{old}, \mathit{new} : \mathit{valtype}$) **returns boolean**

1 : **if** $\mathit{old} = \mathit{new}$ **then return** $(X = \mathit{old})$ **fi**;
2 : $\mathit{delay}(\Delta_{3..7})$;
3 : **if** $X \neq \mathit{old}$ **then return** *false* **fi**;
4 : **if** *Flag*.T&S() **then**
5 : $\mathit{ret} := (X = \mathit{old})$;
6 : **if** *ret* **then** $X := \mathit{new}$ **fi**;
7 : *Flag*.Reset(); **return** *ret*
fi;
8 : $\mathit{delay}(\Delta_{3..7})$; **return** *false*

Figure 8: Implementation of C&S using reads, writes, and delays. Read is implemented by reading X.

Our implementation of Read and C&S using Read, Write, T&S, and delays is shown in Figure 8. A before, we assume an upper bound $\Delta_{3..7}$ on the time taken to execute lines 3 through 7.

This implementation uses a register X, which stores the current value of the implemented shared variable, as well as a boolean *Flag*. A Read operation simply reads X. To perform a C&S(v, w) operation, task p first determines whether it needs to change the value of X (line 1). If not, the operation can immediately succeed or fail, depending on the value of X. Otherwise, p executes a delay of at least $\Delta_{3..7}$ time units, and if $X = v$ still holds, p attempts to modify X by locking *Flag* (using T&S), and then changing X if necessary, before releasing the lock. If a task p fails to acquire the lock, it executes a delay of at least $\Delta_{3..7}$ time units, and then fails.

It is easy to see that Read operations are correctly linearized. A successful C&S(v, w) operation is linearized at the point at which it executes line 1 or line 6. After the linearization point of any successful C&S(v, w)

(including those that succeed at line 1), $X = w$ holds. Thus, to show that each successful C&S(v, w) operation is correctly linearized, it suffices to show that immediately before the linearization point $X = v$ holds. This is clearly true for a C&S(v, w) that succeeds at line 1. If a C&S(v, w) by task p returns *true* from line 7, then p previously locked *Flag*, checked that $X = v$ and then assigned $X = w$. Because no task modifies X without locking *Flag*, $X = v$ holds immediately before p assigns $X := w$. Thus, p's operation is correctly linearized.

To show that a failing C&S(v, w) operation by task p is correctly linearized, it suffices to show that, at some point during the operation, $X \neq v$ holds. Thus, if p returns *false* from line 1, 3, or 7, then p fails correctly. If p returns *false* from line 8, then by the following lemma, p's operation can be correctly linearized (either before or after X changes, depending on the value of v).

Lemma 8: *If a* C&S(v, w) *operation by task* p *reaches line 8, then* X *is modified during* p*'s operation.*

Theorem 5: Read *and* C&S *can be implemented with constant time and space using only* Read, Write, *and delays on any hard real-time multiprocessor system that can provide a delay that is at least as long as the longest time taken to execute a small code fragment.* □

5 Concluding Remarks

Our results leave many open questions regarding lower bounds and algorithms in our real-time task model. In this paper, we have presented several implementations that are impossible in general asynchronous models. It seems likely that many other problems that are impossible (difficult) in asynchronous systems might be possible (easier) in real-time systems. In the full paper, we give solutions to two such problems for real-time uniprocessors. The first is a linear-time, bounded-space, wait-free snapshot algorithm. No such algorithm is known for truly asynchronous systems. The second is an N-task renaming algorithm based on reads and writes that uses only N names, which is impossible for truly asynchronous systems.

Acknowledgement: We are grateful to Gadi Taubenfeld for his comments on an earlier draft of this paper.

References

[1] R. Alur and G. Taubenfeld, "Results about Fast Mutual Exclusion", *Proceedings of the 13th IEEE Real-Time Systems Symposium*, 1992, pp. 12-21.

[2] R. Alur and G. Taubenfeld, "How to Share an Object: A Fast Timing-Based Solution", *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, 1993, pp. 470-477.

[3] R. Alur, H. Attiya, and G. Taubenfeld, "Time-Adaptive Algorithms for Synchronization", *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, May 1994, pp. 800-809.

[4] J. Anderson and M. Moir, "Universal Constructions for Multi-Object Operations", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 184-193.

[5] J. Anderson, S. Ramamurthy, and K. Jeffay, "Real-Time Computing with Lock-Free Shared Objects", *Proceedings of the 16th IEEE Real-Time Systems Symposium*, 1995, pp. 28-37.

[6] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.

[7] M. G. Harbour, M. H. Klein, and J. P. Lehoczky, "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority", *Proceedings of the 12th IEEE Real-Time Systems Symposium*, 1991, pp. 116-128.

[8] P. Jayanti and S. Toueg, "Some Results on the Impossibility, Universality, and Decidability of Consensus", *Proceedings of the 6th International Workshop on Distributed Algorithms*, Springer-Verlag, Nov. 1992, pp. 69-84.

[9] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February, 1987, pp. 1-11.

[10] J.Y.T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks", *Performance Evaluation*, Vol. 2, No. 4, 1982, pp. 237-250.

[11] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, Vol. 30, No. 1, Jan. 1973, pp. 46-61.

[12] A. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT Laboratory for Computer Science, 1983.

[13] Raghunathan Rajkumar, *Synchronization In Real-Time Systems — A Priority Inheritance Approach*, Kluwer Academic Publications, 1991.

[14] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time System Synchronization", *IEEE Transactions on Computers*, Vol. 39, No. 9, 1990, pp. 1175-1185.