Wait-free Object-Sharing Schemes for Real-Time Uniprocessors and Multiprocessors*

James H. Anderson, Rohit Jain, and Srikanth Ramamurthy
Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175

Abstract

Several new wait-free object-sharing schemes for real-time uniprocessors and multiprocessors are presented. These schemes have characteristics in common with the priority inheritance and priority ceiling protocols, but are nonblocking and implemented at the user level. In total, six new object-sharing schemes are proposed: two for uniprocessors and four for multiprocessors. Breakdown utilization experiments are presented that show that the multiprocessor schemes entail less overhead than lock-based schemes.

1. Introduction

In a *wait-free* object implementation [9], operations must be implemented using bounded, sequential code fragments, with no blocking synchronization constructs. In real-time applications, wait-free object implementations are an attractive alternative to more conventional lock-based object-sharing schemes, because wait-free implementations avoid deadlock and priority inversion with no underlying kernel support. However, previous wait-free implementations designed for asynchronous concurrent systems entail overhead that can be excessive in certain applications. The main contribution of this paper is to show that, in real-time systems, efficient wait-free implementations *are* possible. We accomplish this by presenting wait-free implementations that reduce overhead by exploiting the way tasks are scheduled for execution in real-time systems.

We present wait-free implementations both for uniprocessors and shared-memory multiprocessors. Our uniprocessor implementations are surprisingly similar to the priority inheritance protocol (PIP) and the priority ceiling protocol (PCP) [11, 13]. However, our implementations are nonblocking and are implemented entirely at the user level.

*Work supported by NSF grants CCR 9216421 and CCR 9510156, 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.

Wait-free objects are often implemented by means of a *helping scheme*. Before beginning an operation, a task first "announces" its intentions by storing information about its operation in a shared "announce variable". While attempting to perform its own operation, a task also attempts to "help" other tasks with announced operations by performing their operations in addition to its own. Care must be taken to ensure that each operation is executed *exactly* once, and that a helped task can retrieve its return values from memory. This is usually accomplished by using strong synchronization primitives like compare-and-swap (`CAS`) when updating shared data so that tasks do not adversely interfere with each other. In a sense, a helping-based wait-free scheme is a "pessimistic" notion of nonblocking user-level synchronization that is similar to a lock-based scheme, with helping taking the place of blocking. (This can be seen in Figure 2, which is considered below.)[1]

The object-sharing schemes considered in this paper are summarized in Figure 1. In total, six new object-sharing schemes are proposed herein: two for uniprocessors and four for multiprocessors. The uniprocessor schemes listed in Figure 1 are based on a novel helping technique called *incremental helping* [5]. Two such schemes are presented, *IHI* (*incremental helping with inheritance*) and *IHC* (*incremental helping with ceilings*). Incremental helping exploits the priority structure that exists on real-time systems. The general idea of incremental helping, as applied within the IHI scheme, is illustrated in Figure 2. Before beginning an operation on some object, a task must first announce its intentions by updating a shared announce variable associated with that object. Before a task is allowed to do this, however, it must first help any previously-announced operation on that object to complete execution. This scheme requires only one announce variable to implement an object. In contrast, helping schemes for asynchronous systems typically require one announce variable per task [1, 2, 8]. In addition,

[1] The first and third author and colleagues have done much recent work on the use of *lock-free* shared objects in real-time systems [3, 4, 6]. Lock-free and wait-free objects are related, *but different*, notions. As explained in [3, 4, 6], lock-free algorithms constitute an *optimistic* notion of nonblocking user-level synchronization.

Uniprocessor Schemes	**IHI:** Incremental helping with inheritance. **IHC:** Incremental helping with ceilings.
Multiprocessor Schemes	**CH1:** Cyclic helping with one helping ring. **CHN:** Cyclic helping with N helping rings for N objects. **PH1:** Priority helping with one helping ring. **PHN:** Priority helping with N helping rings for N objects.

Figure 1. Wait-free object-sharing schemes considered in this paper.

with incremental helping, each task helps at most one other task, while in helping schemes for asynchronous systems, each task helps all other tasks in the worst case.

In the IHI scheme, the worst-case time to perform an operation is $2 \cdot T$, where T is the execution time of one operation in the absence of preemption. Of the $2 \cdot T$ worst-case execution cost, a factor of T represents the overhead associated with helping. This overhead term is similar to blocking factors that arise in scheduling conditions when using the PIP [11, 13]. The IHC scheme is similar to the IHI scheme, except that priority ceiling information [11, 13] is used when incremental helping is performed. When a task announces an operation on some object, the priority ceiling of that object is recorded in the announce variable. A task helps an announced operation of a lower-priority task only if its priority is lower than the currently-announced priority ceiling. The IHC gives rise to task executions that are very similar to executions that arise with the PCP [11, 13], with helping taking the place of blocking.

The four multiprocessor schemes that we present are obtained by combining incremental helping with two mechanisms for dealing with conflicts across processors. These mechanisms are called *cyclic helping* and *priority helping*, respectively. With cyclic helping, the processors are thought of as if they were part of a logical ring. Tasks are helped through the use of a "help counter", which cycles around the ring. To advance the help counter from processor q to the next processor on the ring, a task must first help the currently-announced task on processor q. If the ring consists of Q processors, then a task may have to help Q other tasks: a lower-priority task on its own processor, and one task on each other processor on the ring. Thus, the worst-case time to perform an operation is at most $T + QT$, where T is the time required for the longest operation. (Tighter worst-case bounds are given in later sections.)

Priority helping is similar to cyclic helping except that, instead of advancing around a logical ring, the help counter is always advanced to the processor with the highest-priority pending operation. Under priority helping, if an operation is of highest priority, then at most one other operation can be completed before it. In all, we consider four multiprocessor schemes, denoted as follows: *CH1* (*cyclic helping with one helping ring for all objects*), *CHN* (*cyclic helping with N helping rings for N objects*), *PH1* (*priority helping with one helping ring for all objects*), and *PHN* (*priority helping with N helping rings for N objects*).

At a high level, the uniprocessor wait-free schemes proposed in this paper are similar to lock-based schemes proposed previously. However, a closer examination reveals some important differences. For example, in implementations of the PIP and PCP schemes, extra overhead is required to perform kernel calls and to maintain various kernel data structures. In addition, some implementations of these protocols are sensitive to context switching costs. In contrast, with the IHI and IHC schemes, extra overhead is required to perform helping. This overhead is partially the result of having to rely on using strong primitives such as `CAS` to avoid interferences while helping. In addition, a performance penalty is paid whenever a given operation is partially helped by several tasks, each of which repeats some steps performed by the others. For many objects, helping overhead can be minimized by careful algorithmic design, but it cannot be entirely eliminated. Whether the IHI or IHC scheme will outperform its lock-based counterpart is dependent on various system parameters (e.g., the cost of invoking synchronization primitives, the cost of performing a kernel call, etc.) and on the particular object being implemented.

While we believe that our uniprocessor schemes are important in their own right, our main interest in them derives from the fact that they are stepping stones towards our multiprocessor implementations. The differences between our multiprocessor schemes and previously-proposed multiprocessor protocols [10, 11, 12, 14] are very striking. Previous multiprocessor schemes give rise to worst-case blocking overheads that are so high as to often preclude practical implementations. This is because of difficulties associated with priority inversions involving tasks executing on different processors. In contrast, wait-free schemes are not susceptible to priority inversions, but incur overhead associated with helping. Simulation results presented herein show that the helping overhead in our schemes has much less of a negative impact on schedulability than blocking factors due to priority inversions in lock-based schemes.

The rest of this paper is organized as follows. In Section 2, we present our uniprocessor schemes, and in Section 3, our multiprocessor schemes. Wait-free algorithms have the

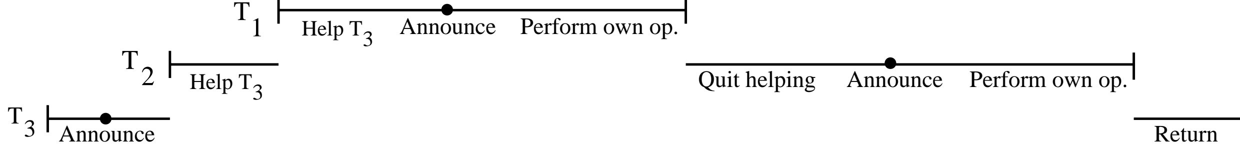

Figure 2. Task T_3 detects no previously-announced task, so it announces its operation. Before T_3 can complete its operation, it is preempted by task T_2. Task T_2 begins to help T_3 to complete its operation, but before it finishes, it is preempted by task T_1. Task T_1 detects that T_3's operation has been announced but is not finished, so it too helps T_3. It then announces its own operation, executes it, and relinquishes the processor to task T_2. Task T_2 detects that T_3's operation is complete, so it announces its own operation, executes it, and relinquishes the processor to T_3. Task T_3 detects that its operation has been completed, so it returns. Note that the overall execution is similar to what one might find in a lock-based system, with helping taking the place of blocking.

reputation of being difficult to design correctly. However, we show that the proposed wait-free schemes can be easily applied to implement different objects by taking near-sequential code for implementing the required operations and "plugging" it into a generic implementation. In addition to algorithms for implementing objects, we also present scheduling conditions that can be applied when tasks share objects using these algorithms. After presenting our various object-sharing schemes, we present results from breakdown utilization experiments in Section 4 that compare our multiprocessor schemes to a multiprocessor variant of the PCP. We end the paper with concluding remarks in Section 5.

2. Uniprocessor Schemes

In this section, we describe the IHI and IHC schemes in detail. We first explain how to implement objects under these schemes by presenting a general framework that can be used to convert virtually any sequential object implementation into a concurrent, wait-free one. The existence of such a framework shows that wait-free objects can be implemented under either scheme without undue effort. We then present scheduling conditions that can be applied to task sets in which objects are shared using the proposed wait-free implementations.

All of the implementations we present make use of a special synchronization primitive called *conditional compare-and-swap* (`CCAS`).[2] This primitive is defined as follows.

`CCAS`(V: **ptr to** *vertype*; ver: *vertype*; X: **ptr to** *valtype*;
old, new : $valtype$) **returns boolean**
⟨ **if** $*V \neq ver \lor *X \neq old$ **then return** *false* **fi**;
$*X := new$;
return *true* ⟩

The angle brackets above indicate that `CCAS` is atomic. As its definition shows, `CCAS` is a restriction of the more well-known two-word compare-and-swap (`CAS2`) instruction in which one word is a compare-only value. In our multiprocessor implementations, this value is a version number that is incremented after each wait-free operation. Such a version number is assumed to not cycle during any single wait-free operation. `CCAS` is useful because the compare-only value can be used to ensure that a "late" `CCAS` operation by a task that has been preempted and then resumed has no effect. Fortunately, `CCAS` is easy to implement even if `CAS2` is not available. In a recent paper [5], we showed that `CCAS` can be implemented in only three high-level language statements if `CAS` (a commonly-available instruction) is available.

[2] Actually, the uniprocessor implementations can be defined using only `CAS`, but this slightly complicates their explanation.

2.1. A Framework for Implementing Objects

The general framework just mentioned is presented in this subsection. In presenting this framework, we focus on the IHC scheme. A similar framework for the IHI scheme can be obtained with just a few modifications, which we describe below. The framework consists of a set of procedures that implement the required helping scheme and a set of procedures that implement the operations of the implemented object. We illustrate the latter by presenting procedures that implement operations on a linked list.

In order to convert the sequential code that implements a given operation into wait-free code, we require that the sequential code can be broken into phases that are idempotent. If a phase is *idempotent*, then executing it more than once has the same effect as executing it exactly once. Under the helping schemes considered in this paper, several tasks may attempt to perform a phase of some operation. Idempotency ensures that tasks do not adversely interfere with each other if this happens. Idempotency can be ensured by means of the following two-step process.

- **Step 1:** Break the sequential code that implements each operation into phases such that no variable that is written in a phase is also read in that phase. This may

require slight modifications to the sequential code. For example, an assignment like "$X := f(X)$", where X is shared, would have to be broken into two phases "$Y := f(X)$" and "$X := Y$", which requires the introduction of a new shared variable Y.

- **Step 2:** Implement each update of a shared variable by using CCAS. This ensures that "late" updates have no effect.

We now illustrate this two-step process by showing how to implement linked lists under the IHC scheme. Our list implementation is shown in Figure 3. In the following paragraphs, we first explain how this implementation works and then consider some modifications that could be made to it.

We begin by describing the shared variables that are used. *First* and *Last* are sentinel nodes at the beginning and the end of the list. *Par*$[i]$ is used to record the parameters of a list operation by task T_i. In our list implementation, values of type *List_Rec* are cast to elements of *Par*. A value of this type consists of the following fields: *pred*, *node*, *copy*, *key*, *rv*, and *subop*. These fields are used to record information about an operation as it is executed. *Par*$[i]$*−>node* is used to store the address of a node to be inserted or deleted. *Par*$[i]$*−>subop*, which is an array of pointers to functions, is used to record the code sequences to be executed during the various phases of operations of T_i. The other fields of *Par*$[i]$ are described below.

Two other shared variables are used in the implementation: *Phase*, which records the "current" phase of each task, and *Ann*, which is used to "announce" an operation to be helped. The announce information includes the index of the task to help (or 0, if no task needs help) and the priority ceiling of that task's pending operation. In this paper, we mainly focus our attention on single-object operations. However, as discussed in Section 5, the algorithms that we present can be used to implement multi-object operations. If all operations are single-object operations, then the *priority ceiling* of an operation on an object is simply the priority of the highest-priority task that accesses that object [13]. In the IHC scheme, one *Ann* variable is used for all implemented objects. This is because, when priority ceiling information is used, a task accessing an object may have to help another lower-priority task accessing a different object. (This is similar to the PCP, where a task may have to block when accessing an object until a lower-priority task finishes accessing a different object.)

We now describe the various procedures used in the implementation. Procedures *Insert*, *Delete*, and *Search* are invoked to perform list operations. Each first calls *Initialize* to perform some object-specific initialization, and then does some operation-specific initialization that includes recording the suboperations to be carried out in the phases of that operation. Next, *Access_Obj* is called.

Access_Obj is the main "driver routine" for implementing the helping scheme. To perform an operation, a task T_i must first announce its intentions by updating the *Ann* variable. However, it is only allowed do so if its priority is higher than the priority ceiling of any currently-announced operation. If T_i's priority is higher, then it places its task identifier and the priority ceiling of its operation in *Ann* and starts executing its operation (lines 3 and 4). Otherwise, T_i must first help the currently-announced operation (lines 5-7). Note that the currently-announced operation must be of a lower-priority task. Because we are assuming a priority-based task model, T_i knows that such a lower-priority task will take no steps until T_i resumes execution. Such knowledge is crucial for the correctness of the algorithm. After completing its own operation, T_i restores the original value it read from *Ann* (line 9). This ensures that a preempted operation may resume.

In the remainder of this description, we let T_i denote a task that attempts to help an operation of some task T_h (note that i and h could be the same task). T_i helps T_h's operation by calling *Help*(h). In *Help*, task T_i first reads the current phase of T_h's operation from *Phase*$[h]$ and then executes the corresponding suboperation *Par*$[h]$*−>subop*$[ph](h, ph)$ (lines 12 and 13). *Phase*$[h]$ = *DONE* signifies that T_h's operation is complete.

The first phase of each list operation is a scan of the list, which is performed by invoking *Findpos*. The scan attempts to locate the predecessor of the first node in the list whose key is at least as large as *Par*$[h]$*−>key*, which stores the key being searched for, inserted, or deleted (lines 17-21). The address of the predecessor is recorded in *Par*$[h]$*−>pred*. After invoking *Findpos*, one or more additional suboperations are performed, depending on the operation.

The case of search is simple: *Par*$[h]$*−>rv* is updated to indicate whether the node searched for was found in the list (lines 41 and 42). This suboperation is performed by invoking *Sch*.

In the case of insert, the suboperation after *Findpos* is *Ins*. *Ins* is almost like the sequential code for inserting a node in a linked list. If the key to insert is not already in the list, then the new node is linked in by first changing its *next* field (line 57) and by then changing the *next* field of its new predecessor (line 58). Note that if a higher-priority task preempts T_i inside the *Ins* procedure and helps T_h, then when T_i resumes it won't be able to corrupt the list. This is because T_h's operation will be in the *DONE* phase, and hence T_i will fail the CCAS operations at lines 57 and 58.

Deleting the node following some node *pred* involves performing the assignment "*pred−>next* := *pred−>next−>next*". Such an assignment violates idempotency, so it must be executed as two separate phases, as discussed earlier. These two phases are implemented by the procedures *Del_1* and *Del_2*, respectively. In *Del_1*, a pointer to the deleted node is returned in *Par*$[h]$*−>node*

```
type nodetype = record key: keytype; val: valtype; next: ptr to nodetype end;
     anntype = record tid: 0..N; ceiling: 0..N end;
     phtype = 0..M ∪ {DONE};
     List_Rec = record pred, node, copy: ptr to nodetype; key: keytype; rv: 0..2; subop: ptr to function end

shared variable
     First, Last: nodetype initially First = (−∞, 0, &Last) ∧ Last = (∞, 0, NIL);      /* List's sentinel nodes */
     Par: array[1..N] of ptr to void;                              /* Par[i] stores parameters to task T_i's operation */
     Phase: array[1..N] of phtype;                                 /* Stores each task's "current" phase */
     Ann: anntype initially (0, 0)      /* Variable in which current operation is announced; Ann.tid = 0 when no operation is pending */

private variable                                                   /* For task T_i, where 1 ≤ i ≤ N */
     tid: 0..N;  new, pred, nextp, nextnextp: ptr to nodetype;  key, nextkey: keytype;  ph, nextph: phtype
```

```
procedure Access_Obj()
1:  hlp := Ann;
2:  if hlp.tid = 0 ∨ i < hlp.ceiling then
3:      Ann := (i, ceiling);
4:      Help(i)
    else
5:      Help(hlp.tid);
6:      Ann := (i, ceiling);
7:      Help(i);
8:      hlp := (0, 0)
    fi;
9:  Ann := hlp
```

```
procedure Help(tid: 1..N)
10: ph := Phase[tid];
11: while ph ≠ DONE do
12:     nextph := Par[tid]−>subop[ph](tid, ph);
13:     CAS(&Phase[tid], ph, nextph);
14:     ph := Phase[tid]
    od
```

```
procedure Findpos(tid: 1..N; ph: phtype)
        returns phtype
15: key := Par[tid]−>key;
16: nextp := &First;
17: repeat
18:     pred := nextp;
19:     nextp := pred−>nextp;
20:     nextkey := nextp−>key
21: until nextkey ≥ key ∨ nextp = &Last;
22: CCAS(&Phase[tid], ph,
        &Par[tid]−>pred, NIL, pred);
23: return 1
```

```
procedure Initialize(tid: 1..N)
24: Par[tid] := (List_Rec*)alloc_list_rec();
25: Par[tid]−>pred := NIL;
26: Par[tid]−>node := NIL;
27: Par[tid]−>copy := NIL;
28: Par[tid]−>subop := NIL;
29: Par[tid]−>rv := 0;
20: Phase[tid] := 0
```

```
procedure Search(key: keytype) returns phtype
31: Initialize(i);
32: Par[tid]−>subop[0] := Findpos;
33: Par[tid]−>subop[1] := Sch;
34: Par[tid]−>key := key;
35: Access_Obj();
36: return(Par[tid]−>rv)
```

```
procedure Sch(tid: 1..N; ph: phtype)
        returns phtype
37: pred := Par[tid]−>pred;
38: key := Par[tid]−>key;
39: nextkey := pred−>next−>key;
40: if key = nextkey then
41:     CCAS(&Phase[tid], ph,
            &(Par[tid]−>rv), 0, 1)
42: else CCAS(&Phase[tid], ph,
            &(Par[tid]−>rv), 0, 2)
    fi;
43: return DONE
```

```
procedure Insert(key: keytype; val: valtype)
44: new := nodealloc();
45: *new := (key, val, NIL) ;
46: Initialize(i);
47: Par[tid]−>node := new;
48: Par[tid]−>subop[0] := Findpos;
49: Par[tid]−>subop[1] := Ins;
50: Par[tid]−>key := key;
51: Access_Obj()
```

```
procedure Ins(tid: 1..N; ph: phtype) returns phtype
52: pred := Par[tid]−>pred;
53: nextp := pred−>next;
54: new := Par[tid]−>node;
55: nextkey := nextp−>key;
56: if nextkey ≠ Par[tid]−>key then
57:     CCAS(&Phase[tid], ph, &(new−>next), NIL, nextp);
58:     CCAS(&Phase[tid], ph, &(pred−>next), nextp, new)
    fi;
59: return DONE
```

```
procedure Delete(key: keytype)
60: Initialize(i);
61: Par[tid]−>subop[0] := Findpos;
62: Par[tid]−>subop[1] := Del_1;
63: Par[tid]−>subop[2] := Del_2;
64: Par[tid]−>key := key;
65: Access_Obj()
```

```
procedure Del_1(tid: 1..N; ph: phtype) returns phtype
66: pred := Par[tid]−>pred;
67: key := Par[tid]−>key;
68: nextp := pred−>next;
69: nextkey := nextp−>key;
70: nextnextp := nextp−>next;
71: if nextkey = key then
72:     CCAS(&Phase[tid], ph, &Par[tid]−>node, NIL, nextp);
73:     CCAS(&Phase[tid], ph, &Par[tid]−>copy, NIL, nextnextp);
74:     return 2
    fi;
75: return DONE
```

```
procedure Del_2(tid: 1..N; ph: phtype) returns phtype
76: pred := Par[tid]−>pred;
77: nextp := pred−>next;
78: nextnextp := Par[tid]−>copy;
79: CCAS(&Phase[tid], ph, &(pred−>next), nextp, nextnextp);
80: return DONE
```

Figure 3. Wait-free implementation of linked-lists for real-time uniprocessors.

(line 72). Also, the value of *pred—>next—>next* is recorded in *Par*$[h]$*—>copy* (line 73). In *Del_2*, the recorded value of *pred—>next—>next* is copied into *pred—>next* (line 79), completing the delete operation.

This concludes our description of the linked-list implementation. When implementing other objects, *Access_Obj* and *Help* remain the same. For each implemented operation, an appropriate initialization procedure has to be defined, as well as procedures that implement the phases of that operation. The procedures implementing the phases correspond almost exactly to ordinary sequential code. In addition to linked lists, we have also used the framework given here to implement red-black trees. In that case, we were able to define the required procedures by copying sequential code almost directly out of an algorithms textbook.

A similar framework for implementing objects under the IHI scheme can also be defined. The IHI scheme differs from IHC scheme only in the way tasks help each other in the *Access_Obj* procedure. In the IHI scheme, each object has its own announce variable. As before, prior to updating an announce variable, a task must first help any previously-announced operation recorded in that variable. In the IHI scheme, a task may have to help one lower-priority operation for each shared object that it accesses.

We should emphasize here that the implementations obtained by means of the above framework are *not* the most efficient ones that exist. In practice, it is important to apply object-specific optimizations to improve performance. For example, the list implementation just presented can be optimized to remove many of the CCAS operations. Such an implementation was recently presented by us in [5]. Although counting lines of code is a very crude way of measuring performance, the optimized list implementation requires only 51 lines of code, compared to 80 lines of code in the implementation presented above. The main utility of the framework we have presented is that it can be used to implement most objects with very little effort.

2.2. Schedulability Analysis for IHC and IHI

We now present scheduling conditions that can be applied when using the object-sharing schemes described in the previous subsection. We begin by presenting terminology and notation that will be used in our analysis.

Terminology and Notation: In our analysis, we assume that all tasks are periodic and share a single processor. We call a single execution of a task a *job*. We assume that all release times and periods are integers, and that jobs can be preempted at arbitrary points during their execution. The following is a list of symbols we will use.

- N: Number of tasks in the system. We use i and j as task indices ranging over $\{1, \ldots, N\}$.
- p_i: Period of task T_i. Tasks are sorted in nondecreasing order by their periods, i.e., $p_i < p_j \Rightarrow i < j$.
- c_i: Worst-case execution cost of task T_i, including its own object accesses, but not the cost of helping.
- S_i: Set of shared objects accessed by T_i. We use x and y to denote shared objects.
- s_y: Worst-case cost of performing one operation (either directly or through helping) on object y.
- $S_{i,j}$: $\cup_{k=i}^{j} S_k$. Set of shared objects accessed by tasks T_i through T_j.
- h_i: $max\{s_y \mid y \in (S_{i+1,N} \bigcap S_{1,i})\}$. The set $S_{i+1,N} \bigcap S_{1,i}$ consists of all objects accessed both by some lower-priority task T_l, $i < l \leq N$, and by some higher-priority task T_k, $1 \leq k \leq i$.
- F_i: A set of at most $N - i$ objects of maximal total access cost satisfying the following two conditions: (i) each object in the set is accessed by some task in $\{T_{i+1}, \ldots, T_N\}$ and by some task in $\{T_i, \ldots, T_1\}$; (ii) each object x in the set can be mapped to a unique task T_l in $\{T_{i+1}, \ldots, T_N\}$ such that T_l accesses x.
- w: Maximum wasted cost of helping due to a preemption, i.e., the cost of steps performed by a task while helping that will be repeated if that task is preempted. □

Given this notation, we are now in a position to present scheduling conditions for the IHC and IHI schemes. In both cases, and throughout this paper, we focus on rate-monotonic (RM) scheduling. Conditions for other scheduling schemes can be derived in a similar manner.

A sufficient scheduling condition for the IHC scheme is given in the following theorem. The expression on the left-hand-side of the inequality in this condition gives the maximum demand placed by T_i and higher-priority tasks in an interval $[0, t)$, where $0 < t \leq p_i$. The right-hand-side gives the available processor time in this interval. Demand in $[0, t)$ consists of three terms. The first term, $\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j$, is the total demand placed on the processor by T_i and higher-priority tasks, ignoring the cost of helping. The second term, h_i, is the cost of helping the longest operation of a lower-priority task with a priority ceiling higher than T_i's priority. The third term, $\sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w$, is an upper bound on the cost of repeated work while helping. Work will be repeated only if a task gets preempted during an object access, so the total cost associated with repeated work is bounded by the number of job releases of tasks that have a higher priority than T_i in the interval $[1, t)$. (The interval $[0, 1)$ is excluded because work cannot be wasted unless some task has executed for at least one time unit.) A much tighter bound on

this term (in both this and later theorems) can be obtained by means of linear programming, as described in [3].

Theorem 1: *Under the IHC scheme with RM scheduling, a set of tasks is schedulable if the following condition holds for each task T_i.*

$$\langle \exists t : 0 < t \leq p_i : \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + h_i + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w \leq t \rangle$$

It is interesting to compare the condition above with that given for the PCP in [11, 13]. The first term in the condition in Theorem 1 is also present in the PCP condition. The second term is replaced by a blocking term in the PCP condition. (In fact, these terms arise for precisely the same reasons.) The main difference lies in the third term. In place of wasted computation due to repeated helping, a cost factor associated with performing kernel calls and maintaining various kernel data structures is incurred in the PCP. As mentioned in the introduction, the w term in our schemes can be made small by careful algorithmic design — in fact, this is an important consideration when optimizing an object implementation obtained by applying the basic framework given in Section 2.1. However, it is not possible to entirely eliminate this term.

We now present a sufficient scheduling condition for the IHI scheme under RM scheduling. In the condition stated below, the first and third terms on the left-hand-side of the inequality arise for the same reasons as explained above. The second term is the cost of helping operations of lower-priority tasks. Because there is a separate announce variable for each object, each task may have to help one lower-priority operation for each object it accesses. F_i is the set of operations of lower-priority tasks on objects that are also accessed by T_i and higher-priority tasks. A task can have at most one announced operation at any time, so each lower-priority task can contribute at most one operation to F_i.

Theorem 2: *Under the IHI scheme with RM scheduling, a set of tasks is schedulable if the following condition holds for each task T_i.*

$$\langle \exists t : 0 < t \leq p_i : \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{y \in F_i} s_y + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w \leq t \rangle$$

The above condition is very similar to that given for the PIP in [11, 13]. As mentioned before, the main difference lies in the fact that in the IHI scheme, wasted computation may result from repeated helping, whereas in the PIP, costs associated with kernel calls are incurred.

3. Multiprocessor Schemes

We now turn our attention to the CH1, CHN, PH1, and PHN object-sharing schemes for real-time multiprocessors. We assume that tasks are initially bound to processors and do not migrate. As in the uniprocessor case, we first present a general framework that can be easily adapted to implement objects under each scheme, and then present scheduling conditions for each scheme.

3.1. A Framework for Implementing Objects

We describe our multiprocessor framework within the context of the CH1 scheme. Modifications required to support the other three schemes are discussed later. The *Access_Obj* and *Help* procedures for the CH1 scheme are shown in Figure 4.

As discussed in the introduction, our multiprocessor schemes make use of one or more "help counters", each of which points to some processor on a logical ring of processors. The schemes differ in how many logical rings are supported and how a help counter is advanced around its ring. In the CH1 scheme, there is a single help counter for all shared objects in the system, and there is one announce variable per processor, which is used to announce all operations on that processor. In the following paragraphs, we explain in detail how helping is done in this scheme by considering the code in Figure 4.

We begin by considering the variable declarations in Figure 4. *Ann*$[q]$ is the announce variable for processor q. *Ann*$[q]$ equals 0 when there is no task to help on processor q. The shared variable V is a compare-only version number that is passed to `CCAS`. It consists of a counter field *cnt* and a boolean field *needhelp*. V*.cnt* is incremented after each wait-free operation and is assumed to not cycle during any single operation. The value of the help counter is given by V*.cnt mod* P. P here is defined to be the total number of processors in the system. The help counter is advanced around the logical ring of processors by means of cyclic helping. In this scheme, processors are considered in order around the ring. When the help counter is advanced to point to processor q, V*.needhelp* is set to true iff there is a task on processor q that needs to be helped. A task is allowed to help a task on processor q only if it detects that $(V.cnt \bmod P = q) \wedge V.needhelp$ holds. Thus, the decision whether or not to help a task on processor q is fixed when the help counter is advanced to point to q. Because this decision is made atomically when the help counter is advanced, there can be no disagreement among tasks as to whether a task on processor q should be helped.

Object-specific procedures in the CH1 implementation are very similar to those used in the uniprocessor IHC implementation presented earlier. In fact, when implementing linked lists, the required procedures (not shown in Figure 4) are virtually the same as before. The only difference is in the way `CCAS` is used. In the multiprocessor implementation, V is used in `CCAS` operations as a compare-only value in

```
type vertype = record cnt: 0..C − 1; needhelp: boolean end          /* These fields are stored in one word */
shared variable
    Par: array [1..N] of ptr to void;                                /* Par[i] stores parameters to task T_i's operation */
    Phase: array[0..N] of phtype initially DONE;                     /* Stores each task's "current" phase */
    Ann: array[0..P − 1] of 0..N initially 0;   /* Ann[q] is announce variable for processor q; = 0 if no currently announced operation on q */
    V: vertype                                      /* V.cnt is the version number; V.cnt mod P is the help counter;...*/
                                                   /*... V.needhelp indicates if help is needed on processor currently pointed to */
private variable                                                     /* For task T_i running on processor mypr */
    tid, nexthelp: 0..N; ver: vertype; ph, nextph: phtype

procedure Access_Obj()
1:  ver := V;
2:  tid := Ann[mypr];
3:  if tid > 0 then
4:     while ver.cnt mod P ≠ mypr ∧ Phase[tid] = 0 do
5:        if CCAS(&V, ver, &Ann[mypr], tid, i) then break fi;
6:        ver := V
       od;
7:     if Ann[mypr] ≠ i then
8:        if ¬(Phase[tid] = DONE ∧
                   (ver.cnt mod P ≠ mypr ∨ ¬ver.needhelp)) then
9:           if ver.needhelp then Help(ver) fi;
10:          nexthelp := Ann[(ver.cnt + 1) mod P];
11:          if nexthelp = 0 ∨ Phase[nexthelp] = DONE then
12:             CAS(&V, ver, ((ver.cnt + 1) mod C, false))
13:          else CAS(&V, ver, ((ver.cnt + 1) mod C, true))
             fi
          fi;
14:       Ann[mypr] := i;
15:       tid := 0
       fi
16: else Ann[mypr] := i
    fi;

/* Access_Obj() Continued */
17: while true do
18:    ver := V;
19:    if Phase[i] = DONE ∧
                (ver.cnt mod P ≠ mypr ∨ ¬ver.needhelp) then
20:       break
       fi;
21:    if ver.needhelp then Help(ver) fi;
22:    nexthelp := Ann[(ver.cnt + 1) mod P];
23:    if nexthelp = 0 ∨ Phase[nexthelp] = DONE then
24:       CAS(&V, ver, ((ver.cnt + 1) mod C, false))
25:    else CAS(&V, ver, ((ver.cnt + 1) mod C, true))
       fi
    od;
26: Ann[mypr] := tid

procedure Help(ver: vertype)
28: tid := Ann[ver.cnt mod P];
29: ph := Phase[tid];
30: while ph ≠ DONE do
31:    nextph := Par[tid]−>subop[ph](tid, ph);
32:    CCAS(&V, ver, &Phase[tid], ph, nextph);
33:    ph := Phase[tid]
    od
```

Figure 4. *Access_Obj*() and *Help*() procedures for the CH1 multiprocessor scheme.

place of *Phase*$[i]$.

As before, *Access_Obj* is called to determine which operation to help. Suppose this procedure is invoked by task T_i executing on processor q. While there is a pending announced operation on q, and while the help counter does not point to q, task T_i tries to store its task identifier in *Ann*$[q]$ by means of a CCAS operation (lines 4-6). T_i can repeatedly fail to update *Ann*$[q]$ only if the help counter continues to be advanced by tasks on other processors. In this event, the help counter will eventually point to processor q. Thus, after at most P attempts, either T_i will succeed in updating *Ann*$[q]$ or the help counter will point to processor q. If T_i fails in updating *Ann*$[q]$, then T_i checks to determine whether it must help a previously-announced operation on processor q (lines 8-9). After performing any required helping on its own processor and advancing the help counter, T_i can safely announce its own operation (lines 14 and 15).

Once T_i has successfully announced its operation, it advances the help counter until its own operation is completed (lines 17-25). To advance the help counter from a processor r to next processor in the ring, task T_i tries to help the currently-announced task on r. This helping is performed by invoking the *Help* procedure at line 21. In the *Help* procedure, the current phase of the operation to be helped is read from the *Phase* array at line 29, and the function to be performed in that phase is called at line 31. Helping continues until the operation being helped enters the *DONE* phase. It follows from this description that task T_i has to help at most P other operations: a lower-priority operation on its own processor and an operation on each of the other $P - 1$ processors.

The PH1 scheme is similar to the CH1 scheme except that priority helping is used. Recall that with priority helping, the help counter is always advanced to the processor with the highest-priority pending operation. Implementing this scheme requires a few straightforward changes to lines 20-23 in Figure 4. In addition, each processor q's announce variable must hold both the identity of the current task to help on processor q and the priority of the currently running task on processor q. These may be different tasks. In particular, if a task T_i on processor q helps a lower-priority task T_j on q, then T_i must first record its own priority in q's announce variable. Otherwise, if there are other announced operations on other processors with priority greater than T_j's but less than T_i's, then T_i may be delayed unnecessarily. This is very similar to priority inheritance in lock-based object-sharing

schemes [11]. When using priority helping to implement an object shared across P processors, advancing the help counter requires an $O(P)$ scan of all announce variables. Our priority helping implementation has the property that if an operation is of highest priority, then at most one other concurrent operation can be completed before it.

The CH1 and PH1 schemes just described might be unduly restrictive for some applications, because they are implemented using only one help counter for all shared objects. If tasks on different processors access disjoint sets of objects, then this may negatively impact concurrency. In such cases, having multiple help counters may significantly improve performance. The CHN and PHN schemes do exactly this. In both schemes, each shared object has its own help counter. For each counter, a logical ring of processors is defined consisting of exactly those processors with tasks that may access the corresponding object. Apart from simple changes required to implement multiple helping rings, most of the implementation details of the CHN and PHN schemes are the same as described above for the CH1 and PH1 schemes, respectively.

For the CHN scheme, a task has to help at most P' other operations when accessing an object shared across P' processors — the reasoning here is exactly the same as given above for the CH1 scheme. Note that P' may be much less than P, so costs associated with helping operations on *remote* processors may be much less here than with the CH1 scheme. On the other hand, one lower-priority local operation may have to be helped for each object access with the CHN scheme, so higher *local* helping costs may be incurred. Similar tradeoffs between local and remote helping costs exist with the PHN and PH1 schemes.

3.2. Schedulability Analysis of Multiprocessor Helping Schemes

In this subsection, we present scheduling conditions to be used in conjunction with the object-sharing schemes presented in the previous subsection. We begin by stating some notational conventions.

Terminology and Notation: The following is a list of symbols that will be used in deriving the scheduling conditions presented in this subsection.

- Most of the uniprocessor notation defined in Section 2.2 will be used here as well. We still have a set of tasks T_1 through T_N ordered by period, but now these tasks may be assigned to different processors.
- P: Number of processors. We use p, q, and r as processor indices ranging over $\{0, \ldots, P-1\}$.
- m_i: Number of global shared objects accessed by T_i. A shared object is *global* if it is accessed by tasks on more than one processor, and is *local* otherwise.
- $m_{i,x}$: Number of accesses of object x by a job of T_i.
- s: Maximum time taken to access any global shared shared object.
- $S^q_{j,i}$: Set of shared objects accessed by tasks on processor q that have a priority higher than or equal to priority of T_i, but lower than that of T_j.
- P_x: Set of all the processors that have tasks accessing global shared object x.
- HP^q_i: Set of tasks on processor q that have a higher priority than task T_i.
- HEP^q_i: Set of tasks on processor q that have a priority higher than or equal to that of task T_i.
- LP^q_i: Set of tasks on processor q that have lower priority than T_i.
- h^q_i: $max(s_y | y \in \cup_{j \in LP^q_i} S_j)$. h^q_i is the maximum cost of accessing any object that is accessed by tasks with priority lower than T_i and running on processor q.
- h^q: Maximum-cost object access by any task on processor q.
- h_i: $max(s_y | y \in \cup_{j \in \{i+1,\ldots,N\}} S_j)$. h_i is the maximum cost of accessing any object that is accessed by tasks with priority lower than T_i.
- F^q_i: A set of at most $|LP^q_i|$ objects of maximal total access cost satisfying the following two conditions: (i) each object in the set is accessed by some task in LP^q_i and by some task in HEP^q_i; (ii) each object x in the set can be mapped to a unique task T_l in LP^q_i such that T_l accesses x.
- G^q_i: A set of at most $N-i$ objects of maximal total access cost satisfying the following two conditions: (i) each object in the set is accessed by some task in $\{T_{i+1}, \ldots, T_N\}$ and by some task in HEP^q_i; (ii) each object x in the set can be mapped to a unique task T_l in $\{T_{i+1}, \ldots, T_N\}$ such that T_l accesses x.
- $HP^r_{i,x}$: All tasks on processor r that have a higher priority than T_i and access object x. □

We now state and prove a pessimistic scheduling condition that applies to cyclic helping schemes. This condition will help in understanding better conditions for the CH1 and CHN schemes given later.

Theorem 3: *Under either the CHN scheme or the CH1 scheme with RM scheduling, a set of tasks is schedulable if the following condition holds for each task T_i running on any processor q.*

$$\langle \exists t : 0 < t \le p_i : \sum_{j \in HEP_i^q} \left\lceil \frac{t}{p_j} \right\rceil \cdot (c_j + m_j \cdot P \cdot s) + \sum_{j \in HP_i^q} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w \le t \rangle$$

Proof Sketch: The terms $\sum_{j \in HEP_i^q} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j$ and $\sum_{j \in HP_i^q} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w$ arise for precisely the same reasons as in our uniprocessor conditions. The term $\sum_{j \in HEP_i^q} \left\lceil \frac{t}{p_j} \right\rceil \cdot m_j \cdot P \cdot s$ arises because $P \cdot s$ is an upper bound on helping costs for each operation on an object. This bound includes the time to help one operation of a lower-priority task executing locally and the time to help one operation on each of at most $P - 1$ remote processors on the corresponding helping ring. In defending the bound on remote helping costs, an interesting case arises. Assume all P processors are on the same helping ring and suppose that a task T_i on processor $P - 1$ succeeds in moving the help counter from processor 0 to some processor p such that $0 < p < P - 1$. (The choice of processors here is not important, but simplifies the explanation.) Suppose further that T_i is preempted by a higher-priority task T_j as soon as the help counter reaches processor p. At that point T_j will overwrite T_i's announce information and continue advancing the help counter around the ring. Once T_i resumes execution, the help counter may point to any processor on the ring, even processor 0. In this case, it seems that the computation performed previously by T_i to advance the help counter from processor 0 to processor p, a cost of $p \cdot s$, is completely wasted. This would seem to imply that charging T_i a cost of $P \cdot s$ for its operation is not correct. However, in advancing the help counter to processor p, T_i *reduces* the distance that T_j must advance the help counter. As a result, T_j's helping cost is reduced by $p \cdot s$. In effect, the cost of the "wasted" steps performed by T_i become part of the $P \cdot s$ cost charged to T_j. □

Although the condition of Theorem 3 exposes much of the intuition as to how cyclic helping costs can be bounded, it is rather pessimistic. The following theorem gives a better condition for the CH1 scheme.

Theorem 4: *Under the CH1 scheme with RM scheduling, a set of tasks is schedulable if the following condition holds for each task T_i running on any processor q.*

$$\langle \exists t : 0 < t \le p_i : \sum_{j \in HEP_i^q} \left\lceil \frac{t}{p_j} \right\rceil \cdot (c_j + m_j \cdot \sum_{\substack{r=0 \\ r \ne q}}^{P-1} h^r) + h_i^q + \sum_{j \in HP_i^q} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w \le t \rangle$$

Given the condition of Theorem 3, the condition above is rather straightforward to understand. In the CH1 scheme, the remote helping cost of each operation can be bounded by $(P - 1) \cdot s$. The expression $\sum_{\substack{r=0 \\ r \ne q}}^{P-1} h^r$ gives an even tighter bound because operations of different costs may be performed from different processors. In addition, at the beginning of an interval over which jobs of T_i and higher-priority tasks on processor q execute continuously, there can be at most one announced operation of a lower-priority task on q that may be helped. h_i^q is an upper bound on the worst-case cost of local helping in such a case.

It is also possible to define a better condition for the CHN scheme. Such a condition is given next.

Theorem 5: *Under the CHN scheme with RM scheduling, a set of tasks is schedulable if the following condition holds for each task T_i running on any processor q.*

$$\langle \exists t : 0 < t \le p_i : \sum_{j \in HEP_i^q} \left\lceil \frac{t}{p_j} \right\rceil \cdot (c_j + \sum_{x \in S_j} \sum_{\substack{p \in P_x \\ p \ne q}} m_{j,x} \cdot s_x) + \sum_{x \in F_i^q} s_x + \sum_{j \in HP_i^q} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w \le t \rangle$$

Like Theorem 4, this condition also differs from that given in Theorem 3 in how helping costs are bounded. In CHN scheme, there is a logical ring associated with each shared object. When accessing a shared object, a task has to advance the help counter at most once around the logical ring for that object. Thus, the remote helping cost for a job of a task T_j running on processor q is given by $\sum_{x \in S_j} \sum_{\substack{p \in P_x \\ p \ne q}} m_{j,x} \cdot s_x$. The term $\sum_{x \in F_i^q} s_x$ bounds local helping costs and arises for the same reasons as in the IHI uniprocessor scheme.

We now turn our attention to the two priority helping schemes. We begin with the PH1 scheme.

Theorem 6: *Under the PH1 scheme with RM scheduling, a set of tasks is schedulable if the following condition holds for each task T_i running on any processor q.*

$$\langle \exists t : 0 < t \le p_i : \sum_{j \in HEP_i^q} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{\substack{r=0 \\ r \ne q}}^{P-1} \sum_{k \in HP_i^r} \left\lceil \frac{t}{p_k} \right\rceil \cdot m_k \cdot h^r + h_i + \sum_{j \in HP_i^q} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w \le t \rangle$$

By this point, the first and last summation terms should require no explanation. In The PH1 scheme, there is one ring of processors that is used for all object accesses. A job J of a task T_i running on processor q has to help operations of all jobs, on all processors other than q, that have higher priority than J and that execute in the interval between J's release and its deadline. The cost of helping such operations is given by $\sum_{\substack{r=0 \\ r \ne q}}^{P-1} \sum_{k \in HP_i^r} \left\lceil \frac{t}{p_k} \right\rceil \cdot m_k \cdot h^r$. In addition, when priority helping is used, a task may have to help one lower-priority task due to the fact that announce variables are being scanned and updated concurrently. This helping cost is reflected by the h_i term in Theorem 6.

The final condition we present is for the PHN scheme. It is stated in the following theorem.

Theorem 7: *Under the PHN scheme with RM scheduling, a set of tasks is schedulable if the following condition holds for each task T_i running on any processor q.*

$$\langle \exists t : 0 < t \leq p_i : \sum_{j \in HEP_i^q} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j \in HEP_i^q} \sum_{x \in S_j} (x \notin S_{j,i}^q) \sum_{\substack{r=0 \\ r \neq q}}^{P-1} \sum_{k \in HP_{j,x}^r} \left\lceil \frac{t}{p_k} \right\rceil \cdot m_{k,x} \cdot s_x + \sum_{x \in G_i^q} s_x + \sum_{j \in HP_i^q} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot w \leq t \rangle$$

The second term above bounds remote helping costs. In the PHN scheme, there is one logical ring per object. When performing an operation on an object x, a job J of a task T_i on processor q has to help all operations on x performed by all jobs, on all processors other than q, that have higher priority than J and that execute in the interval between J's release and its deadline. Also, if J is preempted by a job J' of a higher-priority task T_j on processor q, then the remote helping cost incurred by job J' in helping higher-priority tasks access some object y will contribute to the remote helping cost of J even if J does not access object y. The third term bounds the cost of helping lower-priority tasks. In the PHN scheme, there can be at most one announced operation for every shared object in the system. Also, over any interval during which tasks in HEP_i^q execute on processor q, each lower-priority task can be helped at most once. This is why G_i^q is used in the third term.

4. Experimental Results

In this section, we present results from experiments conducted to compare the performance of the multiprocessor priority ceiling protocol (MPCP) [10, 11, 12] with that of the four wait-free multiprocessor schemes presented in this paper. The comparison is based on RM scheduling conditions given in this paper and in [10, 11, 12]. We have also conducted similar experiments to compare our schemes with the distributed priority ceiling protocol (DPCP) [10, 11, 12], and the results from those experiments are similar to those reported here for the MPCP.

The experiments that were conducted involve randomly-generated task sets consisting of 40 tasks executing on four processors (ten tasks per processor) and up to 72 shared objects, obtained by varying two parameters: *local-access* and *conflicts*. Each task consists of seven phases:[3] three object-access phases and four computation phases. The *local-access* parameter denotes the number of accesses to objects local to a processor performed by each task. If the *conflicts* parameter is k, then at least one object is accessed by tasks on k processors, and no object is accessed by tasks on more than k processors.

Task periods were randomly selected from a predetermined set of 44 periods ranging from 13,650 to 10,348,800 time units. Computation-phase costs ranged from one to 500 time units, subject to the constraint that per-processor utilization is at most one. In all experiments, context switch costs were ignored. Note that these costs have a much greater impact on the MPCP.

In our experiments, the sequential cost of a wait-free operation was assumed to be equal to its lock-based counterpart (the latter includes the cost of performing semaphore operations). We believe that this assumption is reasonable, but further experimental work involving actual testbeds is important to verify its validity. In previous experimental work we have done, we have found that lock-free and wait-free object implementations that do not rely on state-copying give rise to sequential operation costs that are comparable to sequential operation costs in lock-based schemes [6]. The wait-free schemes considered in this paper do not rely on state-copying, and with suitable optimizations should perform very well. In our experiments, object-access costs were randomly generated assuming a normal distribution with mean and standard deviation of 50 and 30 time units, respectively. Only single-object accesses were considered. Also, the wasted cost of helping (w) was assumed to be one-fifth of the largest object-access cost. This assumption is rather pessimistic because the wasted cost of helping is likely to be much smaller in practice.

To compare the performance of the various schemes, we calculated the *breakdown utilization* (BU) for each generated task set. The BU of a task set is determined by scaling the cost of task phases, and is defined to be the maximum total utilization at which a task set is still schedulable. BU curves resulting from our experiments are shown in Figure 5. Each curve in figure was generated from 5,000 task sets.

The BU curves in Figure 5 indicate that the PHN scheme outperforms all the other schemes in all situations. This is because, under the PHN scheme, a task helps at most one lower-priority operation for each object access it performs, and it only helps tasks with which it has a potential conflict. In the PH1 and CH1 schemes, one helping ring is used for all object accesses, which forces a task to help operations of tasks with which no common object is shared. In the CH1 and CHN schemes, cyclic helping is performed, which forces a task to help lower-priority operations on remote processors. The MPCP performs worse than three of our four schemes, and its performance degrades sharply with increasing conflicts.

5. Concluding Remarks

In the wait-free implementations we have presented, operations are performed in a near-sequential manner. As a result, costly mechanisms that require tasks to make copies of an object's state in order to avoid interferences are not required. Costs associated with copying have been perhaps the most significant limitation preventing practical applications of previous wait-free algorithms. In addition, sequential

[3]These phases should not be confused with those in the general framework discussed in Section 2.1.

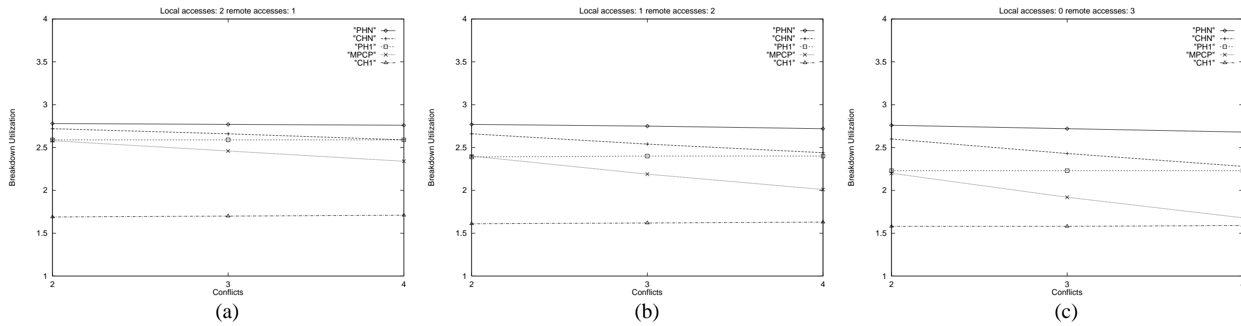

Figure 5. Comparison of the MPCP and wait-free multiprocessor schemes.

execution makes multi-object operations (e.g., atomically removing an item from one list and inserting it into another) straightforward to implement: this simply requires defining appropriate phases to implement the required operation sequence. On the other hand, executing operations in sequence penalizes read-only operations, which cannot interfere with each other and thus should be able to execute concurrently. However, with a slight modification, it is possible to implement read-only operations with greater concurrency. This involves adding an extra "virtual" processor to the helping ring. While the help counter points to the virtual processor, any read-only operation can be performed without helping other operations. If updates are rare, then the help counter would almost always point to the virtual processor, and read-only operations would execute with very little synchronization overhead.

References

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

[2] J. Anderson and M. Moir, "Universal Constructions for Large Objects", *Proc. of the Ninth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 972, 1995, pp. 168-182.

[3] J. Anderson and S. Ramamurthy, "A Framework for Implementing Objects and Scheduling Tasks in Lock-Free Real-Time Systems", *Proc. of the 17th IEEE Real-Time Systems Symp.*, 1996, pp. 94-105.

[4] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay, "Lock-Free Transactions for Real-Time Systems", *Proc. of the First International Workshop on Real-Time Databases: Issues & Applications*, 1996, pp. 107-114.

[5] J. Anderson, S. Ramamurthy, and R. Jain "Implementing Wait-Free Objects on Priority-Based Systems", *Proc. of the 16th ACM Symp. on Principles of Distributed Computing*, 1997, pp. 229-238.

[6] J. Anderson, S. Ramamurthy, and K. Jeffay, "Real-Time Computing with Lock-Free Shared Objects", *ACM Trans. on Computer Systems*, 15(2), May 1997, pp. 134-165.

[7] R. Bettati, *End-to-End Scheduling to Meet Deadlines in Distributed Systems*, Ph.D. Thesis, Computer Science Department, University of Illinois, March 1994.

[8] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Trans. on Progam. Languages and Systems*, 15(5), 1993, pp. 745-770.

[9] G. Peterson, "Concurrent Reading While Writing", *ACM Trans. on Progam. Languages and Systems*, 5(1), 1983, pp. 46-55.

[10] R. Rajkumar, "Real-Time Synchronization Protocols for Shared Memory Multiprocessors", *Proc. of the International Conference on Distributed Computing Systems*, 1990, pp. 116-123.

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

[12] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors", *Proc. of the Nineth IEEE Real-Time Systems Symp.*, 1988, pp. 259-269.

[13] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time System Synchronization", *IEEE Trans. on Computers*, 39(9), 1990, pp. 1175-1185.

[14] J. Sun, R. Bettati, and J. W.-S. Liu, "Using End-to-End Scheduling Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems", *Proc. of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, 1994.