Non-blocking Data Sharing in Multiprocessor Real-Time Systems*

Philippas Tsigas and Yi Zhang
Department of Computing Science
Chalmers University of Technology
S-412 96 Göteborg, Sweden

<tsigas,yzhang>@cs.chalmers.se

Abstract

A non-blocking protocol that allows real-time tasks to share data in a multiprocessor system is presented in this paper. The protocol gives the means to concurrent real-time tasks to read and write shared data; the protocol allows multiple write and multiple read operations to be executed concurrently. Our protocol extends previous results and is optimal with respect to space requirements. Together with the protocol, its schedulability analysis and a set of schedulability tests for a set of random task sets are presented. Both the schedulability analysis and the schedulability experiments show that the algorithm presented in this paper exhibits less overhead than the lock based protocols.

1. Introduction

In any multiprocessing system cooperating processes share data via shared data objects. In this paper we are interested in designing shared data objects for cooperative tasks in real-time multiprocessor systems.

The challenges that have to be faced in the design of inter-task communication protocols for multiprocessor systems become more delicate when these systems have to support real-time computing. In real-time multiprocessor systems inter-task communication protocols i) have to support sharing of data between different tasks e.g. on an operational flight program (OFP), tasks like navigation, maintaining of pilot displays, control of and communication with a variety of special purpose hardware, weapon delivery, and so on; ii) must meet strict time constraints, the HRT deadlines; and iii) have to be efficient in time and in space since they must perform under tight time and space constraints. A nice description of the fine challenges that inter-task communication protocols for real-time systems have to address can be found in [11].

The classical, well-known and most simple solution enforces mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at time to access the data. Mutual exclusion i) causes large performance degradation especially in multiprocessor systems [12]; ii) leads to complex scheduling analysis since tasks can be delayed because they were either preempted by other more urgent tasks, or because they are blocked before a critical section by another process that can in turn be preempted by another more urgent task and so on. (this is also called as the convoy effect) [2]; and iii) leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [4]. Several synchronisation protocols have been introduced to solve the priority inversion problem for uniprocessor [4] and multiprocessor [3] systems. The solution presented in [4] solves the problem for the uniprocessor case with the cost of limiting the schedulability of task sets and also making the scheduling analysis of real-time systems hard. The situation is much worse in a multiprocessor real-time system, where a task may be blocked by another task running on a different processor [3].

Non-blocking implementation of shared data objects is a new alternative approach for the problem of inter-task communication. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Non-blocking inter-task communication does not allow one task to block another task gives significant advantages over lock-based schemes because:

1. it does not give priority inversion, avoids lock con-

*Partially supported by ARTES, a national Swedish strategic research initiative in Real-Time Systems and TFR the Swedish Research Council for Engineering Sciences.

0-7695-0306-3/99 $10.00 © 1999 IEEE

voys that make scheduling analysis hard and delays longer.

2. it provides high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios from two or more tasks both waiting for locks held by the other.
3. and more significantly it completely eliminates the interference between process schedule and synchronisation.

Non-blocking protocols on the other hand have to use more delicate strategies to guarantee data consistency than the simple enforcement of mutual exclusion between the readers and the writers of the data object. These new strategies on the other hand, in order to be useful for real-time systems, should be efficient in time and space in order to perform under the tight space and time constraints that real-time systems demand.

In this paper, we present an efficient non-blocking solution to the general readers/writers inter-task communication problem; our solution allows any arbitrary number of readers and writers to perform their respective operations. With a simple and efficient memory management scheme in it, our protocol needs $n + m + 1$ memory slots (buffers) for n readers and m writers and is optimal with respect to space requirements. Sorencen and Hemacher in [8] have proven that $n + m + 1$ memory slots (buffers) are necessary. Together with the protocol its schedulability analysis and a set schedulability tests for a set of random task sets are presented. Both the schedulability analysis and the schedulability experiments show that the algorithm presented in this paper exhibits less overhead than the lock based protocol. Our protocol extends previous results by allowing any arbitrary number of tasks to perform read or write operations concurrently without trading efficiency. In previous work, Simpson [6], presented a non-blocking asynchronous protocol for task communication between 1 writer and 1 reader which needs 4 buffers. Chen and Burns [7] presented a non-blocking synchronous protocol for n readers and 1 writer that needs $n+1+1$ buffers. Kopetz and Reisinger [2] also presented a non-blocking synchronous protocol for n readers and 1 writer that contains a mechanism to configure the number of buffers to the application requirements, trading memory space for execution time. We also believe that the memory management scheme that we introduce in this paper and use in our protocol is of interest and can be used as an independent component with other non-blocking shared data object implementations.

The rest of this paper is organised as follows. In Section 2 we give a description of the basic characteristics of a multiprocessor architecture and describe the formal requirements that any solution to the synchronisation problem that we are addressing must guarantee. Section 3 presents our protocol. In Section 4, we give the proof of correctness of the protocol. Section 5 is devoted to the schedulability analysis and schedulability experiments that compare our non-blocking protocol with the lock-based one. The paper concludes with Section 6.

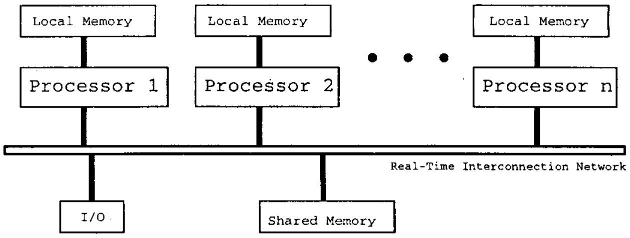

Figure 1. Shared Memory Multiprocessor System Structure

2. Problem Statement

2.1. Real-time Multiprocessor System Configuration

A typical abstraction of a shared memory multiprocessor real-time system configuration is depicted in Figure 1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of cooperating tasks (processes) with timing constraints is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The cooperating tasks now, possibly running on different processes, use shared data objects build in the shared memory to coordinate and communicate. Every task[1] has a maximum computing time and has to be completed by a time specified by a deadline. Tasks synchronise their operations through read/write operations to shared memory.

2.2. General Reader/Writer Problem

In this paper we are interested in the general read/write buffer problem where several reader-tasks

[1] throughout the paper the terms *process* and *tasks* are used interchangably

(the readers) access a buffer maintained by several writer-tasks *(the writers)*. There is no limit on the length of the buffer that can be increased by the writers on-line depending on the length of the data that they want to write in one atomic operation. This shared data object can be used to increase the word length of the system.

The accessing of the shared object is modelled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a read operation or a write operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $<_h$ are called *overlapping.* A complete history h is linearisable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_{\mathrm{h}}$that respects the specification of the object [1]. For our object this means that each read operation should return the value written by the write operation that directly precedes the read operation by this total order ($\rightarrow_{\mathrm{h}}$).

To sum it up as we are looking for a non-blocking solution to the general Read/Writer problem for real-time systems we are looking for a solution that satisfies that:

- Every read operation guarantees the integrity and coherence of the data it returns
- The behaviour of each read and write operation is predictable and can be calculated for use in the scheduling analysis
- Every possible history of our protocol should be linearisable.

We assume that writer-tasks have the highest priority on the host processor and no two writer-tasks execute on one host processor. Two writer-tasks may overlap but no write processes can be preempted by another process. Reader processes may have different priorities and may be scheduled with the writer process on the same processor. This is because writer-tasks usually interact with the environment (e. g. sampler for temperature or pressure), and they are of high prioritiy. To the best of our knowledge the above assumption holds in most real-time systems.

```
Compare_and_Swap(int *mem, register old, new)
{
   temp = *mem;
   if (temp == old)
   {
       *mem = new;
       new = old;
   }
   else
       new = *mem
}
```

Figure 2. The *Compare_ and_ Swap* **atomic primitive**

3. The Protocol

3.1. Idea description

Our construction divides the memory into memory slots and uses a pointer that points to the slot with the latest written data. Each slot has a flag field used by the special memory management mechanism in the protocol. Through this mechanism, i) writers can find a safe slot to write without corrupting the slots from where overlapping readers are getting their values and ii) readers find the slot with the latest information.

In the worst case, n readers occupy n slots to read and m writers allocate m slots to write and the pointer points to another slot. So, in total we need at most $n + m + 1$ slots for our protocol. In [8], Sorensen and Hemacher showed that $n + m + 1$ slots is necessary for this problem.

3.2. Protocol Description

Our protocol uses the instructions *Compare_ and_ Swap*[2] and *Fetch_ and_ Add.* The respective specifications of these instructions are shown in Figure 2 and Figure 3. Most multiprocessor systems either provide these primitives or provide others that can be used to emulate these primitives.

[2] IBM System 370 was the first computer system that introduced Compare_and_Swap

```
int Fetch_and_add(int *mem, int increment)
{
  temp = *mem;
  *mem = *mem + increment;
  return temp
}
```

Figure 3. The *Fetch_ and_ Add* **atomic primitive**

Figure 4 presents commented pseudo-code for our nonblocking protocol. We use $n + m + 1 = TASKS'NUM + 1$ slots. Each slot has a field, called 'used' that is used by the memory management layer of our protocol. There are a number of values that this field can carry, these values together with an informal description of the associated information can be described as follows:

- $k = 0$: indicates that the writer has finished writing in the respective slot but no reader has read it yet
- $k > 0$: indicates that this is the slot with the most recent value and there are k readers reading this slot at this moment
- $-(n+m+1) < k < 0$: indicates that this is not the slot with the most recent value but there are $k+(n+m+1)$ readers currently reading it. These readers started their operations long ago when this slot had the most recent value
- $k = -(n+m+1)$: indicates that this is not the slot with the most recent value and that there is no reader reading it; the combination of these two makes it ideal for a writer to allocate this slot to its current operation and use it
- $k = -2(n+m+1)$: indicates that a writer has allocated this slot and is currently writing in it

```
structure readwrite_buf
{
data: array[LENGTH_BUF] of Character;
used: integer;
/*-2*TASK'NUM means allocated & writing data
  -TASK' NUM means slot is free
  -TASK' NUM +1 ~ -1 means slot is free but
                with readers accessing it
   0 means validate data  */
} LF_buffer /* Lock-free buffer */
workbuf: array[TASK' NUM+1] of LF_buffer
dataptr: pointer to LF_buffer

function newbufcell(): pointer to LF_buffer
 for i=0 to TASK' NUM
  if (cas(work_buf[i].used,
-TASK' NUM ,-2*TASK' NUM )
   return &work_buf[i]
 return NULL

function initdata
  for i=0 to TASK' NUM
      work_buf[i].used=-TASK' NUM
  dataptr=work_buf[0]^

function writebuf(data: pointer to datatye)
  temp = newbufcell()
/* allocate a new buffer cell for write */
  writingsth(temp,data)
/* write data into new buffer cell */
  temp.used=0
/* means data valid in the cell */
  swap(dataptr,old,temp)
/* use cas to achieve non-blocking write */
  faa(old->used, -TASK'NUM )
/* if no reader then it will change to
     -TASK' NUM */

function readbuf(): pointer to datatype
/* use fetch and add to mark
    that current block is in use. */
  loop
    reading=dataptr
  until (faa(reading->used,1)>(-TASK'NUM))
/* increase used by 1 if
    it is greater than -TASK' NUM
    it can not be recycled*/
  return=readingsth(reading)
  faa(reading->used,-1)
```

Figure 4. The Structure and Operations description for our non-blocking shared object

The read/write protocol can be described informally as follows.

A writer runs the following steps whenever it wants to write data into the object:

1. First, it allocates a free slot from the slots with flags entry $k = -(n+m+1)$. The writer uses *Compare_ and_ Swap* to read and change the flag from $k = -(n+m+1)$ to $k = -2(n+m+1)$ in one atomic operation, in this way if several writers want to allocate the same slot, the *Compare_ and_ Swap* atomic operation available at a

hardware level will guarantee that only one will succeed.

2. Then, it writes the data that it wants to write into that slot. If one reader tries to access that slot during this writing, the reader will give up and will be forced to try again as we will see in the readers protocol description.

3. After the writing has finished, the writer changes the flag entry of that slot from $k = -2(n+m+1)$ to 0 .

4. As a next step the writer changes the data pointer variable to this new slot (so that consequent reads find a pointer to the fresh value) with the atomic primitive *Swap* and gets the pointer to the old slot at the same time.

5. Finally, the writer changes the flag entry of the old slot from $k > 0$ to $k < 0$ by subtracting $n+m+1$.

A reader performs the following steps during each read operation:

1. reads the data pointer variable to get a pointer to the slot with the most recently written value

2. uses the atomic primitive *Fetch_ and_ Add* to get and to add 1 on the value k of the flag entry of that slot

3. there are three possibilities for the value that the fetch_ and_ add will return to the reader

   (a) $k \geq 0$, the slot has the most recent value written by all writers until now, and the reader can return this value, so go to 5 to get the value

   (b) $0 > k \geq -(n+m+1)$, the slot has the most recent value with respect to this specific reader from the linearisabilty point of view and the reader can return this value, so go to step 5 to get the value.

   (c) $k < -(n+m+1)$, the slot has been "recycled" by some writer and the data that the slot holds is invalid

4. goto step 1

5. reads the data out of the slot

6. uses the atomic primitive *Fetch_ and_ Add* with value -1 to change the value k of the flag entry of the slot

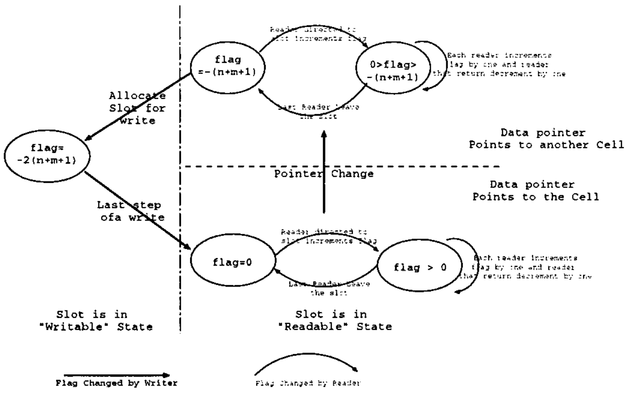

Figure 5. Slot state changing graph

A description of the different states that a slot can be in, together with the description of the actions that cause the change of these states, is depicted in Figure 5.

From the above description, it is easy to check that the protocol: i) guarantees continuous flow of information from the writers to the readers, without any blocking; ii) makes it possible to add or remove readers and writers on the fly without any change to the protocol of the other tasks; and iii) does not require information about how big the buffer has to be.

4. Correctness

4.1. Model and Definitions

In [9] one can find a formalism for the notion of the atomic buffer and the global time assumption that we adopt. We assume that each operation Op has a time interval $[s_{Op}, f_{Op}]$ on a linear time axis. We can think of s_{Op} and f_{Op} as the starting and finishing times of Op. Moreover, we assume that there is a precedence relation on operations which is a strict partial order (denoted by '$<_h$'). Semantically, $a <_h b$ means that operation a ends before operation b starts. If two operations are incomparable under $<_h$, they are said to **overlap**.

A **reading function** R_B for a buffer B is a function that assigns a write operation W to each read operation R on B, such that the value returned by R is the value written by W. It is assumed that there exists a write operation, which initialises the buffer B, that precedes all other operations on B.

A complete history on a buffer is an execution of an arbitrary number of operations according to its protocol. With the linearisability definition described in section 2.2, a complete history on a buffer B is linearisable if there is a total order $\rightarrow_{\mathrm{h}}$ on the set of all the operations of the run such that: (i) The total order $\rightarrow_{\mathrm{h}}$ extends the precedence relation $<_h$, (ii) every read operation r returns a value which is equal to the value written by a write operation that directly precedes r in the total ordering $\rightarrow_{\mathrm{h}}$. An implementation of a buffer is linearisable if all its complete histories are linearisable.

4.2. Correctness Proof

In this subsection will show that any complete history is linearisable. Our proof uses a widely used lemma in the area that can be found also in [10].

Lemma: A complete history h is linearisable iff there exist a function mapping each operation in h to a rational number such that the following three conditions are satisfied:

Precedence if one operation precedes another, then the value of the latter is at least that of the former

Uniqueness different write operations have different numbers

Integrity for each read operation there exists a write operation with the same value which it doesn't precede.

Now we associate a tag field with the pointer variable that associates a tag value to each value that is written in this variable in the following way: every time a write operation updates the pointer with a swap operation the tag is increased by one. The tag field and variable are auxiliary and are introduce only to help the proof.

Now, it is easy to associate a tag value to each write operation and each read operation in a way that guarantees the above mentioned conditions. A write operation is associated with the tag value that it writes and a read operation is associated with the tag value that it read when it read the pointer variable; in this way different write task are associated to different values (tag values only increase) and read tasks are associated with the respective write operations that wrote the values that they return.

Theorem Each complete history of our protocol is linearisable.

5. Schedulability Analysis

5.1. Worst Case Analysis

As we mentioned in Section 2 we assume that writers have the highest priority on the nodes where they are running. In our protocol i) write operations are guaranteed to finish independently from interleaving with other task operations after a finite number of steps; ii) read operations are subject to retry under certain circumstances. A read operation might have to retry only if there are more than two write operations concurrently with it. An overlap between a reader operation and other read operations does not cause reader to retry. In this section we will estimate the number of steps that a read operation will need, in the worst case, before it finishes; any write operation will finish after 5 steps.

For our analysis we will use the following notation:

- P_w: period of the writer
- C_i: Compute time of task i
- T_r: read latency caused by one retry
- D_i: the deadline of task i
- N_i: maximum number of interventions
- Wi: the worst case executing time of task i

Because tasks on different processors are decoupled, the scheduling problem for a multiprocessor system is converted to the scheduling problem on several uniprocessor systems. If we can find out the worst case executing time of each task then we can use any scheduling algorithm for uni-processor systems to schedule them.

If there exist a schedule that every reader task meets its deadline, then the max number of interventions for each read operation is bounded by

$N_i = \left\lceil \frac{D_i}{2*P_w} \right\rceil$

The worst case execution time of task i can then be represented as

$W_i = C_i + N_i * T_r = C_i + \left\lceil \frac{D_i}{2*P_w} \right\rceil T_r$

5.2. Rate-Monotonic

Once the worst case execution time of task i has been determined, the schedulability for a multiprocessor system can be worked out as follows:

On each processor, there is a set of periodic tasks. These tasks are assigned priorities according to their periods, T_i. Higher priority tasks can preempt lower priority tasks. Then, a set of n periodic tasks can be scheduled by the rate-monotonic algorithm if the following condition is met:

$$\sum_{i=1}^{n} \left(\frac{W_i}{T_i}\right) < n(2^{1/n} - 1)$$

where

$$W_i = C_i + N_i * T_r = C_i + \left\lceil \frac{T_i}{2 * P_w} \right\rceil T_r$$

5.3. Scheduling experiment

We conduct the following scheduling experiments: **In the first experiment** all tasks need to access to the multi-word buffer. There are fixed number of writers that write to the buffer with period P_w. Every reader task needs the same computing time and all tasks have the same deadline and period. We try, in the experiments, to schedule as many reader tasks as possible. All the experiment parameters are listed as following:

C_i:	Compute time of task i: 800usec
T_{rw}:	Read/Write buffer time: 100usec
D_i:	the deadline of task: 10msec
T_r:	read latency caused by one retry: 10usec
P_w:	period of the writer: 1msec
P_r:	period of the reader: 10msec
bcnt :	the number of buffers for the message, used by Kopetz's protocol: 4

For the non-blocking algorithm, we calculate the worst case execution time:

$W_i = C_i + \left\lceil \frac{D_i}{2*P_w} \right\rceil T_r = 800 + \left\lceil \frac{10}{2*1} \right\rceil * 10 = 850usec$

In this experiment, we compare our algorithm with a lock-based algorithm. The lock-based protocol is using PCP to avoid Priority inversion. Figure 6 shows the result of the schedule simulation. The solid line represents the ideal number of reader tasks based on the lock algorithm that can be scheduled and the dashed line represents that of our non-blocking protocol. As it can be seen, the schedulability of our non-blocking protocol is the same as or better than the other protocol all the time.

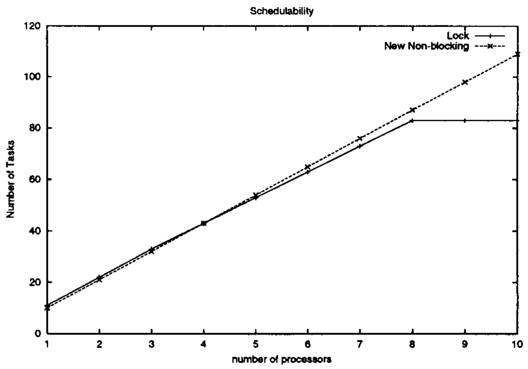

Figure 6. Fixed computing time schedule experiment

In the second experiment, we consider different randomised computing times for different reader tasks. All parameters except the computing time for the reader tasks are the same as the first experiment. The worst case execution time for the non-blocking algorithm is calculated with the equation in section 5.2. We create two sets of randomised computing time tasks with different random seeds. Figure 7 shows the result of two task sets. The results also show that with randomised computing time our non-blocking protocol also gives better scheduling results compared to the lock-based one.

6. Conclusion

In this paper, we presented a non-blocking protocol that allows real-time processes to share data in a multiprocessor system. The protocol provides the means for concurrent real-time tasks to read and write the shared data; the protocol allows multiple write and multiple read operations to be executed concurrently. Together with the protocol its schedulability analysis and a set of schedulability tests are given. Both the schedulability analysis for a task set and the schedulability experiments show that the algorithm presented in this paper exhibit less overhead than the lock based protocols.

The space requirements of our protocol is the same with the lower bound shown by Sorencen and Hemacher [8]. Our protocol only needs $n+m+1$ memory slots where n is the number of readers that perform their read operations concurrently and m is the number of writers that can write concurrently. Our protocol extends previous results by allowing any arbitrary number of tasks to perform read or write operations concurrently without compromising efficiency.

We believe that the memory management scheme that we introduce in this paper and used in our protocol can be used as an independent component with other non-blocking shared data object implementations; we are currently looking at it.

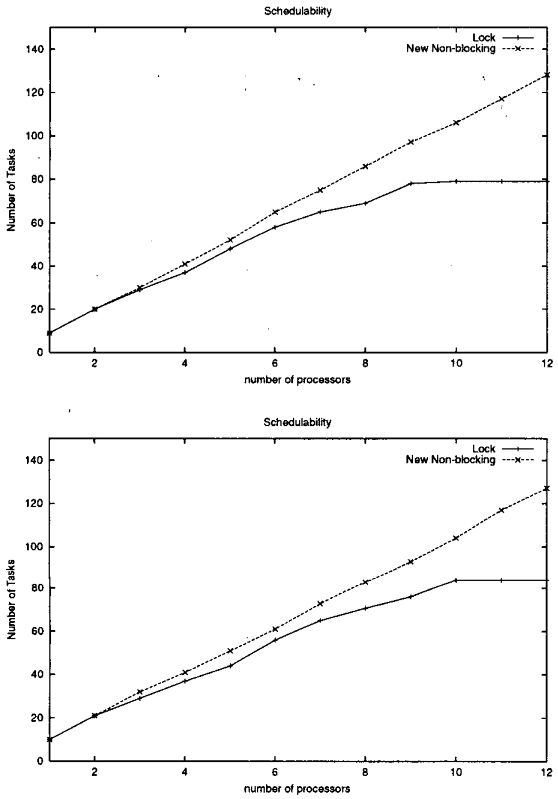

Figure 7. Random computing time schedule experiment

References

[1] Maurice P. Herlihy, Jeannette M. Wing Linearizability: A Correctness Condition for Atomic Objects, *TOPLAS*, 12(3), July 1990, pp. 463-492

[2] H. Kopetz, J. Reisinge The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem, *Proceedings of the Real-Time Systems Symposium*, 1993, pp. 131-137

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

[4] L. Sha and R. Rajkumar, J. P. Lehoczky Priority Inheritance Protocols: An Approach to Real-Time Synchronization *IEEE Transactions on Computers* Vol. 39, 9 (Sep.) 1990, pp. 1175-1185

[5] S. V. Adve, K. Gharachorloo Shared Memory Consistency Models: A Tutorial, *IEEE Computer*, Dec. 1996, pp. 66-76

[6] H. R. Simpson Four-slot fully asynchronous communication mechanism, *IEE Proceedings*, Jan. 1990, pp. 17-30

[7] J. Chen, A. Burns A Fully Asynchronous Reader/Writer Mechanism for Multiprocessor Real-Time Systems. *Technical Report YCS-288, Department of Computer Science, University of York*, May 1997.

[8] P. Sorensen, V. Hemacher A Real-Time System Design Methodology, *INFOR* 13, 1 (Feb.) 1975, pp. 1-18

[9] L. Lamport On interprocess communication, part i: basic formalism, part ii: basic algorithms, *Distributed Computing,* 1986, pp. 77-101

[10] M. Li, J. Tromp, P. Vitanyi How to share concurrent wait-free variables *Journal of the ACM*, July 1996, pp. 723-746

[11] Stuart Faulk, David Parnas On Synchronization in Hard-Real-Time Systems. *Communication of ACM*, Vol. 31, Mar. 1988, pp. 274-287

[12] A. Silberschatz, Peter B. Galvin *Operating System Concepts* , Addison Wesley, 1994.