The Non–Blocking Write Protocol NBW: A Solution to a Real–Time Synchronization Problem

Hermann Kopetz and Johannes Reisinger

Technical University of Vienna
Departement of Real-Time Systems
Vienna, Austria

Abstract

The synchronization problem between a communication controller writing a data structure into a common memory and a set of concurrently executing real–time tasks reading this data structure is investigated. Two versions of an adaptable new non–blocking protocol are presented and a schedulability analysis for a task set using these protocols is given.

1 Introduction

The design of an inter–task communication protocol for hard real time systems is a delicate challenge [2]. The well known classical solutions, i.e., the enforcement of mutual exclusion between a reader and a writer, leads to a formidable scheduling problem. The scheduling analysis of a task set where mutual exclusion between the tasks has to be considered is complex because two independent delay mechanisms interact:

- A task is delayed because it is blocked before a critical region.
- A task is delayed because it is preempted by a more urgent task.

In 1990 Sha et al. [6] have presented the priority inheritance protocol that extends the rate monotonic scheduling algorithm introduced by Liu and Layland [5] to handle task dependencies caused by mutual exclusion. The associated schedulability test for hard real–time tasks considers for every task priority the worst possible delay caused by preemption from higher priority tasks and by blocking from lower priority tasks accessing the same critical region (data structure). In the priority inheritance protocol the writer of information can be blocked before a critical region until a reader has finished its critical section.

During the design of a communication controller for our TTP protocol [4] we uncovered a synchronization scenario where it does not make sense to block a writer. Consider the case of a writer writing periodically, e.g. every millisecond, the current time into a data structure. A reader that has started to read this data structure is preempted by a higher priority task before it has finished the read operation. During this preemption time the data structure is blocked and the writer cannot update the time anymore. When the reader returns sometimes later it completes the read operation. However, what it gets is not the current time, but the time of the past interruption.

To avoid such a scenario, this paper presents a non–blocking protocol for inter–task communication in hard real–time systems and develops an associated schedulability test. In [7], Simpson presented a non-blocking asynchronous protocol for task communication between one writer and one reader, which needs a fixed amount of buffers (four) for each message. The non-blocking protocol presented in this paper supports a single writer and multiple readers. It contains a mechanism to configure the number of buffers to the application requirements, trading memory space for execution time.

The rest of this paper is organized as follows. In Section 2 we explain the architectural assumptions and introduce some properties that any solution to the stated synchronization problem must possess. Section 3 presents the protocol and develops an argument for the safety of the proposed protocol. Section 4 is devoted to the schedulability of a task set that uses this protocol. In Section 5 we introduce a refined version of this protocol, which reduces the task response time at the expense of more memory. The paper concludes with Section 6.

2 Problem Statement

2.1 Architectural Assumptions

We assume a distributed real–time system that consists of a set of nodes connected by a broadcast communication channel (Fig. 1).

Each node contains a CPU, a memory, a communication controller, and a dual–ported memory between the communication controller and the host CPU (Fig. 2). Some nodes, the interface nodes, possess an I/O interface to communicate with I/O devices in the environment.

1052-8725/93 $03.00 © 1993 IEEE

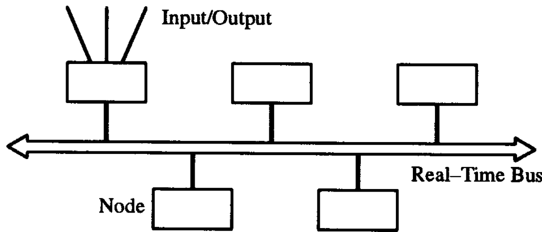


Fig. 1: Distributed Computer System

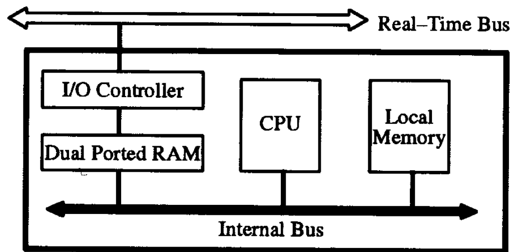


Fig. 2: Node Architecture

The communication controller receives messages from the broadcast communication channel and writes these messages into the dual–ported memory. The messages contain observations about real–time entities (e.g. temperature, pressure, etc.) in the environment. Since the validity of these observations is invalidated by the progression of real–time, the appropriate message semantics in such an application is that of a state messages [3], i.e., a new version of a message overwrites the previous version and messages are not consumed by the reader. This type of message semantics is very similar to the semantics of a variable in a programming language. In this architecture the network is thus more or less hidden behind the memory abstraction.

We assume that access to the communication channel is realized by a TDMA (time division multiple access) or a similar access protocol, where the minimal interval between two message instances concerning the same state variable in the environment is known a priori.

There is a set of concurrently executing tasks {T} that execute on the host CPU. Every one of these tasks T_i is characterized by a maximum execution time c_i, and has to be completed by a deadline d_i after its invocation. We call $l_i = d_i - c_i$ the laxity of task T_i. Since there is only one CPU for the execution of a set of real–time tasks, task preemption is supported by the real–time operating system.

2.2 The Synchronization Problem

In this paper we focus on the synchronization problem between the communication controller writing a real–time data structure into the dual ported memory and the set of asynchronously executing real–time tasks reading this data structure. This is a special form of the classical reader/writer problem. The communication controller is the writer and the tasks executing on the host computer are the readers. The specifics of this synchronization problem relate to the fact that the data structure contains real–time data. Whereas in "classical" solutions of the reader/writer problem the writer can be delayed to maintain the integrity of the data structure while it is being read, such an approach is of questionable utility if the data structure contains time dependent data.

2.3 Properties of the Solution

We are interested in solutions to this reader/writer problem that satisfy the following properties:

Safety Property

If a read operation completes successfully, it must be guaranteed that it has read an uncorrupted version of the data structure.

The interference relation of read/write operations is asymmetric, i.e., a reader does not interfere with another reader, nor does it invalidate the results of a write operation. However, a write operation can destroy other concurrent read and write operations. Since in our system there is only one writer, the safety property has to be checked for read operations only.

Timeliness Property

The tasks containing the read operations must complete their execution before their deadlines.

In a hard real–time system the execution time of all operations must be limited by a known upper bound. This is a stronger version of the classical liveness property, *that progress must be made.*

Non Blocking Property

The writer cannot be blocked by the readers.

At an abstract level, the information flow from a writer to a reader is unidirectional. If this unidirectional information–flow property can be maintained at all system levels, it is possible to add or remove readers without any change to the writer. Furthermore, no buffer space has to be provided in the communication controller. This reduces the complexity of the controller and simplifies the composition and testing of large systems.

3 The Non–Blocking Protocol

3.1 Rationale

We propose a solution to this reader/writer problem that takes advantage of the architectural characteristics described

above. Since there is only one writer to the data structure, it can write whenever it needs to write. There is no possibility that any one of the concurrently executing read operations will invalidate the results of this single writer.

The readers, however, must be more careful. They can read at any time but must check at the end of the read operation whether the writer has interfered during the execution of the read. If such an interference is observed, the reader must discard the results of the previous read operation and restart the read operation again. In the schedulability analysis it must be shown that the number of retries that can occur in the worst possible scenario is limited by a known upper bound.

3.2 Protocol Description

We associate a concurrency control field, *CCF*, with every critical data structure. *CCF* is stored in a single word of memory and is initialized with 0 during the initialization phase. We make the assumption that a single–word read or write operation of *CCF* is atomic. This atomicity has to be guaranteed by the hardware.

Before starting the write operation, the writer increments *CCF* by one. It then performs the write and after it has completed, it increments *CCF* by one again. If the value of *CCF* is increased beyond its range R (determined by the memory word length) it wraps around. We assume that R is large enough so that the maximum number of interferences during a single read operation is less than R/2.

Before starting a read operation, the reader stores the current value of *CCF* in a local variable *CCF_Begin*. It then performs the read and after it has completed it reads *CCF* again and stores it in a second local variable *CCF_End*. At this point the read phase has terminated. The reader then checks whether the writer has interfered with the read operation by inspecting the following interference condition. A writer has interfered with the reader if the value of *CCF_Begin* is different from the value of *CCF_End* or if the value of *CCF_Begin* is odd. If such an interference is observed, the reader has to restart the read operation. The detailed protocol for writing and reading a message is given in the following:

Initialization:

```
        CCFi := 0;
```

Write message i:

```
start: CCF_old := CCFi;
       CCFi := CCF_old + 1;
       <write bufi>
       CCFi := CCF_old + 2;
```

Read message i:

```
start: CCF_begin := CCFi;
       <read bufi>
       CCF_end := CCFi;
       If CCF_end ≠ CCF_begin or
          CCF_begin = odd then goto start;
```

The boolean expression within the last statement of the reader is the *interference condition*.

To increase the protocol efficiency, it is possiible to execute the second clause of the interference condition immediately after reading CCF_i.

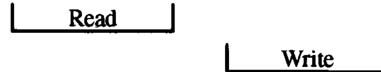
3.3 Correctness Argument for the Safety Property

In this section we present a correctness argument to show that whenever a reader terminates a read operation the safety property is maintained.

Because of the atomicity of the single–word read and write a reader either starts before or after a writer and terminates its read phase either before or after a write operation.

Only the following six temporal relations between a read operation and a write operation are therefore possible:

(1) Write after read:



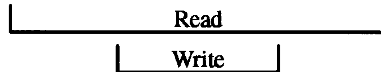
(2) Read after write:

(3) Read start before write finish:

(4) Write start before read finish:

(5) Read within write:

(6) Write within read:



In relations (1) and (2) the read and write operations are not concurrent. The interference condition evaluates to false and the read will terminate safely. In relations (3), (4), and (6) the differing values of *CCF_Begin* and *CCF_End* force a retry of the read operation. Although *CCF_Begin* and *CCF_End* are the same in relation (5) the value of *CCF_Begin* will be odd and cause the interference condition to evalu-

ate to true. Since the list of relations is exhaustive, the safety property will be maintained in all possible circumstances.

4 Schedulability Analysis

Since in the architecture discussed here the writer has its own processor (the communication controller) a preemption of the writer by a high priority reader and vice versa is not possible. As a consequence the schedulability analysis is simplified.

4.1 Timeliness Analysis

The interactions between the writer and the reader can lead to an extension of the execution time of the reader. In the following analysis we first examine the worst case extension of the execution time by a single interference and then we establish a bound on the maximum number of interferences. We will use the following notation throughout the timeliness analysis:

Attributes of messages:

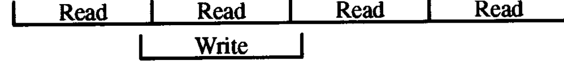
d^r ..... maximum execution time of a read operation (without retry)
d^w ..... maximum execution time of a write operation
mint ... minimum arrival interval of messages

Attributes of tasks:

c_o ..... maximum computation time of task not considering read–retries
c_n ..... maximum computation time of task considering read–retries
d ...... deadline of task
l_o ...... minimum latency of task not considering read–retries ($l_o = d - c_o$)
l_n ...... minimum latency of task considering read–retries ($l_n = d - c_n$)
N_i ..... maximum number of interferences of read operations by write operations

Single Interference

Because there is no point in time at which we can be sure that a read operation is not interfered with by a write operation, we must take into account that read operations have to be executed repeatedly until they succeed. Let us assume that the duration of read and write operations is approximately equal (that is, $d^r - \delta < d^w < d^r + \delta$ for $\delta \ll d^w$). Under this assumption, a single interference of a read operation by a write operation can cause up to three additional read operations and therefore extend the execution time of a single read operation by $3d^r$:



Number of Interferences

Due to preemptions of tasks by other tasks with higher priorities read operations may span a rather large interval of time which can lead to more than one interference of a single read operation. Each of these interferences will extend the execution time of the task by up to $3d^r$ time units. The maximum execution time of a task considering read–retries therefore computes to

$$c_n = c_o + 3N_i d^r$$

and the minimum latency is

$$l_n = l_o - 3N_i d^r.$$

We can bound the number of interferences of a read operation of a task by assuming that the chosen scheduling algorithm will guarantee that all tasks of the task set {T} will terminate before their deadlines, provided the proper maximum execution times are available. We further assume that a minimum time (*mint*) between successive write operations into the same data structure is known a priori. If such a *mint*–value cannot be established, the worst case number of interferences cannot be bounded.

As shown previously, each read operation can be interfered at least once, independent of *mint*, execution time, and latency. For our further considerations, we assume that the latency of the tasks *considering* the read–retries (l_n) is known. Thus a read operation must be preempted for an interval of at least $mint - d^w - 2d^r$ ($mint > d^w - 2d^r$ is assumed in the rest of the paper) in order to be interfered with by two subsequent write operations (see Fig. 3).

Each additional preemption interval which lasts at least $mint - d^w - 2d^r$ may lead to an additional interference. The sum of all preemption times is bounded by l_n, otherwise the task cannot be guaranteed not to miss its deadline. Therefore the maximum number of interferences computes to:

$$N_i = \left\lfloor \frac{l_n}{mint - d^w - 2d^r} \right\rfloor + 1 \Rightarrow$$

$$N_i \le \frac{l_n}{mint - d^w - 2d^r} + 1 \Rightarrow$$

$$N_i \le \frac{l_o - 3N_i d^r}{mint - d^w - 2d^r} + 1 \Rightarrow$$

$$(N_i - 1)(mint - d^w - 2d^r) \le l_o - 3N_i d^r \Rightarrow$$

$$N_i mint + N_i(d^r - d^w) \le l_o + mint - d^w - 2d^r \Rightarrow$$

$$N_i \le \frac{l_o + mint - d^w - 2d^r}{mint + d^r - d^w} \Rightarrow (d^r \approx d^w = d^{rw})$$

$$N_i \le \frac{l_o + mint - 3d^{rw}}{mint} \Rightarrow$$

$$N_i \le \left\lfloor \frac{l_o + mint - 3d^{rw}}{mint} \right\rfloor$$

The maximum execution time of the task considering read–retries computes to (we can use = instead of ≤ because c_n is an upper bound for the execution time of the task):

$$c_n \;=\; c_o + 3d^{rw}\left\lfloor \frac{l_o + mint - 3d^{rw}}{mint} \right\rfloor$$

We must further check that $2N_i < R$ (the range of the concurrency control field).

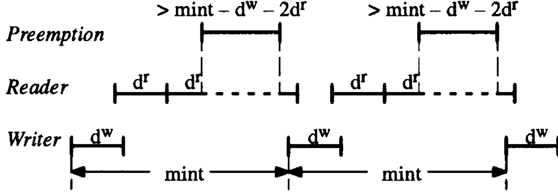


Fig. 3: Interferences of read operations

4.2 Schedulability Test

For the schedulability analysis the execution times of all tasks must be extended by the amount calculated above. Since the protocol is non–blocking, no further considerations about the consequences of the task interactions on the schedulability have to be taken into account, i.e., the task set with the extended execution times can be viewed as independent. This is important because many scheduling algorithms for scheduling a set of independent tasks are known, e.g., rate monotonic, earliest deadline, least laxity [1].

4.3 An Example

Let us take an example from the field of automotive electronics to determine the amount of the execution time extension. In this application the following parameters are realistic:

Message size:	12 bytes (6 words with a size of 16 bits)
read/write time d^{rw}:	10 µsec
Execution time c_o:	3 msec
Deadline d:	10 msec
Laxity l_o:	7 msec
mint:	2 msec

Execution time extension:

$$ete = 3d^{rw}\left\lfloor \frac{l_o + mint - 3d^{rw}}{mint} \right\rfloor =$$

$$= 3 \times 10 \times \left\lfloor \frac{7000 + 2000 - 30}{2000} \right\rfloor =$$

$$= 30 \times 4 = 120 \ \mu\text{sec}$$

The extended execution time of this task will thus be 3.12 msec, an extension of about 4% over the original execution time.

5 Extension of the Protocol

The protocol described in the previous sections suffers from two shortcomings:

- It is not possible to handle tasks with a very low laxity, especially tasks with laxity 0.
- If the sum of the read times of all messages read by a task is non-negligible compared to the execution time of the task, the protocol is rather inefficient.

Assume that d^{rw} in the above example is 200 µsec instead of 10 µsec. In this case, the execution time extension increases to 2400 µsec, increasing the execution time by 80%. In this Section we will present a method which allows the execution time overhead of tasks of a real–time system to be adapted to the specific needs of an application. Of course, this gain in execution time is not free. It has to be paid for by more memory.

This extended protocol is based on the allocation of more than one buffer for each message. These additional buffers will be used to set up periods in time in which a message is guaranteed not to change. The buffers are written to in a cyclic manner. The protocol guarantees that the reader always reads the most recent version of a message which was available when the reading procedure started.

The protocol for writing and reading messages does not differ very much from the protocol described in Section 3. The original protocol uses CCF_i only to determine an interference of a read operation by a write operation. In the extended protocol, CCF_i is used in addition to determine the number of the buffer which is accessed by the reader or the writer. In addition to CCF_i, we need a constant $bcnt_i$ which stands for the number of buffers reserved for message number i. The range R_i of CCF_i must be a multiple of $2 * bcnt_i$.

- The **writer** has to access the buffers in a cyclic manner. Because each write-operation increments CCF_i by two, using $\lfloor CCF_i/2 \rfloor \bmod bcnt_i$ as the number of the actual buffer guarantees that the buffers are accessed cyclically.
- The **reader** has to use the latest available instance of a message. $\lfloor CCF_i/2 \rfloor \bmod bcnt_i$ always specifies the buffer which is currently being written or which will be written next, therefore $(\lfloor CCF_i/2 \rfloor - 1) \bmod bcnt_i$ determines the most recent instance of a message which is not currently being written.

  An instance of a message is guaranteed not to be overwritten by a newer instance of the same message if less than $bcnt_i$ instances of the message were written in the interval [*start of read, end of read*]. This means

that the interference condition is fulfilled if and only if the value of the concurrency control field after completion of the read-operation (*CCF_end*) minus the value of the concurrency control field just after completion of writing this particular instance of the message ($2 * \lfloor CCF_begin/2 \rfloor$) is greater than the number of increments of CCF_i when writing $bcnt_i - 1$ message instances to this particular buffer ($bcnt_i * 2 - 2$).

Note: An already started, but not terminated write operation to a buffer (characterized by an odd concurrency control field) does not influence the selection of a read-buffer, therefore *CCF_begin* – 1 is the value of the concurrency control field after completion of writing the currently read instance of the message in case *CCF_begin* is odd. This fact is expressed by the arithmetic term $2*\lfloor CCF_begin/2 \rfloor$.

In the following protocol, $buf_i[j]$ denotes the j'th buffer reserved for an instance of message i (note that for $bcnt_i = 1$ this protocol is equivalent to the one presented in Section 3):

Initialization:

```
        CCFi := 0;
```

Write message i:

```
start: CCF_old := CCFi;
       CCFi := CCF_old + 1;
       <write bufi [ ⌊CCF_old/2⌋ mod bcnti ]>
       CCFi := CCF_old + 2;
```

Read message i:

```
start: CCF_begin := CCFi;
       <read bufi [ (⌊CCF_begin/2⌋ - 1)
        mod bcnti ]>
       CCF_end:= CCFi;
*      if CCF_end < CCF_begin
          then CCF_end = CCF_end + Ri;
       If CCF_end - ⌊CCF_begin/2⌋ * 2 >
          bcnti * 2 - 2
          then goto start;
```

The line marked with a * is needed because of the limited range of R_i.

5.1 Schedulability Analysis

For the schedulability analysis of the extended protocol we use the same notation as in Section 4, with the additional parameter **bcnt**, which represents the number of buffers reserved for the message.

For $bcnt > 1$, each interference causes exactly one additional read operation:

$c_n = c_o + N_i d^r$ and $l_n = l_o - N_i d^r$

A write operation to buffer i interferes with a read operation only if the read operation started when the message in buffer i was the most recent one. The worst case is therefore a read operation which started immediately before a write operation has completed and which ends after the next write operation has started (if $bcnt = 2$, otherwise $bcnt - 2$ write operations may be in between the two considered write operations). Therefore, a read operation must last at least

$$(bcnt - 1)mint - d^w$$

time units in order to cause an interference (see Fig. 4), and must thus be preempted for at least

$$(bcnt - 1)mint - d^w - d^r$$

time units (because a read operation lasts d^r time units). Each successive read operation must be preempted for at least

$$(bcnt - 1)mint - d^r$$

time units in order to cause an interference.

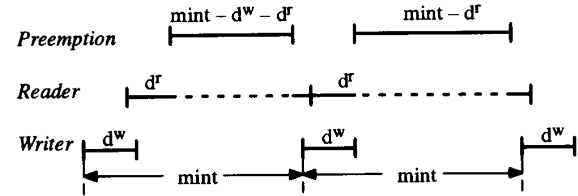


Fig. 4: Interferences of read operations (bcnt=2)

The maximum number of interferences therefore computes to:

$$N_i = \left\lfloor \frac{l_n - ((bcnt - 1)mint - d^w - d^r)}{(bcnt - 1)mint - d^r} \right\rfloor + 1 \Rightarrow$$

$$N_i \le \frac{l_n - ((bcnt - 1)mint - d^w - d^r)}{(bcnt - 1)mint - d^r} + 1 \Rightarrow$$

$$N_i \le \frac{l_o - N_i d^r - ((bcnt - 1)mint - d^w - d^r)}{(bcnt - 1)mint - d^r} + 1 \Rightarrow$$

$$(N_i - 1)((bcnt - 1)mint - d^r) \le l_o - N_i d^r - (bcnt - 1)mint + d^w + d^r \Rightarrow$$

$$N_i((bcnt - 1)mint) \le l_o + d^w \Rightarrow$$

$$N_i \le \frac{l_o + d^w}{(bcnt - 1)mint} \Rightarrow$$

$$N_i \le \left\lfloor \frac{l_o + d^w}{(bcnt - 1)mint} \right\rfloor$$

The maximum execution time of the task considering read–retries computes to (we can use = instead of ≤ because c_n is an upper bound for the execution time of the task):

$$c_n = c_o + d^r \left\lfloor \frac{l_o + d^w}{(bcnt - 1)mint} \right\rfloor$$

We must further check that $2 bcnt\ N_i < R$ (the range of the concurrency control field).

Using this new formula, the execution time extension in the example from the beginning of this section ($d^{rw} = 200$ µsec) is reduced from 2400 µsec to

$$ete = d^r \left\lfloor \frac{l_o + d^w}{(bcnt - 1)mint} \right\rfloor = 200 \times \left\lfloor \frac{7000 + 200}{1 \times 2000} \right\rfloor$$

$$= 200 \times 3 = 600 \ \mu\text{sec}$$

for $bcnt = 2$ and to

$$ete = d^r \left\lfloor \frac{l_o + d^w}{(bcnt - 1)mint} \right\rfloor = 200 \times \left\lfloor \frac{7000 + 200}{4 \times 2000} \right\rfloor$$

$$= 200 \times 0 = 0 \ \mu\text{sec}$$

for $bcnt = 5$.

6 Conclusion

We have presented two versions of a non–blocking algorithm for the communication between a communication controller and a set of asynchronously executing real–time tasks. These algorithms are characterized by a unidirectional information flow, i.e., the writer is not influenced by the activities of the readers. As a consequence the controller design is simplified since no provision for the internal buffering of information within the controller has to be made.

From the point of view of schedulability, the task set can be considered independent, provided the maximum execution time of the tasks is increased by an amount that depends on the frequency of the writer and the laxity of the reader. We have shown that this increase in the execution time is less than 10 per cent in a typical scenario from the field of automotive electronics. To handle cases in which the read/write time of a message is not negligible compared to the execution time to the task, we have presented an extension to the simple protocol which allows the execution time overhead to be reduced to any required amount of time by the use of additional space within the shared memory.

The protocols presented in the paper consider preemptive tasks. With non–preemptive tasks or with tasks, which can be preempted only at pre-determined points in time, the scenario becomes much simpler: When using one buffer, the execution time extension is $3d^r$ (because only one write operation can interfere a read operation), and when providing two buffers there is no execution time extension at all.

The transposition of the scheduling problem of a set of communicating task into a set of independent tasks is significant, because many result from the field of schedulability analysis are only applicable if the independence assumption holds. In the future we will investigate whether this transposition of a set of communicating task into a set of independent tasks by the provision of appropriate hardware support can be generalized to other architectures.

7 Acknowledgement

This work has been supported, in part, by ESPRIT Project PDCS 2, financed by the Austrian Science Foundation (FWF). Many discussions within the MARS group at the Technical University of Vienna are warmly acknowledged. We also have to thank Krithi Ramamritham, Neil Speirs, Dave Powell, and Yves Crouzet for their useful comments on earlier versions of this paper.

References

[1] S–C. Cheng, John A. Stankovic, and Krithi Ramamritham. Scheduling Algorithms for Hard Real–Time Systems — A Brief Survey. In John A. Stankovic and Krithi Ramamritham, Editors, *IEEE Tutorial on Hard Real–Time Systems*, pages 150 – 173, IEEE Computer Society Press, 1988.

[2] S.R. Faulk and D.L. Parnas. On Synchronization in Hard Real–Time Systems. *Communications of the ACM*, 31(3), March 1988.

[3] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed Fault–Tolerant Real–Time Systems: The MARS Approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[4] Hermann Kopetz and Günter Grünsteidl. TTP — A Time Triggered Protocol for Real–Time Systems. In *Proceedings of the 23rd Symposium on Fault–Tolerant Computing*, Toulouse, France, June 1993.

[5] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real–Time Environment. *Journal of the ACM*, pp. 46 – 61, February 1973.

[6] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real–Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175 – 1185, Sep. 1990.

[7] H.R. Simpson. Four–Slot Fully Asynchronous Communication Mechanism. *IEE Proceedings*, 137(1):17 – 30, January 1990.