Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information*

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers University of Technology and Göteborg University
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

A space efficient wait-free algorithm for implementing a shared buffer for real-time multiprocessor systems is presented in this paper. The commonly used method to implement shared buffers in real-time systems is based on mutual exclusion. Mutual exclusion is penalised by blocking that typically leads to difficulties in guaranteeing deadlines in real-time systems. Researchers have introduced non-blocking algorithms and data structures that address the above problems. Many of the non-blocking algorithms that appeared in the literature have very high space demands though, some even unbounded, which makes them not suitable for real-time systems. In this paper we look at a simple, elegant and easy to implement algorithm that implements a shared buffer but uses unbounded time-stamps and we show how to bound the time-stamps by using the timing information that is available in many real-time systems. Our analysis and calculations show that the algorithm resulting from our approach is space efficient. The protocol presented here can support an arbitrary number of concurrent read and write operations.

1. Introduction

In real-time systems and in distributed systems in general we have several concurrent tasks that need to communicate and synchronise in order to be able to fulfil the responsibilities of the system. There are several different means to accomplish this in a multiprocessor system. In general the tasks communicate using shared data objects. The shared data objects can be centralised or distributed, and can be accessed uniform or non-uniform. The major requirement from manufacturers when designing these shared data objects is to keep space constraints on random access memory as low as possible; in 1998, 1.5 billion 8-bit micro-controllers were sold, compared to only 63.7 million 32-bit micro-controllers [7].

In order to enforce consistency on shared data objects one commonly used method is 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 [16]; 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) [11]; and iii) leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [15]. Several synchronisation protocols have been introduced to solve the priority inversion problem for uniprocessor [15] and multiprocessor [13] systems. The solution presented in [15] solves the problem for the uniprocessor case with the cost of limiting the schedulability of task sets and also making the scheduling analysis system, where a task may be blocked by another task running on a different processor [13].

To address the problems that arise from blocking, researchers have proposed non-blocking implementations of shared data structures. Two basic non-blocking methods have been proposed in the literature, *lock-free* and *wait-free*. *Lock-free* implementations of shared data structures guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. In cases with overlapping accesses, some of them might have to repeat the operation in order to correctly complete it. This implies that there might be cases in which

[*] This work is partially funded by: i) the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council for Engineering Sciences.

the timing may cause some process(es) to have to retry a potentially unbounded number of times, leading to a for hard real-time systems unacceptable worst-case behaviour, although usually they perform well in practice. In *wait-free* implementations each task is guaranteed to *correctly* complete any operation in a *bounded* number of its own steps, regardless of overlaps and the execution speed of other processes; i.e. while the lock-free approach might allow (under very bad timing) individual processes to starve, wait-freedom strengthens the lock-free condition to ensure individual progress for every task in the system.

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, and gives significant advantages over lock-based schemes because:

1. it cannot cause priority inversion and avoids lock convoys 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 scheduling 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 different operations on 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.

Some of the wait-free protocols presented in the literature have very high space demands, some even require unbounded space, which makes them of no practical interest for real-time systems. In real-time systems usually tasks come together with their timing characteristics, such as their worst-case execution time, their period and etceteras. In this paper we look at a simple, elegant and easy to implement algorithm that implements a shared buffer but uses unbounded time-stamps and we show how to bound the time-stamps by using the timing information that is available in a real-time system. The solution allows any arbitrary number of readers and writers to concurrently access the buffer. The resulting algorithm as we show has low memory demands.

Previously Chen and Burns in [6], exploited the use of the timing information for the construction of a non-blocking shared buffer; they were the first to show how to use timing-based information to implement a fully asynchronous reader/writer mechanism; in their work they considered the case where there is only one writer. The algorithm presented in this paper allows arbitrary number of readers and writers to perform their respective operations. Research at the University of North Carolina [2, 3] and [14] by Anderson et al. has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems. Research work investigating the relation between non-blocking synchronisation and real-time systems dates back to 1974 [17, 18]. Massalin and Pu [12] and Greenwald and Cheriton [9] were the first to develop lock-free real-time kernels. Last but not least the real-time specifications of JAVA [5] include wait-free synchronisation.

The rest of this paper is organised as follows. In Section 2 we describe basic characteristics of the computer systems that we are considering together with the formal requirements that any solution to the synchronisation problem that we are considering must guarantee. In Section 3 we show how to use the timing information in order to bound the time-stamps of the unbounded protocol that is also presented in this section. Section 4 presents some examples showing the effectiveness of our results. The paper concludes with Section 5.

2. System and Problem Description

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 co-operating tasks[1] (processes) with timing constraints are 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 co-operating tasks, possibly running on different processes, use shared data objects built in the shared memory to co-ordinate and communicate. Every task 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. The shared memory may not though be uniformly accessible for all nodes

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

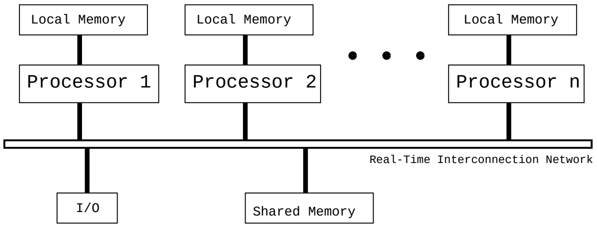


Figure 1. Shared Memory Multiprocessor System Structure

in the system. Some processors can have slower access or other restrictions like no access at all to some part of the shared memory.

2.2. The Model

In this paper we are interested in the problem of constructing an atomic shared buffer.

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 [10]. 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 reader/writer problem for real-time systems we are looking for a solution that satisfies:

- 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.

3. The Protocol

3.1. The Unbounded Algorithm

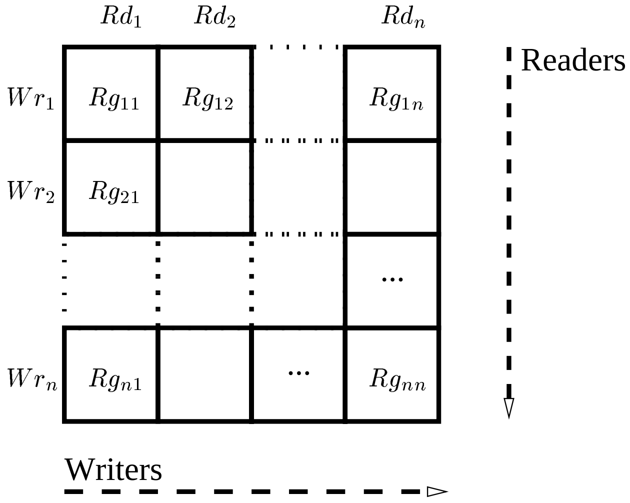


Figure 2. *Architecture of the Algorithm*

We first start with a simple, elegant and easy to implement unbounded protocol that appeared in [19]. The algorithm uses a matrix of 1-reader 1-writer registers, see figure 2. We denote the reader task and the writer task running on processor i with Rd_i and Wr_i respectively. The matrix is formed in a way such that each register Rg_{ij} can be read by processor j and written by processor i.

The algorithm originally uses unbounded time-stamps and consequently the data read from or written into each of the registers contains a data pair of a value and a tag, see figure 4. Each of the registers are read from or written to in one atomic operation. In the algorithm the tag value is unbounded. The pseudo-code for the algorithm can be viewed in figure 3. The tag value indicates the freshness of the value, a higher tag means a newer value. In this algorithm the reader reads in columns and the writers writes in rows as seen in figure 2. When the reader wants to get the latest value from the shared register it reads all the registers in its column and takes the value with the highest tag. Then it writes this value together with the tag to all registers in its row. When the writer wants to write a new value it first looks for the highest possible tag in the matrix by reading a column. Then it writes the new value in its row together with that tag value incremented by one.

```
THE UNBOUNDED ALGORITHM FOR N-READER
N-WRITER SHARED REGISTER

Reader (on processor i):
    $tag_{max}$ := 0
    for j := 1 to n
        if tag($Rg_{ji}$) > $tag_{max}$ then
            $tag_{max}$ := tag($Rg_{ji}$)
            value := value($Rg_{ji}$)
    for j := 1 to n
        $Rg_{ij}$ := (value,$tag_{max}$)
    return value

Writer (on processor i):
    $tag_{max}$ := 0
    for j := 1 to n
        if tag($Rg_{ji}$) > $tag_{max}$ then
            $tag_{max}$ := tag($Rg_{ji}$)
    for j := 1 to n
        $Rg_{ij}$ := (value,$tag_{max} + 1$)
```

Figure 3. The unbounded algorithm

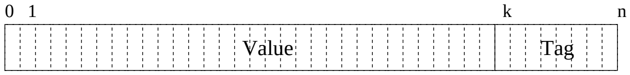


Figure 4. *The Register Structure*

The value for the tags during the execution increase rapidly and thus many bits have to be allocated for the tag value on the register to ensure there is no overflow. As the register contains a limited number of bits, this means that we have fairly few bits to allocate for the actual value. In real-time systems random access memory is also a limited resource and registers usually contain few bits. Further the system is usually required be able to run continuously for a very long period, which means that we have to allocate a lot of bits for the tag field, to ensure there is no overflow.

Let us now consider a system with eight processors and eight writer tasks, one task on each CPU. Assume that the period of each task is 10 ms, and that the tasks are interleaved in time as it is shown in figure 5. Each task starts its execution after the previous task has started to write the incremented tag to one of the registers, but not necessarily all. In this way this register will be scanned by the next writer tasks when they are scanning for the highest tag value in the system. The last writer finishes its execution before the first writer restarts its execution and the procedure repeats itself over and over. In this scenario, each invocation of a writer task will increase the tag by one, and in each period we will have increased the tag by eight. This means that in just a second of the systems execution we will have a tag value of 800, that requires 10 bits. For an hour of execution, we will have a tag value of 2 880 000, occupying 22 bits. This clearly indicates that we cannot use the algorithm as is in a real-time system with limited memory capabilities, and therefore it is of great importance to be able to bound the size of the tag field. In the next subsection we show how to use the timing information that is available in real-time systems to efficiently bound the size of the time-stamps.

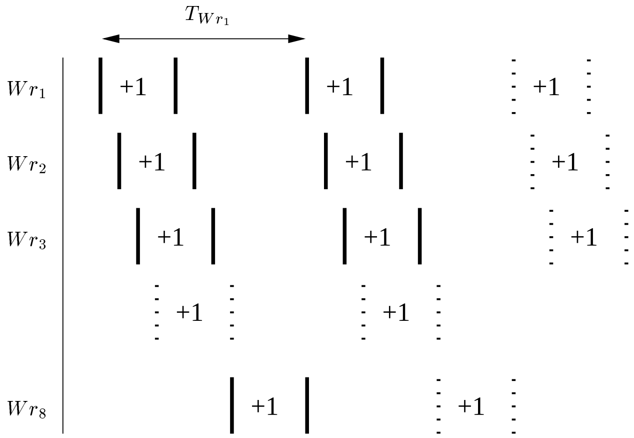


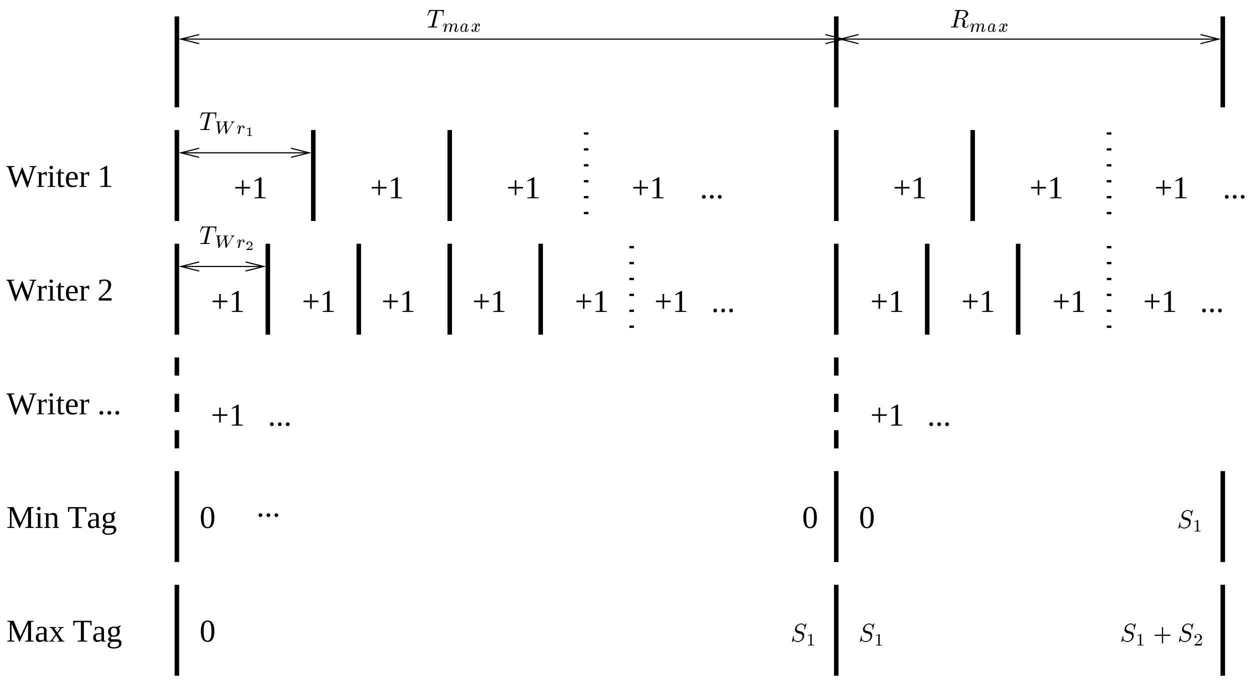
Figure 5. *Rapidly Increasing Tags*

3.2. Bounding The Time-Stamps

In this part of the paper we will see how to recycle the tags. The way that the algorithm is going to work is similar to the way the algorithm that uses unbounded time-stamps works. Namely, the writer produces a new time-stamp every time it writes, and the reader returns the value of the most recent time-stamp. The idea is to maintain a bounded number of time-stamps, and keep track of the ordering in which they were issued.

We are assuming that all tasks are periodic and that all tasks will meet their deadlines which are shorter than their respective periods. In real-time systems it is very often the case that we have very good information about the tasks.

We assume that we have n tasks in the system, indexed $t_1 \ldots t_n$. For each task t_i we will use the standard notations T_i, C_i, R_i, D_i and B_i to denote the period, worst case execution time, worst case response time, deadline and blocking time (the time the task can be delayed by lower priority tasks), respectively. Also $hp(i)$ denotes the set of tasks with higher priority than task t_i. The deadline of a task is less or equal to its period.



$S_1 = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{Wr_i}} \right\rceil$

$S_2 = \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{Wr_i}} \right\rceil$

Figure 6. *Tag Range*

For a system to be safe, no task should miss its deadlines, i.e. $\forall i \mid R_i \leq D_i$. The response time R_i for a task in the initial system can be calculated using the standard response time analysis techniques [4] as:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

In the next part of this section we are going to use the following notation, where Rd_i and Wr_i denotes the reader respective the writer tasks:

$T_{Wr_{max}} = \max_{i \in \{1..n\}} T_{Wr_i}$
$T_{Rd_{max}} = \max_{i \in \{1..n\}} T_{Rd_i}$
$T_{max} = \max\{T_{Wr_{max}}, T_{Rd_{max}}\}$

$R_{Wr_{max}} = \max_{i \in \{1..n\}} R_{Wr_i}$
$R_{Rd_{max}} = \max_{i \in \{1..n\}} R_{Rd_i}$
$R_{max} = \max\{R_{Wr_{max}}, R_{Rd_{max}}\}$

In order to bound the time-stamps we will first try to find an upper bound for $t_2 - t_1$, where t_1 is and t_2 are respectively the time stamps that a task can observe in two consecutive invocations in any possible execution of the unbounded algorithm.

Let us now take an arbitrary execution ϵ, for simplicity we assume that the tags have been initialised to 0. We are considering the worst possible scenario. Like in the previous section we assume that each writer task will increase the highest tag value by one. We are assuming for the worst case that the task with the longest period T_{max} is scanning the matrix for the highest tag in the beginning of its execution, before any other task has written any other value. This task is then suspended for almost all of its period and writes the highest tag value in its row at the very end of the task period. Thus the lowest tag will still be zero and the upper boundary for the highest tag will be:

$$S_1 = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{Wr_i}} \right\rceil$$

At the start of the execution of the task with the longest response time R_{max}, this task starts to scan for the highest value and will therefore get S_1 as the result. We are now assuming that this task is writing this tag in its row at the very end of its response time. And thus the lowest tag in the system will now be S_1. During the execution of this task, the highest tag may have been increased at most by:

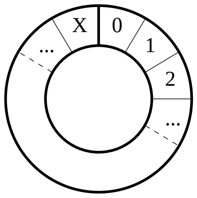


Figure 7. *Tag Value Recycling*

$S_2 = \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{Wr_i}} \right\rceil$

Therefore the highest possible tag in the system at the end of this execution of this task is:

$$MaxTag = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{Wr_i}} \right\rceil + \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{Wr_i}} \right\rceil$$

This means that during the execution that spanned from time 0 to time $T_{max} + R_{max}$ the tag values will span between zero and $S_1 + S_2$.

Lemma 1 *In any possible execution the time-stamps that two consecutive tasks can observe are going to be* $MaxTag = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{W_i}} \right\rceil + \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{W_i}} \right\rceil$ *far apart.*

The above lemma gives us a bound on the length of the "window" of active time-stamps for any task in any possible execution. In the unbounded construction the writers, by producing larger time-stamps every time they slide this window on the $[0, \dots, \infty]$ axis, always to the right. The approach now is instead of sliding this window on the set $[0, \dots, \infty]$ from left to right, to cyclically slide it on a $[0, \dots, X]$ set of consecutive natural numbers, see figure 7. Now at the same time we have to give a way to the tasks to identify the order of the different time stamps because the order of the physical numbers is not enough since we are re-using time-stamps. The idea is to use the bound that we have calculated for the span of different active time-stamps. Let us then take a task that has observed t_1 as the lowest time-stamp at some invocation τ. When this task runs again as τ', it can conclude that the active time-stamps are going to be between t_1 and $(t_1 + MaxTag) \mod X$. On the other hand we should make sure that in this interval $[t_1, \dots, (t_1 + MaxTag) \mod X]$ there are no old time-stamps. By looking closer to the previous lemma we can conclude that all the other tasks have written values to their registers with time stamps that are at most $MaxTag$ less than t_1 at the time that τ wrote the value t_1. Consequently if we use an interval that has double the size of $MaxTag$, τ' can conclude that old time-stamps are all on the interval $[(t_1 - MaxTag) \mod X, \dots, t_1]$.

Task	Period	Task	Period
Wr_1	1000	Rd_1	500
Wr_2	900	Rd_2	450
Wr_3	800	Rd_3	400
Wr_4	700	Rd_4	350
Wr_5	600	Rd_5	300
Wr_6	500	Rd_6	250
Wr_7	400	Rd_7	200
Wr_8	300	Rd_8	150

Table 1. Example task scenario on eight processors

Therefore we can use a tag field with double the size of the maximum possible value of the tag.

$$TagFieldSize = MaxTag * 2$$
$$TagFieldBits = \lceil \log_2 TagFieldSize \rceil$$

In this way τ' will be able to identify that v_1, v_2, v_3, v_4 (see figure 8) are all new values if $d_2 + d_3 < MaxTag$ and can also conclude that:

$$v_3 < v_4 < v_1 < v_2$$

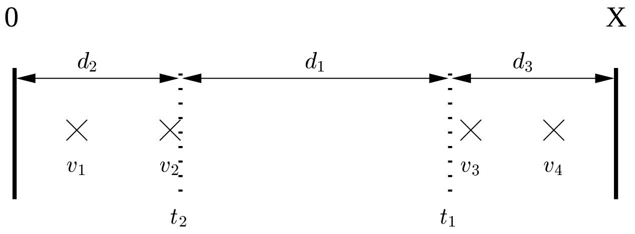


Figure 8. *Tag Reuse*

The new mechanism that will generate new tags in a cyclical order and also compare tags in order to guarantee the linearisability is presented in figure 9.

The proof for the linearisability of this construction is the same as the linearisability proof of the unbounded one.

Theorem 1 *The algorithm presented in this section implements a bounded multi-reader, multi-writer buffer that uses bounded memory space.*

4. Examples

In order to show the effectiveness of our analysis, consider the scenario described in table 1. The tasks are running on 8 processors, where writer Wr_i and reader Rd_i are executing on the same processor. We are also assuming that

COMPARISON ALGORITHM FOR BOUNDED TAG SIZE

```
$tag_{max}$ := tag($Rg_{j1}$)
for j := 1 to n
    tag := tag($Rg_{ji}$)
    if (tag > tag$_{max}$ and (tag - tag$_{max}$) $\leq$ MaxTag)
        or ( tag < tag$_{max}$
        and ( tag + TagFieldSize - tag$_{max}$ ) $\leq$ MaxTag )
    then
         tag$_{max}$ := tag
```

NEW TAG GENERATION FOR BOUNDED TAG SIZE

```
    for j := 1 to n
        $Rg_{ij}$ := (value,
           ($tag_{max} + 1$) modulo $2^{TagFieldBits}$ )
```

Figure 9. Algorithm changes for bounded tag size

the reader and writer on the same processor are executing atomically with respect to each other. All deadlines are considered to be met.

If we assume that the maximum response time is equal to the maximal task period and apply the formulas from the analysis we get:

$$T_{max} = R_{max} = 1000$$

$$MaxTag = 1 + 1 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 4 + 4 = 38$$

$$TagFieldSize = 38 * 2 = 76$$

$$TagFieldBits = \lceil \log_2 76 \rceil = 7$$

A tag field size of 7 bits is relatively low considering that we are looking at a scenario with different task periods. Using 16-bit registers for implementing the shared register we can safely use 9 bits for the actual value contents. Without bounding the tag size, we would have reached a maximum tag value of 68400 in only one hour of execution, thus needing more than 16 bits.

If we consider the scenario discussed in subsection 3.1 of this paper, where we had 8 writer tasks, the bounded version that we propose needs only 4 bits.

5. Conclusions

We have studied an algorithm for wait-free implementation of an atomic n-reader n-writer shared register. The algorithm uses unbounded time-stamps.

We have shown how to use timing information available on real-time systems to bound the time-stamps. According to our examples the modified algorithm has small space requirements.

References

[1] B. Allvin, A. Ermedahl, H. Hansson, M. Papatriantafilou, H. Sundell, Ph. Tsigas Evaluating the Performance of Wait-Free Snapshots in Real-Time Systems *SNART'99 Real Time Systems Conference*, pp. 196-207, Aug 1999.

[2] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 346-355, Dec. 1998.

[3] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 111-122, Dec. 1997.

[4] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell and A.J. Wellings Fixed Priority Preemptive Scheduling: An Historical Perspective. *Real-Time Systems*, Vol. 8, Num. 2/3, pp. 129-154, 1995.

[5] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull The Real-Time Specification for Java. *Addison Wesley*, 2000.

[6] J. Chen, A. Burns Loop-Free Asynchronous Data Sharing in Multiprocessor Real-Time Systems Based on Timing Properties *RTTAS 99*, Nov 1999.

[7] R. David, N. Merriam, and N. Tracey How Embedded Applications using an RTOS can stay within On-chip Memory Limits *Proc. of the Industrial Experience Session, the 12th Euromicro Conference on Real-Time Systems*, June 2000.

[8] A. Ermedahl, H. Hansson, M. Papatriantafilou, Ph. Tsigas Wait-free Snapshots in Real-time Systems: Algorithms and their Performance . *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA '98)*, pp. 257-266, 1998.

[9] M. Greenwald and D. Cheriton The Synergy Between Non-Blocking Synchronization and Operating System Structure. *Proceedings of the Second Symposium on Operating System Design and Implementation*, pp. 123-136, 1996.

[10] M. HERLIHY Wait-Free Synchronization. *ACM TOPLAS*, Vol. 11, No. 1, pp. 124-149, Jan. 1991.

[11] H. KOPETZ AND J. REISINGER The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem. *Proc. of the 14th Real-Time Systems Symp.*, pp. 131-137, 1993.

[12] H. MASSALIN AND C. PU A Lock-Free Multiprocessor OS Kernel. *Technical Report CUCS-005-91, Computer Science Department, Columbia University*, June 1991.

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

[14] S. RAMAMURTHY, M. MOIR, AND J. ANDERSON. Real-time object sharing with minimal support. *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pp. 233-242, May 1996.

[15] L. SHA AND R. RAJKUMAR, J. P. LEHOCZKY Priority Inheritance Protocols: An Approach to Real-Time Synchronization *IEEE Transactions on Computers* Vol. 39, 9, pp. 1175-1185, Sep. 1990.

[16] A. SILBERSCHATZ, PETER B. GALVIN Operating System Concepts . *Addison Wesley*, 1994.

[17] P. SORENSEN A Methodology for Real-Time System Development. *Ph.D. Thesis, University of Toronto*, 1974.

[18] P. SORENSEN AND V. HEMACHER A Real-Time System Design Methodology. *INFOR, Vol. 13, No. 1*, pp. 1-18, Feb. 1975.

[19] P.M.B. VITANYI, B. AWERBUCH Atomic Shared Register Access by Asynchronous Hardware *27th IEEE Annual Symposium on Foundations of Computer Science*, pp. 233-243, Oct 1986.