Technical Report no. 2003-02

Simple Wait-Free Snapshots for Real-Time Systems with Sporadic Tasks[1]

Håkan Sundell **Philippas Tsigas**

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2003

[1]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.

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2003-02
ISSN: 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2003

Abstract

A wait-free algorithm for implementing a snapshot mechanism for real-time systems is presented in this paper. Snapshot mechanisms give the means to a real-time task to read a globally consistent set of variable values while other concurrent tasks are updating them. Such a mechanism can be used to solve a variety of communication and synchronisation problems, including system monitoring and control of real-time applications. Typically, implementations of such mechanisms are based on interlocking. Interlocking protects the consistency of the shared data by allowing only one process at a time to access the data. In a real-time environment locking typically leads to difficulties in guaranteeing deadlines of high priority tasks because of the blocking. Researchers have introduced non-blocking algorithms and data structures that address the above problems. In this paper we present a simple and efficient wait-free (non-blocking) snapshot algorithm by making use of timing information that is available and necessary to the scheduler that schedules the tasks of real-time systems. Experiments on a SUN Enterprise 10000 multiprocessor system show that the algorithm that we propose here, because of its simplicity, outperforms considerably the respective wait-free snapshot algorithm that is not using the timing information.

1 Introduction

In any multiprocessing system co-operating processes share data via shared data structures. To ensure consistency of the shared data structures programs typically rely on some form of software synchronisation. In this paper we are interested in designing a shared data structure for co-operative tasks in real-time multiprocessor systems allowing processes to read a globally consistent set of variable values while other concurrent tasks are updating them.

The challenges that have to be faced in the design of inter-task communication protocols for multiprocess systems become more delicate when these systems have to support real-time computing. In real-time multiprocess systems inter-task communication protocols i) have to support sharing of data between different tasks; 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.

The classical, well-known and most simple solution when designing shared data structures enforces mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at a time to access the data. Mutual exclusion i) causes large performance degradation especially in multiprocessor systems [SilG94]; 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) [KopR93]; and iii) leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [ShaR90]. Several synchronisation protocols have been introduced to solve the priority inversion problem for uniprocessor [ShaR90] and multiprocessor [Raj90] systems. The solution presented in [ShaR90] 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 [Raj90].

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 the timing may cause some process(es) to have to retry a potentially unbounded number of times, leading to an unacceptable worst-case behaviour for hard real-time systems. However, they usually 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 can not cause priority inversion, 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.

In this paper we show how to exploit information that is part of the special nature of the real-time systems in order to design a simple snapshot algorithm with one scanner that is efficient in time and space as needed. The algorithm that we propose here outperforms significantly — due to its simplicity — the respective one not using this information [Warp99a, ErmHPT98]. Experiments on a SUN Enterprise 10000 has shown that the new construction gives 400 % better response time for the update operations for all scenarios and with 20 % better response time for all practical settings. Please notice that we have one scan task at a time and multiple concurrent update tasks per component in multiple components.

Previously Chen and Burns in [CheB99a], exploited the use of the same information for the construction of a non-blocking shared buffer. Research at the University of North Carolina [AndJJ98, AndJS97] and [RamMA96] 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. In [Warp99a, ErmHPT98] it has also been shown that wait-free methods actually can be very efficient and relatively low demanding in memory consumption. In our experimental evaluation of the protocol we compare with this solution.

The rest of this paper is organised as follows. In Section 2 we describe the computer systems that we consider and give a description of the problem. Section 3 presents our protocol and later we show how to bound the size of the buffers used in the algorithm. Section 4 shows some experiments. 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

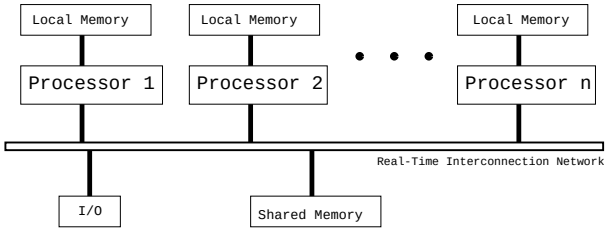

Figure 1. Shared Memory Multiprocessor System Structure

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 now, 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.

2.2 The Model

In this paper we are interested in the snapshot problem or snapshot object, which involves taking an "instantaneous" picture of a set of variables, all in one atomic operation. The snapshot is taken by one task, the *scanner*, while each of the snapshot variables may concurrently and independently be *updated* by other processes (called *updaters*). A snapshot object is also called a *composite register*, consisting of a number of *components* (indexed 1 through c), which constitute the entities which can be updated and snap-shot. We will use the two terms (snapshot object and composite register) interchangeably.

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 snapshot operation or an update 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

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

```
// Global variables
snapshotindex: integer
value[NR_COMPONENTS][∞]: valtype
// Local variables
tempindex,k,index: integer

procedure Update(cid:integer, data:valtype)
U1   value[cid][snapshotindex]:=data;

procedure Scan(snapshotdata[NR_COMPONENTS]: valtype)
S1   tempindex:=snapshotindex;
S2   snapshotindex:=tempindex+1;
S3   for k:=0 to NR_COMPONENTS-1 do
S4       for index:=tempindex to 0 step -1 do
S5           if value[k][index] ≠ NIL then
S6               snapshotdata[k]:=value[k][index];
S7               break;
```

Figure 2. Pseudocode for the Unbounded Snapshot Algorithm

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 $\rightarrow_h$that respects the specification of the object [Her91].

3 The Protocol

3.1 The unbounded version

We first start with a simple unbounded snapshot protocol that first appeared in Kirousis et.al. [KirST91a]. The protocol uses buffers of infinite length, the architecture of this protocol is shown in figure 3. The pseudo-code for the algorithm is presented in figure 2. The architecture of our unbounded construction is as follows: For each component $k = 0, \ldots, c-1$, we introduce an unbounded number of subregisters $value[k][l]$, $l = 0, \ldots, \infty$ which are written to by the updater of the corresponding component and are read by the scan function. We call these subregisters **memory locations**. The second index of each memory location $value[k][l]$ is its address (the first indicates the corresponding component). A memory location holds a value that belongs either to the set of values of the corresponding component or is a special new value denoted by **NIL**. The type of all these values is denoted by *valtype*. We call them **component values**. Initially, the subregisters $value[k][l]$ for $k = 0, \ldots, c-1$ and $l = 1, \ldots, \infty$ hold the value **NIL**, while the subregisters $value[k][0]$, $k = 0, \ldots, c-1$ hold a value from the set of values of the corresponding component. Moreover, we introduce a subregister $snapshotindex$ which holds as value an integer (a pointer to a memory location). This subregister can be written to by the scanner and can be read by all updaters. It is initialised with the value 0.

In the protocol the scanner is the controller: it is the one who determines where the updaters must write. All that an updater has to do is to write its value to the memory location forwarded by the scanner through a pointer. More specifically, the protocol works as follows: An updater first reads $snapshotindex$ and then writes its value to the memory location of the corresponding component that is pointed to by $snapshotindex$. The scanner, on the other hand, first increments $snapshotindex$ by one; stores its new value into a local variable $tempindex$ and then for each component $k = 0, \ldots, c-1$ gets the value to be returned by reading $value[k][tempindex-1], \ldots, value[k][0]$ in this order until it gets a value which is not **NIL**. The scanner, by forwarding to the updater, with its very first sub-operation, a new subregister, which it does not use again during the current snapshot, it succeeds to avoid reading values written by update operations that started after its own starting point. Moreover, the scanner, by scanning the subregisters in the reverse order from the one that they were forwarded in previous operations and by returning the first "non-empty" value, it achieves to return non overwritten values.

The snapshot protocol presented here is based on the following idea: if each scan returns for each component a value which is not overwritten (cf. figure 4(a)) and which is written by an update which started before the start of the scan (cf. figure 4(b,c)), then the solution satisfies an atomicity criterion [And93, And94, KirST91a] that enables us to argue for each component separately and hence leads to a more modular proof. For the following paragraphs and the intuitive understanding of the solution, the reader should keep in mind that the intuitive presentation of the criterion is summarised in figure 4.

3.2 Bounding the Construction

The systems that we are a looking at are real-time uniprocessor or multiprocessor systems. In these systems tasks come to the respective scheduler with a number of parameters that allow these schedulers to decide whether these tasks are schedulable.

We assume that we have n tasks in the system, indexed $t_1 \ldots t_n$. The tasks can be either periodic or sporadic. For each task t_i we will use the standard notations T_i, C_i, R_i and D_i to denote the period (i.e. $\min$ period for sporadic tasks), worst case execution time, worst case response time and deadline, respectively. The deadline of a task is less or equal to its period.

For a system to be safe, no task should miss its deadlines,

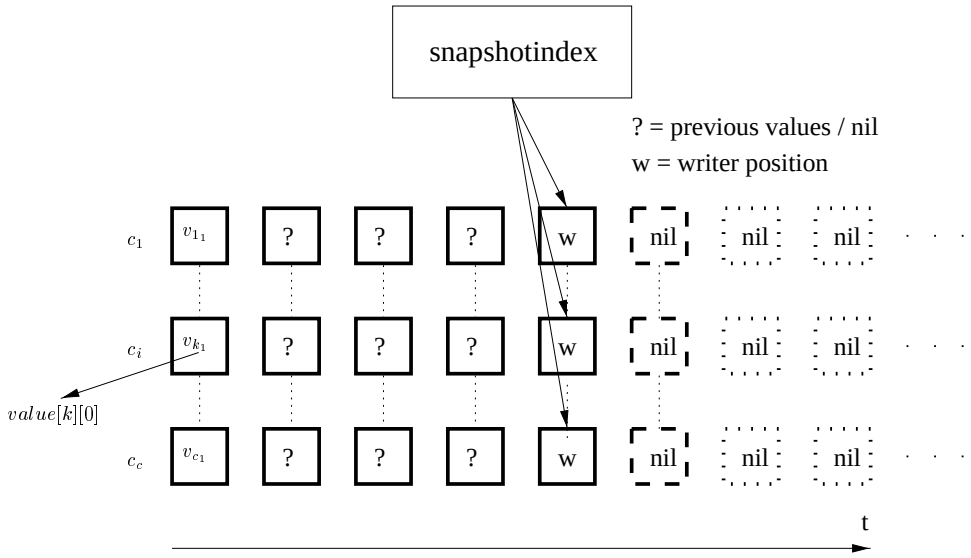

Figure 3. Unbounded Snapshot Protocol

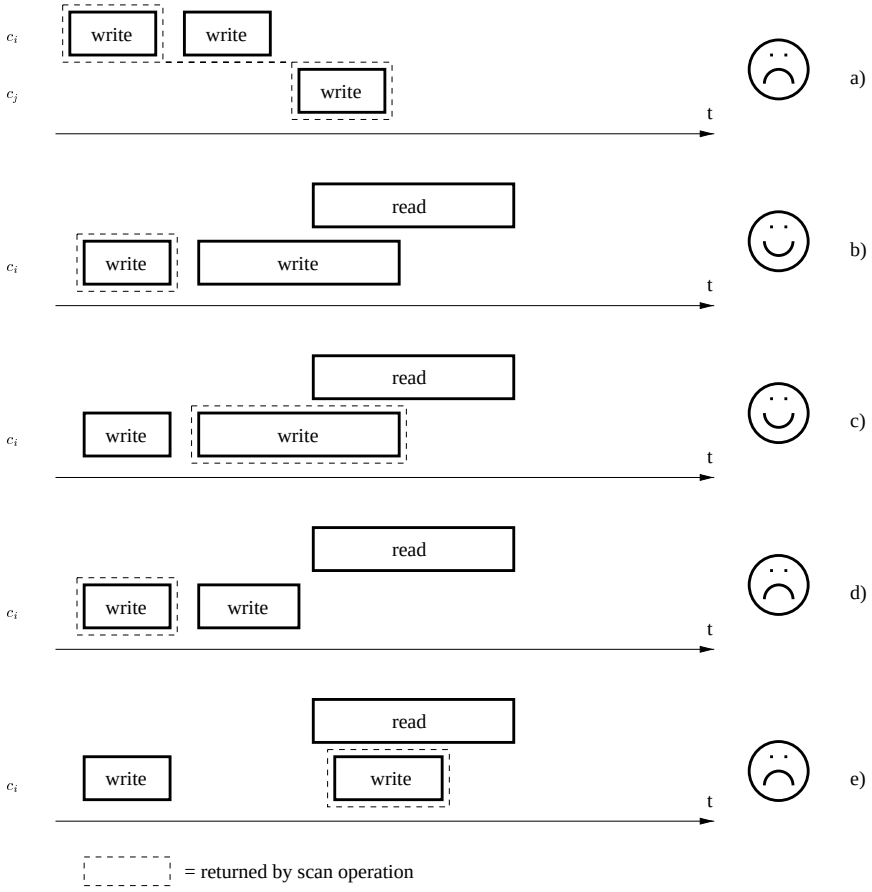

Figure 4. Intuitive presentation of the atomicity/linearisability criterion satisfied by our wait-free solution

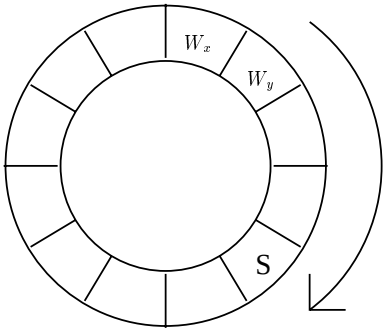

Figure 5. A cyclic buffer with several updater tasks and one snapshot task

i.e. $\forall i \mid R_i \leq D_i$.

For a system scheduled with fixed priority, the response time for a task in the initial system can be calculated using the standard response time analysis techniques [AudBDTW95]. If we with B_i denote the blocking time (the time the task can be delayed by lower priority tasks) and with $hp(i)$ denote the set of tasks with higher priority than task t_i, the response time R_i for task t_i can be formulated as:

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

The summand in the above formula gives the time that task t_i may be delayed by higher priority tasks. For systems scheduled with dynamic priorities, there are other ways to calculate the response times [AudBDTW95].

We will use T_S to denote the snapshot task period and T_{W_i} to denote the updater tasks period. To simplify the formulas we assume that tasks can be preempted at arbitrary points during their execution and that there are no overheads for context switching or interrupt handling. We also assume that one of the tasks in the system acts as a scanner task, say t_{scan}, but in the original system it doesn't have any mechanism to get a consistent snapshot.

In this subsection, we will show how to transform the unbounded space protocol of the previous subsection into one that uses bounded space only.

In the bounded space protocol as well, we are going to keep the role of the scanner as the controller of the game. It still is the one who determines the subregister where the updater is going to write. However, because the number of the subregisters must be bounded, instead of forwarding a new subregister each time, the scanner has to find an obsolete subregister which will be forwarded to the updater after erasing its contents. We call this procedure of erasing the contents of a subregister and its forwarding to the updater,

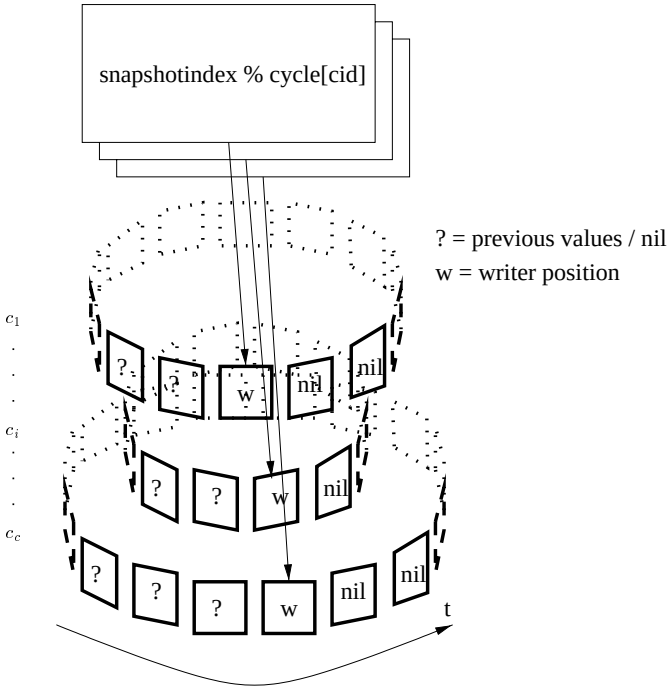

Figure 6. Bounded Snapshot Protocol

as *recycling* of the subregister.

We keep the techniques used in the previous algorithm, that is: (i) The updaters write to the memory location forwarded by the snapshot operation. (ii) The snapshot operation, by forwarding with its very first sub-operation a recycled subregister, which it is not going to use again during the current snapshot, it succeeds to avoid reading component values written by update operations which start after its own starting point. (iii) The scanner in each snapshot operation reads the remaining memory locations in the reverse order from the one that they had been previously forwarded.

Thus, the problem of designing a correct algorithm that

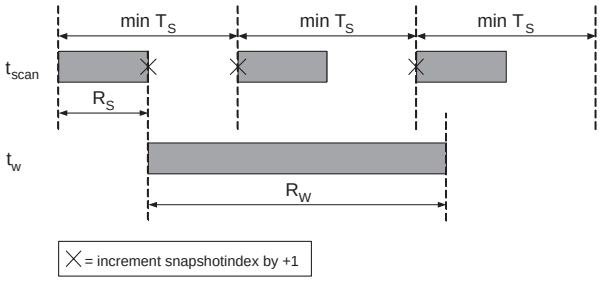

Figure 7. Estimating the buffer length - worst case scenario

```
// Global variables
snapshotindex:integer
value[NR_COMPONENTS]:pointer to valtype
cycle[NR_COMPONENTS]:integer
// Local variables
index,k,tempindex:integer

procedure Initialize
I1    for index:=0 to NR_COMPONENTS-1 do
I2        value[index]:= new valtype[cycle[index]];

procedure Update(cid:integer, data:valtype)
U1    tempindex:=snapshotindex modulo cycle[cid];
U2    value[cid][tempindex]:=data;

procedure Scan(snapshotdata[NR_COMPONENTS]:valtype)
S1    tempindex:=snapshotindex+1;
      /* clean phase */
S2    for k:=0 to NR_COMPONENTS-1 do
S3        value[k][tempindex modulo cycle[k]] := NIL;
S4    snapshotindex:=tempindex;
S5    tempindex:=tempindex-1;
      /* read phase */
S6    for k:=0 to NR_COMPONENTS-1 do
S7        index:=tempindex modulo cycle[k];
S8        while index ≠ (tempindex+1) modulo cycle[k] do
S9            if value[k][index] ≠ NIL then
S10               snapshotdata[k] := value[k][index];
S11               break;
S12           index:=(index-1) modulo cycle[k];
```

Figure 8. Pseudo-code for the Bounded Snapshot Algorithm

uses a bounded number of subregisters is reduced to the problem of having the scanner choose each time a *provably* obsolete subregister for recycling. By doing this we can use the timing information that comes together with the task set in real-time systems. For the beginning please note that the unbounded construction that was presented in the previous section has the nice property that the scanner task is always at least one position ahead of the updaters when accessing the buffers, see figure 5. This leads us to consider replacing the unbounded buffer with a cyclical buffer mechanism where the buffer slots are now going to be "forwarded" cyclically by the scanner. Each circular data buffer now is implemented by an array of l entries. Each entry is capable of holding one copy of the data that an updater wants to write. The next step then is to analyse the conditions that the cyclical buffers have to satisfy in order to maintain the safety properties that were described above. Note that the buffer length can be of different length for each individual component, and that the buffer length is dependent on the timing characteristics of the updaters that write to this component, and also dependent on the timing characteristics of the scanner task which advances the buffer index. See figure 8 for the algorithm pseudo-code and figure 6 for an explanation how the algorithm interacts with the cyclic buffers.

As the updater always has to have a valid buffer slot to write to, we know that we need a buffer of at least length one. So to calculate how many more indexes we need for each buffer we compare the maximum time it takes for an updater to finish, to the minimum time it takes between two subsequent increments of the index done by the scanner function. First we assume that the tasks always execute within their response times R with arbitrary many interruptions, and that the execution time C is comparably small. This means that the increment respective the write to the buffer slot can occur anytime within the interval for the response time. The maximum time for an updater function to finish is the same as the response time R_W for its task t_W. The minimum time between two index increments is when the first increment is executed at the end of the first interval and the next increment is executed at the very beginning of the second interval, i.e. $T_S - R_S$. The minimum time between the subsequent increments will then be the period ($\min$ for sporadic tasks) T_S. The worst case scenario between an updater and the scanner is when the snapshotindex is incremented directly after the updater has read it, as it is shown in Figure 7. Regardless of the timing characteristics of the involved tasks, it will always be possible to have at least one increment, and adding the one always needed by the updater this adds up to an absolute minimum of two buffer slots.

If $R_W \leq (\min T_S - R_S)$ then:

$$l = 2 \tag{2}$$

If $R_W > (\min T_S - R_S)$ then:

$$l = \left\lfloor \frac{R_W - (\min T_S - R_S)}{\min T_S} \right\rfloor + 3 \tag{3}$$

We are now combining those two expressions into one single expression. The floor function can safely turned into a ceiling function by subtracting one from the constant in equation 3. We also have to consider the longest buffer length we need considering all the updater tasks that are updating the same component. If we denote the length of the buffer for component k with l_k, and the group of updaters to component k with $wr(k)$, our calculations lead us to the following formula:

$$l_k = \left\lceil \frac{\max_{i \in wr(k)} R_{W_i} - \min T_S + R_S}{\min T_S} \right\rceil + 2 \tag{4}$$

Scenario	Scan Period (us)	Update Period (us)	Buffer Length
1	500	50	3
2	200	50	3
3	100	50	3
4	50	50	4
5	50	100	6
6	50	200	10
7	50	500	22

Figure 9. Descriptions of Scenarios for experiment

The last formula that we have calculated describes the buffer length that our construction needs in order to guarantee the safety property of our circular-buffer. It can be clearly seen that the buffer lengths keep very low when the snapshot task period is bigger than the updater task period, actually very similar buffer lengths as what can be achieved with more sophisticated snapshot algorithms, like the wait-free [Warp99a, ErmHPT98] that does not use the timing information but instead uses more advanced synchronisation primitives that the tasks can use in order to synchronise.

4 Experiments

A number of experiments have been performed in order to measure experimentally the performance of the new construction. To give an interesting and appropriate comparison we have done experiments with the wait-free snapshot algorithm [Warp99a, ErmHPT98], and then done the similar experiments with the bounded-time algorithm presented in this paper. The experiments have been executed on a Sun Enterprise 10000 parallel machine with 64 processors. The system considered is consisting of 1 scan process and 10 updater processes. The tasks have been generated as periodic tasks, with one task per cpu. The periods of the scan and update tasks have been changed according to some selected scenarios, see figure 9. Several long executions of the scenarios have been executed and the average response times for the scan and update operations have been measured. The wait-free respective the bounded-time algorithms have been executed with exactly the same environment and parameters. The buffer lengths have been computed according to equation 4 presented in the analysis. To give an interesting and appropriate comparison we have compared the algorithm presented here with the wait-free snapshot algorithm presented in [Warp99a, ErmHPT98]. Both these algorithms use the same unbounded memory construction, [Warp99a, ErmHPT98] bounds it efficiently without using the timing information.

The result of the experiments can be viewed in figures 10 and 11. According to the experiments, the new construction gives 400 % better response time for the update operations for all scenarios and with 20 % better response time for all scenarios that are common in practical settings compared to [Warp99a, ErmHPT98]. The protocol presented in [Warp99a, ErmHPT98] can perform better than the algorithm presented here, but only with respect to the scan operations, and only when the scan period is lower than the update period. The reason for this is that for the construction presented here, the buffer lengths increase as the period of a scan operation increases. But as we mentioned above the new construction gives 400 % better response time for the update operations for all scenarios. These are very significant results as we usually do a lot more update operations than scan operations. Although the scan operation can be slower for the bounded-time for some scenarios, we can assume that we will get a trade-off because of the benefits with the faster update operations.

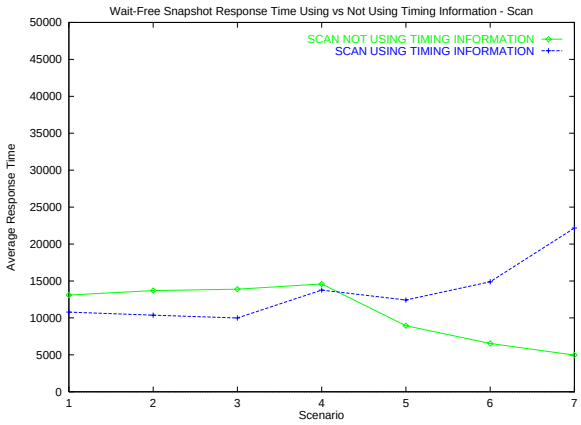

Figure 10. Experiment with 1 Scan and 10 Update processes - Scan task comparison

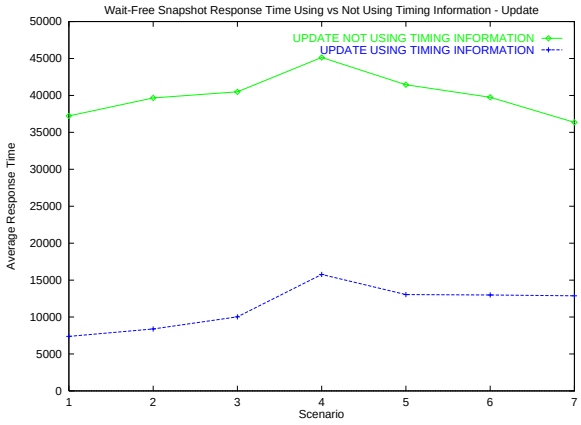

Figure 11. Experiment with 1 Scan and 10 Update processes - Update task comparison

5 Conclusions and Future Work

We have looked at the problem of taking a snapshot of several shared data components in a concurrent system by using timing information about the system that is available on real-time systems. By exploiting this information we design a simple snapshot algorithm with one scanner that is efficient in time and space. The efficiency of the algorithm was experimentally evaluated on a SUN Enterprise 10000 multiprocessor.

We believe that our construction with simple modifications works also for the multi-scanner case.

References

[Warp99a] B. Allvin, A. Ermedahl, H. Hansson, M. Papatriantafilou, H. Sundell, Ph. Tsigas. Evaluating the Performance of Wait-Free Snapshots in Real-Time Systems. In *SNART'99 Real Time Systems Conference*, pages 196–207, Aug 1999.

[And93] J. Anderson. Composite registers. In *Distributed Computing, 6*, pages 141–154, 1993.

[AndJJ98] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355. Dec. 1998.

[And94] J. Anderson. Multi-writer composite registers. In *Distributed Computing, 7*, pages 175–195, 1994.

[AndJS97] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 111–122. Dec. 1997.

[AudBDTW95] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell and A.J. Wellings. Fixed Priority Preemptive Scheduling: An Historical Perspective In *Real-Time Systems Vol. 8, Num. 2/3* , pages 129–154, 1995.

[CheB99a] J. Chen, A. Burns. Loop-Free Asynchronous Data Sharing in Multiprocessor Real-Time Systems Based on Timing Properties. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA 99)*, Nov 1999.

[ErmHPT98] A. Ermedahl, H. Hansson, M. Papatriantafilou, Ph. Tsigas. Wait-free Snapshots in Real-time Systems: Algorithms and their Performance. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 257–266, 1998.

[Her91] M. Herlihy. Wait-Free Synchronization. In *ACM TOPLAS, Vol. 11, No. 1* , pages 124–149, Jan. 1991.

[KirST91a] L.M. Kirousis, P. Spirakis and Ph. Tsigas. Reading Many Variables in One Atomic Operation: Solutions with Linear or Sublinear Complexity. In *IEEE Transactions on Parallel and Distributed Systems*, **5**(7), pages 688–696, July 1994.

[KopR93] H. Kopetz and J. Reisinger. The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem. In *Proc. of the 14th Real-Time Systems Symp.*, pages 131–137, 1993.

[Raj90] R. Rajkumar. Real-Time Synchronization Protocols for Shared Memory Multiprocessors. In *10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[RamMA96] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242. May 1996.

[ShaR90] L. Sha and R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In *IEEE Transactions on Computers, Vol. 39, 9*, pages 1175–1185, Sep. 1990.

[SilG94] A. Silberschatz, Peter B. Galvin. Operating System Concepts. Addison Wesley, 1994.