Wait-free snapshots in real-time systems: algorithms and performance

Andreas Ermedahl*
Dept. of Comp. Systems
Uppsala University
751 05 Uppsala, Sweden
E-Mail: ebbe@docs.uu.se

Hans Hansson*
Dept. of Computer Eng.
Mälardalens University
721 23 Västerås, Sweden
E-Mail: han@idt.mdh.se

Marina Papatriantafilou†, Philippas Tsigas‡
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {ptrianta, tsigas}@cs.chalmers.se

Abstract

Snap-shot mechanisms are used to read a globally consistent set of variable values. Such a mechanism can be used to solve a variety of communication and synchronization problems, including system monitoring and control of real-time applications. Methods based on locking (e.g. using semaphores) are penalized by blocking, which typically leads to difficulties in guaranteeing deadlines of high priority tasks. Lock-free methods, which take a snap-shot and then check if it corresponds to a consistent system state, have unpredictable timing-behavior, since they may have to retry an unpredictable number of times. Clearly, a method which combines the predictability of locking-based methods with the low interference (no blocking) of lock-free methods is desirable.

In this paper we present one such method, based and the concept of wait-freeness*. A wait-free method is a lock-free method which is guaranteed to correctly complete in a bounded number of steps. The price to pay for this predictability in the timing domain is the need for more than one copy of the shared objects. In addition to proving our method correct, we evaluate it analytically by formulating and comparing schedulability equations for snapshots in systems using lock-based, lock-free, and our wait-free method. We also outline ideas to study the effects of using the different snapshot methods in distributed (CAN-based) systems. Our evaluations indicate that our method is an efficient and safe alternative to traditional lock-based and lock-free methods.*

*This work is performed within the Advanced Software Technology (ASTEC) competence center, supported by the Swedish board for technical development (NUTEK)

†Partially supported by a grant of the Swedish Research Council for Engineering Sciences (TFR)

‡Part of this work has been done while this author has been a visiting researcher at the Real-Time Systems Group, Department of Computer Systems (DOCS), Uppsala University, Sweden

1. Introduction

In most systems, access to shared resources and synchronization among tasks is controlled by *locking*. Methods that provide upper bounds on the time a higher priority task has to wait for locks held by lower priority tasks (*the blocking time*) have been introduced (e.g. the priority inheritance protocol, the priority ceiling protocol, and the immediate priority ceiling protocol [7, 14, 18, 20]). The key mechanism in these protocols is to dynamically adjust priorities, thereby avoiding situations in which a high-priority task is delayed by lower priority tasks that have preempted a task holding a lock for which the high-priority task is competing, i.e. *priority inversion*. Priority inversion is a serious problem because it can make a task wait too long and miss its deadline. A recent example is the priority inversion problem in the Mars Pathfinder which caused the operating system to repeatedly reset the system.

Avoiding locking: *Wait-free* and *lock-free* interprocess communication/coordination permit access to concurrent objects without the use of locking. Therefore, they eliminate the problem of priority inversion altogether. In the *lock-free* approach, *processes* (or *tasks*)[1] access shared objects concurrently without the use of locks. 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 a for hard real-time systems unacceptable worst-case behavior. In a *wait-free* protocol 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

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

task in the system.

Intuitively, both approaches imply that, for reasons of correctness and/or time-efficiency, there may be a need both for keeping more than one copy of the shared object and to have some form of coordination among the processes to direct readers and writers to the appropriate copy. Both methods offer guarantees not only regarding efficiency, but also regarding *fault-tolerance*, as opposed to the traditional, exclusion-based methods, i.e. they avoid situations in which a process that crashed while holding a lock prevents other precesses from making progress..

Actually, the first suggestion for lock-free synchronization [12] and the wait-free approach to real-time communication were first discussed at least two decades ago [21], but was "lost" in the real-time systems community until recently, when it was revived by Kopetz and Reisinger [11], followed by a series of interesting results and the consistent research effort by Anderson et.al. (including [3]) and the more recent work by Chen and Burns [8].

The work presented in this paper addresses the *snapshot problem*, 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.

Snapshot objects are particularly useful and important tools for interprocess communication and coordination. Since they can return to the scanner a consistent global state of the system, they can provide support for decision algorithms and they can also be used to solve a variety of communication and synchronization problems, e.g., consistency checking in transaction-based systems, distributed debugging, stable property detection (deadlock, termination detection, etc.), concurrent time-stamping, system monitoring and control, including many real-time applications, such as automotive or avionics control systems [15, 17].

It should be noted that lock/wait-freedom is a desired property of snapshot solutions not only because it avoids priority inversion, but because it enables the scanner to obtain a consistent view of the system *without freezing* it.

Given its importance, the problem has been extensively studied in the literature of wait-free protocols (cf. e.g. [1, 2, 5, 10])

An implementation of a composite register (snapshot object) consists of a data structure of appropriately initialized shared variables and a set of procedures to implement the *scan* and *update operations*. The *time complexity* of an implementation is the maximum number of accesses to the shared memory per operation, while the *space complexity* is the number of shared variables needed. We measure them as a function of the number of processes which share the composite register.

The lock-free and wait-free conditions are relevant to the timing of the the implementation; the basic *correctness* condition is *linearizability* [9] (*atomicity*). This condition requires that although operations may overlap in time, their effects must be the same as the effects of some *sequential* execution, i.e. that an external observer conclude that they happen in sequence.

Our contribution: We build on previous work on the problem and on wait-free implementations with applications in real-time systems [10, 8] and we propose more efficient wait-free protocols for scanning, updating, and the use of this snapshot implementation for real-time systems appropriate for automotive/avionics monitoring/control. We make no assumptions about relative speeds of the different system parts; hence, our solutions are appropriate for completely asynchronous uniprocessor and multiprocessor systems.

Our wait-free protocols are based on the principal idea of the deterministic solutions for one scanner task in [10], which leads to a wait-free snapshot implementation without imposing more than a minimal (constant) overhead per update and per scan operation, retaining the (necessary) linear time complexity for the (single) scanner. This makes the protocol suitable for the considered application domains. To the best of our knowledge, none of the other wait-free protocols in the literature can achieve this performance for the case of a single scanner process. Inspired by the work of Chen and Burns [8], we propose an enhancement by a subprotocol that achieves multiple-phase agreement between pairs of processes, using simple boolean *TEST&SET* variables [8]. Despite the fact that these variables have been shown to have higher synchronization power than needed for solving the problem in a wait-free manner in general systems [9], as we show, by using them, we have significant gain in the memory requirements and in the time-efficiency of the solution, both of which are very important for embedded real-time systems. Moreover, these variables are always available and are not costly (cf. section 4). This enhancement enables significant savings in space (namely by $2n$ shared variables in a system with n processes) and time (namely by at least $2n$ sub-operations per scan), compared to the solution in [10].

We evaluate our algorithms analytically and we present the results of our experimental study and evaluation in a simulated real-time system appropriate for automotive/avionics monitoring/control. The system consists of

CAN-bus-connected nodes, each of which is connected to a set of devices, whose measurements activate the updates in the system via I/O controllers. The scan operations are executed by a specific controller task in the system. We perform a comparative study and evaluation among lock-based, lock-free and our wait-free snapshot. Our results suggest that our algorithms are promising, more efficient and safe alternatives to lock-based algorithms for the above and similar real-time applications, and that they do not really imply additional cost to the system, compared to the lock-based approach. Compared to the lock-free approach, our wait-free solution not only provides stronger guarantees, but it is also more predictable; i.e. retains its performance under more frequent updates, when the lock-free scan may take longer time.

2. The wait-free snapshot solution

To get better intuition on the requirement on a consistent snapshot, one must keep in mind that we do not want the scan to return inconsistent values, e.g., values written by updates that have occurred after an already scanned value has been updated. Figure 1 shows undesired behavior by a scan (i.e. part of an inconsistent snapshot). In the figure the horizontal lines represent the duration in time of the operations. If components k, l record fuel level in two fuel tanks of a vehicle and the updates report consumption the snapshot in figure 1 will erroneously lead to the conclusion that the system has more fuel left than it actually does.

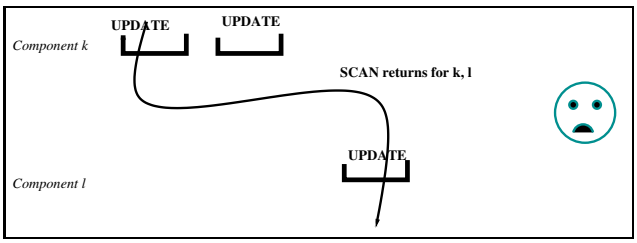


Figure 1. *An inconsistent snapshot*

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 2(c)) and which is written by an update which started before the start of the scan (cf. figure 2(a,b)), then the solution satisfies an atomicity criterion [2, 10] that enables us to argue for each component separately and hence leads to a more modular proof. The criterion and the proof that the solution satisfies are presented formally in section 5. 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 summarized in figure 2.

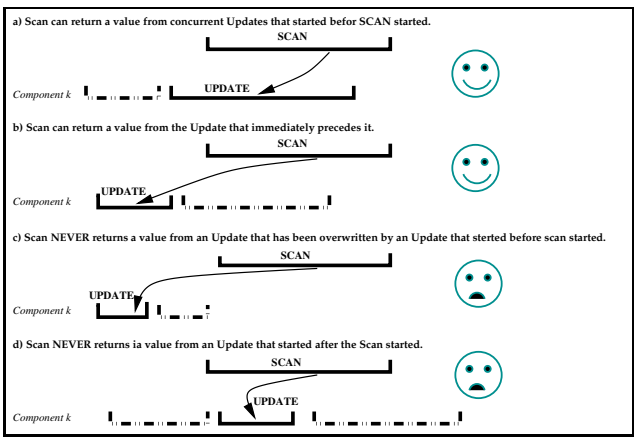


Figure 2. The atomicity/linearizability criterion satisfied by our wait-free solution

First consider the case where for each component there is only one updater (i.e. no concurrent updates in a component); this is also the case in figures 3 and 4. To guarantee the behavior required by the criterion that we informally explained above, for each component there are several shared variables (only 3 suffice, as we show later) to hold the value of the component. All that an updater has to do is to write its value where it is told to by the scanner through a pointer. With its first sub-operation in each scan, the scanner forwards a pointer for each component (array *NEXT* in figure 4) to one of these sub-registers; by not reading these sub-registers in the current scan, it achieves not to return values written by write operations which start after its own starting point (cf. fig. 2(a),(b),(d)). Moreover, by reading the remaining sub-registers in each $BUFFER[k]$ in the reverse order from the one that they were forwarded (by the previous writes of *NEXT*) in previous scans and by returning for each component k the first non-empty value, it achieves to return non-overwritten values ((cf. fig. 2(c)).

To guarantee that this will work as intended, the scanner must make sure that when it chooses a sub-register to forward, that sub-register is not about to be used by an update that started earlier and was guided to write there. Failing to do so, the scanner may miss updates. Therefore, there must be some *tracing* involved, so that a *scan*, before deciding which sub-registers will be forwarded in the next invocation of *scan* (and before "clearing" those sub-registers), knows which are the "dangerous" ones. For this, each scan plays a "tracing" game for each component, *if* it detects the start of an update operation after the last scan. The tracing game is based on a multiple-phase agreement between pairs of processes, using simple boolean *TEST&SET* variables [8]. To remember the "dangerous" sub-registers (besides the one most recently forwarded), it keeps an array

```
                    / variables shared among scanner, updater processes /
var BUFFER: array [1..c][1..3] of shared valtype;
                                        / the actual value-holders /
    NEXT: shared array [1..c] of 1..3;
                    / index variable: the scanner guides the updaters /
    PREF_SCAN, PREF_UPDATE: array [1..c] of shared 1..3;
                                / index variables for the tracing game /
    SMTU: array [1..c] of shared boolean;
                                        / Scanner Must Trace Updater /
    TS: array [1..c] of shared Test&Set boolean;

procedure update(k : 1..3, val : valtype)
                                / update procedure for component k /
    var copy_next[1..k], wptr : 1..3 ;          / local variables /
begin
    RESET(TS[k]);
    SMTU:= 1;
    copy_next:= READ(NEXT);
    WRITE(PREF_UPDATE[k] := copy_next[k]);
    if TEST&SET(TS[k]) then
        wptr := copy_next[k]; / copy_next[k] equals PREF_UPDATE[k] /
    else
        wptr := READ(PREF_SCAN[k]);
    end_if
    WRITE(BUFFER[k][wptr] := val);
end
```

Figure 3. *Wait-free snapshot (single updater per component, c components): Shared variables and* update *procedure*

prev_trace. Roughly speaking, in the kth entry of this array, the scanner keeps information about which of the three sub-registers in $BUFFER[k]$ will the most recently started update to the kth component use to write its value.

In the case where there are multiple tasks that may concurrently update a component (say, m updaters per component), the basic idea of the algorithm remains the same. The scanner, in order to keep track of their actions, *interacts with each updater separately*, while the order and the reasoning of its actions remains the same as in the single-updater case. This necessitates the extension of each data structure for each component into an array, each entry of which is to give guidance or trace information to/from a specific scanner. It also necessitates the availability of one value-holder sub-register *for each* concurrent update for each component; hence, for each component now we need the *BUFFER* array to be of dimension $m + 2$, instead of 3 which was the case for the single updater per component. It must be pointed out that, during each *scan*, a *unique* sub-register is forwarded to the updaters of each component k, as before. It is the asynchrony among the updaters that necessitates tracing each one separately, hence having situations of m "dangerous" sub-registers for a component.

```
procedure scan(): array [1..c] of valtype
    var k : 1..c;                                   / local variables /
        prep_next[1..c]: static int;
        last_read_value[1..c]: static valtype;
        order[1..c][1..3]: static 1..3;
                / scanner remembers the order to scan the subregisters /

    procedure read_registers(k : 1..c): valtype
                / auxiliary procedure to find last value for component k /
    begin
        for i = 2, ..1 do          / read component-k's, subregisters /
            tmp := READ(BUFFER[k][order[k][i]);
                        / ... in the reverse order that they were forwarded /
            if tmp ≠ nil return(tmp) end_if
        end_for
        return(last_read_value[k]);
    end

    procedure rearrange_order(k : 1..c)
    / auxil. procedure to prepare component k structure for next scan /
    begin
        {set prep_next[k] s.t. ≠ prep_next[k]∧ ≠ prev_trace[k]};
                                / prepare NEXT to forward in next scan /
        {left-rotate the values in order[k][x] , ..., order[k][3],
        where order[k][x] = prep_next[k]}
                                    / i.e. order[k][3] := prep_next[k]; /
                / also keep the order to use in read_registers next time /
    end

begin                                   / main body for procedure scan /
    WRITE(NEXT:= prep_next);
    for k = 1, . . . , c do
                                                / for each component k do /
        last_read_value[k] := read_registers(k);
        if SMTU[k] = 1 then
            / then must trace update; else prev_trace[k] remains as it is /
            SMTU[k] := 0;
            WRITE(PREF_SCAN[k] := prep_next[k]);
                                        / prep_next[k] equals NEXT[k] /
            if TEST&SET(TS[k]) then
                prev_trace[k] := prep_next[k];
                                / prep_next[k] equals PREF_SCAN[k] /
            else
                prev_trace[k] := READ(PREF_UPDATE[k]);
            end_if
        end_if
        rearrange_order(k);
        WRITE(BUFFER[k][prep_next[k]] := nil);
                                / must "clean" it before the next scan /
    end_for
    return(last_read_value);
end
```

Figure 4. *Wait-free snapshot (single updater per component, c components): The* scan *procedure*

3. Analytical evaluation

In this section we define equations to calculate the overhead that each type of snapshot implementation imposes

to the RT-system. We then analytically compare our wait-free snapshot algorithm with lock-based and lock-free implementations. We will focus on uniprocessor systems even though our wait-free algorithm also can be used, without modifications, in a multiprocessor system.

We assume that we have n tasks in the system, indexed $t_1 \ldots t_n$. For task t_i we will use the standard notations T_i, C_i, R_i, D_i and B_i to denotes 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, $lp(i)$ and $hp(i)$ denote the set of tasks with less and higher priority than task t_i, and $pri(i)$ denotes the priority of task t_i. We use $cs(i)$ to denote the set of critical sections[2] that task t_i accesses, $C_{i,s}$ to denote the worst case execution time for task t_i in critical section s, and $ceil(s)$ to denote the ceiling priority of critical section s. The ceiling priority is the highest priority of any task that may access the critical section. Finally, to estimate how much other tasks will be affected by the snapshot, we let $n_{i,op}$ denote the number of times task t_i makes the operation *op* and C_{op} denote the worst case execution time for making the operation *op*.

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 [6] as:

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

The summand in the formula gives the time that task i may be delayed by higher priority tasks. As noted before, we assume a uniprocessor system and to simplify the formulas we assume that tasks have no jitter, can be preempted at arbitrary points during their execution, has unique priorities (given in a deadline monotonic order), do not experience blocking, 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 snapshot task, say t_{snap}, but in the original system doesn't have any mechanism to get a consistent snapshot.

In the following subsections we estimate the overhead, both on t_{snap} and on the other tasks in the system, that each type of snapshot implementation will impose to the original system. As we shall see both R_i and C_i will be changed.

The Wait-Free Protocol: The cost for running our wait-free implementation of the snapshot algorithm is easily bounded. We only have to add the extra computation time to each task for performing memory accesses, (i.e., *read* or *write*) needed to determine and control where the snapshot task will be reading (see the code in figure 3 and 4). For other memory accesses no modifications are needed, but we have to add two extra buffers for each component of the composite register (snapshot object). The response time equation for *all* tasks becomes:

$$R_i = C'_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C'_j \tag{2}$$

where the execution time C_i for all tasks except the snapshot task has been extended to include the extra time to do update operations instead of write operations, i.e:

$$C'_i = C_i + n_{i,update} * (C_{update} - C_{write}) \tag{3}$$

and for the snapshot task, t_{snap}:

$$C'_i = C_i + n_{i,scan} * (C_{scan} - C_{read}) \tag{4}$$

Lock-based protocols: In a lock-based protocol, each component of the composite register (snapshot object) consists of one atomic sub-register that holds the value of the component and has a lock associated with it[3]. An update must get the lock of the component that it wants to update before writing the new value to the sub-register. A scan must get the locks of all the sub-registers (i.e. must *freeze* all the updaters) before reading the components values. During the *actual* reading time there is no overlapping write (they have to wait for the locks), hence the snapshot obtained is consistent.

As mentioned in the introduction, the use of locking must be accompanied by the use of an appropriate method to prevent priority inversion. The priority ceiling protocol (PCP) [14] and the immediate priority ceiling protocol (IPCP) [20] ensure that a task i can only be blocked (or delayed) at most by *one* critical section of any lower priority task locking a semaphore with ceiling greater than or equal to the priority of task i.

When having lock-based snapshot in the system, both PCP and IPCP have a response time formula for *all* tasks like:

$$R_i = C'_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C'_j \tag{5}$$

where

$$B_i = max_{\{\, j,s \mid j \in lp(i) \,\wedge\, s \in cs(j) \,\wedge\, ceil(s) \geq pri(i) \,\}} C_{j,s} \tag{6}$$

and

$$C'_i = C_i + n_{i,take} * C_{take} + n_{i,rel} * C_{rel} \tag{7}$$

[2] throughout this section we often abuse the term *critical section* to also denote "access to shared data" even in the lock-free and wait-free cases

[3] Another solution is to have a unique lock for all the components, but this would reduce the concurrency even further.

and $take$ and rel are the operations to take and release semaphores, respectively. The maximum blocking time, B_i, which a task i can wait for a lower priority task to execute, is calculated by investigating all tasks with lower priority than task i and all the semaphores that these tasks can lock. For those semaphores with a ceiling higher than or the same as the priority of task i, the maximum blocking time is the longest computation time a lower priority task might execute in a critical section.

In a uni-processor system we can exploit the priority structure of the tasks so that semaphore taking and releasing can be implemented using just priority changes. But still these priority changes will be costly concerning execution time when they will be implemented using operating system calls. If we implement the snapshot task as a low priority task most high priority tasks will also experience large amount of blocking.

Lock-free Protocols: A very simple lock-free snapshot implementation, with minimal overhead in each update and memory requirements is the following:

Each component of the composite register (snapshot object) consists of one atomic sub-register that holds the value of the component, as in the lock-based case. In addition, the implementation requires a boolean variable NOTE*, shared by the scanner and all the updater tasks. On each update, before writing the new value to its component, the updater makes a "note" (by writing the value 1 to* NOTE*). A scan starts to take the snapshot by resetting* NOTE *to 0 and subsequently reading the values of all the components; it then checks whether there have been overlapping updates (by checking the value of* NOTE*) and decides whether it should retry.*

For estimating the worst case response time for the snapshot task assume that the snapshot task gets preempted (by a higher priority task) when it is almost finished, i.e., the scan is invalidated just before it successfully complete.

We can observe that a high priority task which restarts the snapshot will never restart it more than once during the same instance, since it will execute until completion before the snapshot task (which executes at lower priority) can restart.

The response time formula therefore becomes:

$$R_i = C'_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C'_j \tag{8}$$

where $C'_i = C_i$ for tasks that do not access the snapshot memory, $C'_i = C_i + C_{write}$ for tasks that accesses snapshot memory and

$$C'_i = C''_i + \sum_{\exists s | j \in hp(i) \wedge s \in cs(i) \wedge s \in cs(j)} \left\lceil \frac{R_i}{T_j} \right\rceil * C''_i \tag{9}$$

where

$$C''_i = C_i + C_{write} + C_{read} + C_{compare} \tag{10}$$

for the snapshot task t_{snap}.

The summand in the outer formula (8) gives the time that the scan can be delayed by higher priority tasks and the summand in the inner formula (9) gives an upper limit on how much time the snapshot task might spend in retry loops.

We note that C'_i can become quite large for the snapshot task when we are spending time in retry loops. Each time R_i is recalculated, the worst possible computation time must also be recalculated. Actually, Equation 9 is very pessimistic, in that it for each preemption of a potentially interfering task assumes the worst-case interference (i.e., that the scan has to be restarted immediately before it successfully completes). It shall be noted that it is only tasks with priority lower than the snapshot task that will have modified response times. This should be compared to the lock-based approach, where all tasks except the ones with lowest priority are penalized.

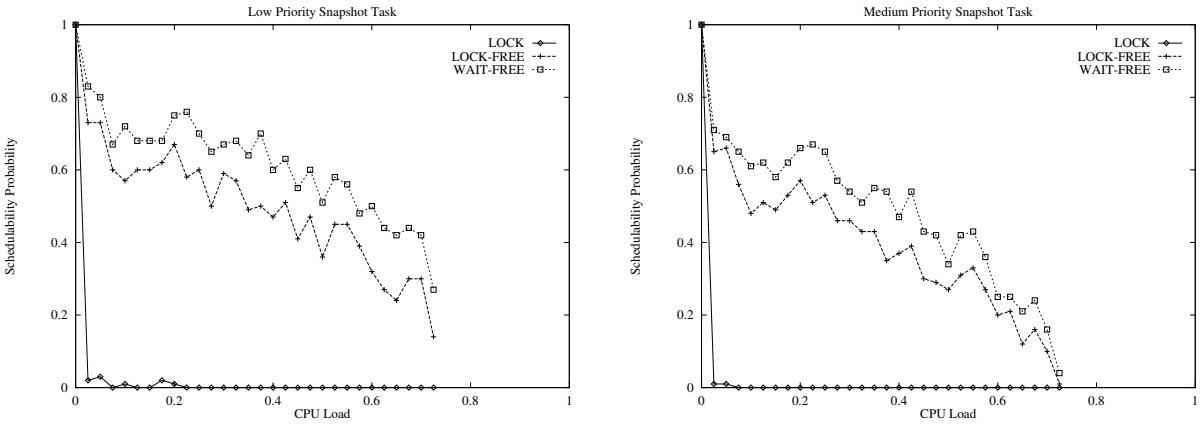
The major disadvantage with lock-free snapshot implementations is that the risk for repeated retries, especially when we are running the scan at low priority (and running it at a high priority will penalize a larger set of tasks).

Schedulability Testing: To evaluate which snapshot implementation gives highest chance for schedulability, we started by creating a system without a snapshot mechanism, but with a snapshot task t_{snap} that just reads the snapshot values without providing any guarantee that they are consistent. In the evaluation we only use task sets, including such a t_{snap} task, that are schedulable using the schedulability formula (1).

Three different systems using the three different snapshot mechanisms (lock, lock-free and wait-free) are then added and new schedulability tests are performed on each system.

We use high priority tasks with relatively short execution time (C_i) to model I/O-devices. In our evaluation, we fixed the number of tasks and I/O-devices and their fraction of the system load, but varied the total system load. The values assumed for scanning, reading, etc. were fixed assuming (pessimistically) that the wait-free operation for reading a value will take the same time as taking/releasing a semaphore. The C_i, P_i, D_i and $cs(i)$ parameters for each task were randomly generated between specified limits. The parameters used are given in figure 6. The analysis results are presented in figure 5 (a) and (b).

The result clearly indicates that for the parameters used both the wait-free and lock-free methods outperform the lock-based one. Other observations from this and related



(a) low priority snapshot task (b) medium priority snapshot task

Figure 5. *Schedulability admission tests for system with the parameters given in figure 6.*

experiments is that when the number of small tasks or the number of snapshot values increases the lock-free method deteriorates, since the chance of repeated retries increases. In the lock-based method the blocking for high priority tasks increase with the number of snapshot values, hence larger number of values gives lower chance of schedulability.

4. Experimental evaluation

We are currently performing a simulation study, in which we compare our wait-free snapshot algorithm with the lock-based and lock-free methods outlined in Section 3.

Experimental Architecture: The basic component of our experimental architecture is a micro-controller with several I/O-devices, one CPU (without cache) and RAM memory. Furthermore, we assume a real-time operating system which supports preemption, that the I/O-devices produce values by reading sensors regularly, and that they interrupt the CPU when they need I/O transfer from the device to memory. Upon an interrupt, the CPU executes the code given in a user-provided interrupt routine.

We also extend our study to the case of several nodes connected via an interconnection network, e.g., a CAN-bus, which is the network we will consider. Basically, the CAN-bus arbitration works as a priority driven scheduler, where the highest prioritized message at any of the connected nodes will be transmitted. The CAN-bus is broadcast bus on which all nodes conceptually receive all messages simultaneously. On each node, network accesses are handled by a CAN controller, which can be seen as an I/O-processor for message handling. When a message arrives, the CPU is notified by an interrupt. Tindell et.al. [22] have showed

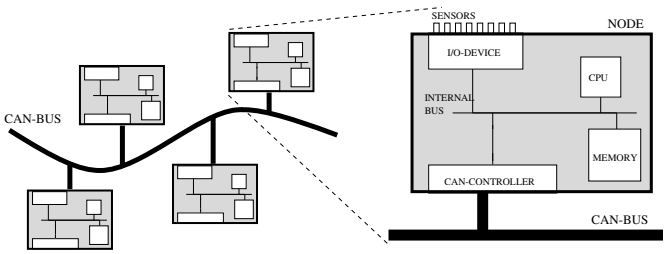


Figure 7. *A CAN-based architecture with four micro-controllers*

that the fixed priority response time equations can be easily extended for analysis of CAN-bus message delays.

We assume that the tasks in each node read values from the sensors as well as produce values that other tasks can use in their computations. The hardware supports Test&Set operations (used by the wait-free snapshot) and the operating system provides the semaphores needed for the lock-based schemes. We further assume that only the most recent sensor value can be buffered at the corresponding I/O-device, that the interrupt handlers (I/O-controllers) have priority over application tasks, and that higher priority interrupts can preempt lower priority interrupts.

Snapshot in a single node system: The simulations will be made using a discrete event simulator written in the programming language Erlang [4]. A selection of the same scenarios generated for the schedulability tests will be simulated to get an average performance. To estimate the cost introduced by adding a snapshot task, the average completion time for tasks, the maximum number of missed writes and average extra delay on other tasks due to an added snapshot task will be measured.

Fixed values			
Number of tasks	10	C_{read}, C_{write}	2
Number of devices	40	Tasks Tot CPU load	90%
$C_{semtake}, C_{semrel}$	10	Device Tot CPU load	10%
C_{scan}, C_{update}	10	Analysis per Load	100

Varying values	Min	Max		Min	Max
Task accesses to cs	1	100	Task Deadline (D_i)	20000	50000
Device accesses to cs	1	2	Device CompTime	10	20
Task CompTime (C_i)	100	2000	Device Period	50	10000
Task Period (P_i)	2000	50000	Device Deadline	500	20000

Figure 6. *Schedulability analysis parameters*

Snapshot in CAN-multiple-node systems: We will also extend our study to snapshots in multiple micro-controller CAN-bus connected systems, assuming that the scanner task is running in one of the controllers in the system and that the snapshot involves all the I/O devices and sensors in the system. When the scanner takes a snapshot it sends a snapshot-request message (with high or low priority, depending on how urgently the snapshot is needed) on the CAN-bus. This message will reach all the nodes at almost the same time and each CAN-controller will raise an interrupt to its CPU.

Below we explain how the snapshot implementations of the previous sections are made for this system. In all three approaches the update protocols are as before, and the scanner protocols start by sending a single broadcast "scan-start" message on the bus. Then the different scan protocols behave as follows:

The lock-based approach

1. on reception of the "scan-start" message, the processes responsible for the scan in each node locks all the local variables of the node components and reads and returns their values (via messages over the CAN-bus) in response to the above message
2. the scanner waits to receive responses from all nodes and then sends an "unfreeze" message on the bus
3. the process responsible for the scan in each node unlocks its components in response to the "unfreeze" message

The lock-free approach

1. on reception of the "scan-start", the process responsible for the scan in each node resets its *NOTE* variable, reads the values of its components in the node and reports them to the scanner via the bus
2. the scanner waits to hear from all nodes and then sends a new message requesting the nodes to check their *NOTE* variables
3. the process responsible for the scan in each node reports the value of its *NOTE* variable to the scanner
4. the scanner checks whether all the *NOTE* variables are 0, in which case the snapshot is complete and consistent, otherwise it repeats the scan, starting with the first step above.

The wait-free approach: In response to the "scan-start" message, the **WRITE**($NEXT :=$ *prep_next;*) is executed in each component and the process responsible for the scan in each node is activated to execute the scan on the local components, after which it reports the values to the global scanner. The difference in the initiation time between different nodes can safely be assumed (cf. CAN-behavior description earlier in this section) to be smaller than the time it takes an update to complete uninterrupted, hence the correctness of the solution is guaranteed (cf. also section 5)

5. Correctness of wait-free snapshots

The basic correctness condition for a wait-free implementation of an object is *linearizability*, i.e. although operations of concurrent processes may overlap in time, each one of them appears to have effect instantaneously, in an order that preserves the register actions' semantics.

In a global time model each operation q "occupies" a time interval $[b_q, f_q]$ on one linear time axis ($b_q < f_q$); There is a precedence relation on operations (denoted by '$\rightarrow$'), which is a strict partial order: $q_1 \rightarrow q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $\rightarrow$ are called *overlapping*. The precedence relation is extended to relate sub-operations of operations; naturally, if $q_1 \rightarrow q_2$, then for any sub-operations op_1 and op_2 of q_1 and q_2, respectively, it holds that $op_1 \rightarrow op_2$.

A *run* is an execution of an arbitrary number of operations according to the respective protocols. Given a run of a composite register implementation, a *reading function* π_k for any component k is a function that assigns an update operation u to each scan operation s, such that the value returned by s for component k —according to the *scan*operation performed— is written by u —according to the *update*operation performed. It is assumed that for each component there exists an update operation which initializes the component and precedes all other operations on it.

A run on a composite register construction is *atomic* or *linearizable*, if the partial order $\rightarrow$ on its operations can be extended to a strict *total* order $\Rightarrow$, such that for any scan s and for each component k it holds that [13]:

1. $\pi_k(s) \Rightarrow s$ and
2. there is no update u on X_k such that $\pi_k(s) \Rightarrow u \Rightarrow s$.

A construction is atomic if all its runs are atomic. When sub-registers are atomic, the precedence relation $\rightarrow$ is a total order when restricted to sub-operations on a single sub-register.

Lemma 1 [Atomicity Criterion, [2, 10]] *A construction of a single scanner composite register (snapshot object) is atomic if and only if for each component X_k, the updates on it can be serialized by a total order $\Rightarrow_k$, which is compatible with the precedence relation $\rightarrow$ and satisfies the following conditions:*

1. *Each component independently is a consistent atomic register; i.e. each $\Rightarrow_k$ and π_k satisfy all the following:*
   - (No-Irrelevant) *for each scan s, it is not the case that $s \rightarrow \pi_k(s)$*
   - (No-Old) *for each scan s there exists no update u on X_k so that $\pi_k(s) \Rightarrow_k u \rightarrow s$*
   - (No-New-Old-Inversion) *for any two scans s_1 and s_2 and for any component X_k, it is not the case that: $s_1 \rightarrow s_2$ and $\pi_k(s_2) \Rightarrow_k \pi_k(s_1)$.*
2. *For any pair of components X_k and X_l and for any scan s, it is not the case that there exist updates v and u on X_k and X_l respectively such that $\pi_k(s) \Rightarrow_k v \rightarrow u \stackrel{=}{\Rightarrow}_l \pi_l(s)$, where $u \stackrel{=}{\Rightarrow}_l \pi_l(s)$ means that either $u \Rightarrow_l \pi_l(s)$ or $u = \pi_l(s)$.*

The second condition guarantees that a scan may not return for one component a value which is very old compared to the value it returns for another component (cf. fig.1).

The proof here follows the lines of the proof of the deterministic snapshot implementation in [10]. The following lemma enables to argue about *each component separately*, which means a great simplification.

Lemma 2 *If a snapshot implementation satisfies the first condition of the atomicity criterion of lemma 1 and it also satisfies that, for every run, for each scan s and each component k, if $u = \pi_k(s)$ or $u \Rightarrow_k \pi_k(s)$ then $b_u < b_s$, then the implementation also satisfies the second condition of the atomicity criterion of lemma 1.*

Proof: Suppose, towards a contradiction, that there are updates v and a u on two components X_k and X_l, respectively, such that $\pi_k(s) \Rightarrow_k v \rightarrow u \stackrel{=}{\Rightarrow}_l \pi_l(s)$. Then, since by hypothesis $b_w < b_s$, we get that $\pi_k(s) \Rightarrow_k v \rightarrow s$, a contradiction. □

Since we have a single scanner case, we have that the scan operations are totally ordered in time, hence we can enumerate them, having $s^{[i]}$ denote the ith scan. We also define for each scan s, $tag(s) = i$ if $s = s^{[i]}$. Similarly, enumerate the instances of *NEXT*, as they become forwarded by the scan operations (via the first **WRITE** in each of them); *NEXT*$^{[i]}$ is forwarded by $s^{[i]}$. Let also, if some x = *NEXT*$^{[i]}[k]$, then $tag(x) = i$. In other words, by the enumerating the scans, we assign to each pointer to a sub-register that is being forwarded a tag, which equals the tag of the scan that forwards it (via *NEXT* and the copy that it writes in *PREF_SCAN*).

For any update u let $uptr_u$ denote the pointer to the sub-register where u writes its value. Let then $tag(u) = tag(uptr_u)$. In other words, each update u inherits the tag of the scan that forwarded the pointer to the sub-register where u writes its value.

The lemmas below prove one-by-one the conditions required by the lemma 2 for an arbitrary run r and an arbitrary component k, first for the *single* updater per component case, i.e., $\Rightarrow_k$ is actually $\rightarrow_k$.

Lemma 3 (No-Irrelevant) *for each scan s, it is not the case that $s \rightarrow \pi_k(s)$*

Proof: (outline) It is straightforward from the protocol, that a scan returns values only from updates which have completed. □

Lemma 4 *If u, u' are updates in component k and $u \rightarrow u'$, then $tag(u) \leq tag(u')$.*

Proof: (outline) Each update assigns to $wptr$ a value that it read from *NEXT*$[k]$ or from *PREF_SCAN*$[k]$. Clearly, the tag of the value that u' reads from *NEXT*$[k]$ is greater that or equal to the tag of value that u reads from *NEXT*$[k]$, since it is either the same or there has been some new scan that forwarded a new instance of *NEXT* in between u and u' read of *NEXT*. Moreover, the tag of the value that any update u gets from *PREF_SCAN*$[k]$ (if it reads *PREF_SCAN*$[k]$ at all, i.e. if it fails in the *TEST&SET* invocation) is greater that or equal to the tag of value that it reads from *NEXT*$[k]$. This is because in order that fails in the *TEST&SET* invocation, there must be an overlapping scan s which invoked the *TEST&SET*$(TS[k])$ after u reset it and before u invoked it. This scan s's *PREF_SCAN*$[k]$ has the same tag as its *NEXT*, and there can be no other scan in between that write of *PREF_SCAN*$[k]$ by s and the invocation of *TEST&SET*$(TS[k])$ by u (because of the *SMTU*$[k]$ checking), hence the claim holds. □

Lemma 5 (No-Old) *For each scan s there exists no update u on X_k so that $\pi_k(s) \rightarrow_k u \rightarrow s$.*

Proof: (outline) By contradiction, using the previous lemma, and the fact that from the scanner procedure we

have (sub-procedure *read_registers*) that the sub-registers in *BUFFER*$[k]$ are read in decreasing tag value. □

Lemma 6 *For each scan s, if $u = \pi_k(s)$ or $u \rightarrow_k \pi_k(s)$ then $b_u < b_s$.*

Proof: (outline) Suppose, towards a contradiction that exists s such that $b_{\pi_k(s)} > b_s$, i.e. that s returns a value written by an update that starts after s has started.

Since every update completes with the atomic write of its value to a place in *BUFFER*$[k]$, it holds that $f_u < f_s$, i.e. that $\pi_k(s)$ completes before s completes.

We know that the value that $\pi_k(s)$ reads for *NEXT* is the one written by s. Using similar reasoning as in lemma 4 (using the *SMTU* and *TEST&SET* tracing game between the scanner and the updater) it follows that $\pi_k(s)$ will write its value in the register pointed to by *NEXT*$[k]$ (as s wrote it and $\pi_k(s)$ read it). But s does not read *BUFFER*$[k]$[*NEXT*$[k]$], hence we have the contradiction.

Since it holds that $b_{\pi_k(s)} < b_s$, it will also hold that $b_u < b_s$ for any u that $u \rightarrow_k \pi_k(s)$. □

Lemma 7 (No-New-Old-Inversion) *for any two scans s_1 and s_2 and for any component X_k, it is not the case that: $s_1 \rightarrow s_2$ and $\pi_k(s_2) \rightarrow_k \pi_k(s_1)$.*

Proof: (outline) By contradiction, using the previous lemma. □

For the *multiple-updater-per-component case*, we need to define $\Rightarrow_k$. Define the total order among updates in one component, using their tag values, as defined above; updates with same tag write on the same sub-register, hence order them by the atomicity of that sub-operation. The proof then follows the same lines, using the defined ordering.

References

[1] Y. AFEK, H. ATTIYA, D. DOLEV, E. GAFNI, M. MERRITT, AND N. SHAVIT Atomic snapshots of shared memory. *J. Assoc. Comp. Mach.*, 40:4(1993), 873–890.

[2] J. ANDERSON Composite registers. *Distributed Computing* 6(1993), 141-154. Multi-writer comosite registers. *Distributed Computing* 7 (1994), pp. 175-195.

[3] J. ANDERSON, S. RAMAMURTHY, M. MOIR, AND K. JEFFAY Lock-Free Transactions for Real-Time Systems. *Real-Time Database Systems: Issues and Applications*, A. Bestavros, K.J. Lin, and S.H. Son, (eds.), Kluwer Academic Publishers, pp. 215-234, May 1997.

[4] J. ARMSTRONG AND R. VIRDING AND C. WIKSTRÖM AND M. WILLIAMS *Concurrent programming in Erlang*, Prentice Hall, 1996.

[5] H. ATTIYA, M. HERLIHY AND O. RACHMAN Efficient Atomic Snapshots Using Lattice Agreement. *Proc. of the 6th Int'l Workshop on Distributed Algorithms*, pp. 35-53, 1992.

[6] N.C. AUDSLEY, A. BURNS, R.I. DAVIS, K.W. TINDELL AND A.J. WELLINGS Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Systems*, Vol. 8, Num. 2/3, pp. 129-154, 1995.

[7] T. BAKER Stack-based scheduling on real-time processes. *Real-Time Systems*, 3(1), pp. 97–69, March 1991.

[8] J. CHEN AND A. BURNS Asynchronous Data Sharing in Muliprocessor Real-Time Systems Using Process Consensus. *10th Euromicro Workshop on Real-Time Systems*, 1998. also Research report, Department of Computer Science, Un. of York, 1997.

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

[10] L.M. KIROUSIS, P. SPIRAKIS AND PH. TSIGAS Reading Many Variables in One Atomic Operation: Solutions with Linear or Sublinear Complexity. *IEEE Transactions on Parallel and Distributed Systems*, **5**(7), pp. 688-696, July 1994.

[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] L. LAMPORT Concurrent Reading and Writing. *Comm. of the ACM*, Vol. 20, No. 1, Nov. 1977, pp. 806–811.

[13] L. LAMPORT (1986) On interprocess communication, part i: basic formalism, part ii: basic algorithms. *Distributed Computing* **1**, 77-101.

[14] J.P. LEHOCZKY AND L. SHA AND J.K. STROSNIDER Aperiodic Responsiveness in Hard Real-Time Environments. Proc. of the IEEE Real-Time Systems Symp., pp. 262–270, 1987.

[15] C.D. LOCKE, L. LUCAS, AND J. GOODENOUGH Generic Avionics Software Specification, Technical Report CMU/SEI-90-TR-8, Software Engineering Institute, Carnegie Mellon University, December 1990.

[16] M. PAPATRIANTAFILOU AND PH. TSIGAS Wait-Free Consensus in In-Phase Multiprocessor Systems. *Proc. of the 7th IEEE Symp. on Parallel and Distributed Processing*, pp. 312-319, 1995.

[17] C.-S. PENG, K.J. LIN, AND C. BOETTCHER Real-Time Database Benchmark Design for Avionics Systems. *Proc. of the First International Workshop on Real-Time Databases: Issues and Applications*, pp. 92-99, March 1996.

[18] R. RAJKUMAR Synchronization in Real-Time Systems – A Priority Inheritance Approach. *Kluwer Academic Publications*, 1991.

[19] S. RAMAMURTHY, M. MOIR, AND J. ANDERSON Real-Time Object Sharing with Minimal System Support. *Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing*, pp. 233-242, May 1996.

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

[21] P. SORENSEN AND V. HEMACHER A Real-Time System design Methodology. *INFOR*, 13(1), Feb 1975, pp.1–18.

[22] K.W. TINDELL AND H. HANSSON AND A.J. WELLINGS Analysing Real-Time Communications: Controller Area Network (CAN). *Proc. of IEEE RTSS'94*, pp. 259–263, 1994.