An Asynchronous Nonblocking Coordination and Synchronization Protocol for a Parallel Robotic Control Kernel[*]

Philippe Stellwag, Wolfgang Schröder-Preikschat, Daniel Lohmann
{stellwag,wosch,lohmann}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg
Computer Science 4, Martensstrasse 1
91058 Erlangen, Germany

ABSTRACT

Over the last 25 years, performance improvements by the steady increase of CPU clock frequencies were the driving factor for innovations in the domain of computationally intensive embedded applications. Now the free lunch is over [12] — developers have to parallelize their systems in order to achieve further improvements by integration of multi-core platforms. In embedded systems, this is even more challenging than in the domain of desktop computers, as safety properties and hard real-time constraints impose a much stronger demand on determinism. In this experience report, we present a concrete coordination and synchronization problem for a double buffering procedure that arose on our ongoing attempts to parallelize a robotic control kernel. This double buffering procedure used by two tasks must assure a consistent data flow without data losses. Therefore, we approach a fast bounded wait-free solution, which does not suffer from priority inversion.

1. INTRODUCTION

The doubling of the clock frequency of processors all two years [8] was and is an important factor for the development and integration of innovations (such as collision detection) into a robotic control kernel (RC). For a few years, processor manufacturers have been selling processors with more than one execution core per die; they damped the rapid increase of the clock frequency of single execution cores because of physical restrictions like power consumption or development of heat.

For the industry of robotic controls this means that 'the free lunch is *really* over' [12]. Innovations in the sector of robotic controls are driven by raw processing power; leaving out these innovations would inevitably lead to lost sales figures. Hence, the existing RC software has to be decomposed in order to be able to distribute it over multiple execution units.

[*]This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7-1.

1.1 Contribution

Previous approaches of asynchronous communication as described in Sec. 4 only assure reading and writing valid states. Thereby overwriting data, which was not fetched, and multiple reading the same data can happen, which is not acceptable in our scenario.

The subject of this paper is an asynchronous, bounded wait-free [4] coordination and synchronization protocol, based on a process consensus of two tasks. This protocol assures valid state transitions for a continuous data flow. In our domain data losses are not tolerable, as they can lead to loss of human life or damage to the environment. This restriction is typical for the sector of safety-critical real-time embedded systems.

The wait-free property of our protocol ensures that it does not suffer from priority inversion; neither does it depend on (potentially expensive) kernel objects. Furthermore, our protocol uses a bounded wait-free process consensus mechanism without the need of retry loops based on compare-and-swap (CAS). We only use test-and-set (TAS) operations, which is more deterministic than protocols based on retry loops.

1.2 Outline

The paper is organized as follows: Sec. 2 contains the problem analysis. This is followed by the description, discussion and evaluation of our asynchronous bounded wait-free protocol in Sec. 3. Related work is shown in Sec. 4. We summarize our results and experiences in Sec. 5 and use this to derive our goals for further research.

2. PROBLEM ANALYSIS

2.1 Overview

Our RC controls industrial production robots in, e.g., the automobile industry. It communicates with interfaces, such as the human machine interface (HMI), the programmable logic controller (PLC) for cyclic or acyclic I/O, and the drives for controlling the robotic axes, as illustrated in Fig. 1.

Internally, the RC consists of three tasks[1], which run

[1]There may more than three tasks, however, it is irrelevant for the scope of this paper.

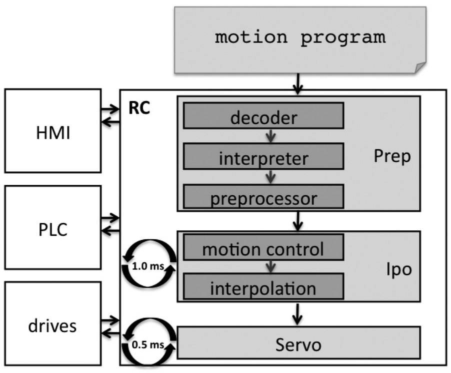

Figure 1: RC interfaces to hardware and software components

on a time-triggered preemptive real-time operating system (RTOS) with priorities.

Prep: This task decodes, interprets and prepares the motion program. The motion program defines a sequence of commands, which induces the robot to do something. The acyclic Prep task is not subject to time constraints, but has to adduce a certain flow rate to feed the Ipo task. Therefore, the Prep task has the lowest priority. The prepared data, the so-called blocks, is enqueued to a FIFO queue.

Ipo: The cyclic Ipo task is mainly responsible for motion control and interpolation of blocks, which come from the Prep task (over the FIFO queue). The Ipo task is dispatched every 1.0 ms, which must be a multiple of the servo cycle.

Servo: The Servo task does fine interpolation and feedforward/position control. This task has the highest priority in our RC. It works for one ipo cycle on the blocks that the Ipo task has produced in the previous ipo cycle. The Servo task is dispatched every 0.5 ms.

2.2 Status Quo

Communication among tasks takes place through global objects located in shared memory. There is no memory protection; hence tasks can directly access the code and data sections of other tasks.

In Fig. 2 the sequential task schedule is shown (one ipo cycle consists of two servo cycles). The Servo task is dispatched every servo cycle. As it has the highest priority, it cannot be preempted by other tasks and always runs to completion inside the servo cycle. After that, the Ipo task is dispatched from the underlying RTOS. In our example in Fig. 2, the Ipo task can be preempted by the Servo task, but it must run to completion within its ipo cycle, because the calculation results must flow back to the next Ipo run. The Prep task runs in unoccupied cycle time.

In the rest of this paper, we are looking at the *communication object* between Ipo and Servo task (also called actuators) that is a static two-slot array. The ownership structure of one slot element is for one ipo cycle attached to either the Ipo (writer or W) and Servo (reader or R) task, respectively; hence Ipo and Servo have exclusive access to their respective slot for one ipo cycle (which consists of two servo cycles in Fig. 2). The Ipo task writes the new position, velocity and acceleration setpoints to its buffer slot, which the Servo task then needs in the next ipo cycle to control the drives. New

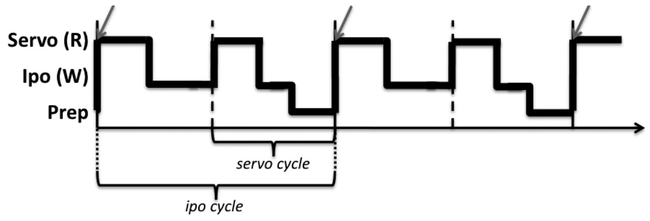

Figure 2: today's sequential task schedule

ownership of the buffer slots is negotiated in the first servo cycle relating to a ipo cycle in context of the Servo task (see arrows in Fig. 2). For our implementation of the status quo this means an index variable that representing the slot element for a task, has to be changed at the beginning of every ipo cycle, so that both actuators can decide where to read from or write into. Because the Servo task has the highest priority in our system and always runs to completion, the double buffering procedure of *turn further* this simple 'circular buffer', which is executed in context of the Servo task, is consistent for both tasks. After negotiating the ownership structure, Ipo writes for the residual ipo cycle into its slot; Servo reads from its slot, respectively.

2.3 Transition to SMP

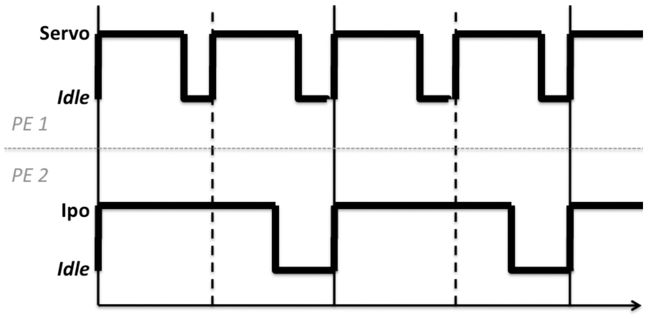

Figure 3: parallel task schedule between Ipo and Servo task

In the SMP version, both computationally intensive tasks are executed on different processing elements (PE) alias execution cores and at closer cycle times, as illustrated in Fig. 3. Ipo and Servo work for one ipo cycle on their own blocks. Now, we have to coordinate and synchronize this circular buffer so that the following criteria are fulfilled:

Correctness. This seems to be a self-evident point, but conventional locking strategies with mutexes, spinlocks or semaphores suffer from multiple problems. The incorrect use of locks can lead to unbounded priority inversion, starvation, deadlocks, livelocks and race conditions. Additionally, locks can lead to convoy effect and introduce jitter. Nonblocking lock protocols, such as the priority inheritance or priority ceiling protocol [6, 14], can induce high overhead in time and space. The priority inheritance protocol [6] cannot guarantee to be free from starvation.

Our protocol should give no reason for concurrency problems, such as deadlocks or pointer recycling. The pointer recycling problem (also known as ABA problem) comes up by using nonblocking techniques based on CAS [7, 15]. The ABA phenomenon can lead to race conditions and hence to corrupted data. Besides all that, our protocol must be priority aware.

No assumptions about timing. Assumptions about timing information (such as that the Ipo task is not triggered until Servo has defined the new ownership structure for the next ipo cycle) do not scale well with more PE and limit parallelism. Additionally, new cycles for such trival coordination points would introduce complexity even if we use a time-triggered RTOS. Furthermore, we have to keep in mind that both tasks can access their slot elements in parallel at any time.

Minimal overhead. The contention of this two-slot-array is obviously very low, as there are only two actuators accessing one of the two slot elements for one ipo cycle. The taking of a new ownership decision at the beginning of every ipo cycle must be coordinated so that both tasks come to a bilateral consensus. This fact of minimal contention gives raise to the design of a fast wait-free [4] protocol with an optimistic synchronization strategy, without using expensive locks. Minimal overhead is especially important for the Servo task, which must guarantee short cycle times.

Minimal jitter. Jitter becomes noticeable on axes movements and therefore is a critical point, even if a jitter introduced by a coordination and synchronization protocol for a circular buffer is very small. Hence, we have to monitor the jitter introduced by our protocol.

In the next Section, we present an asynchronous process consensus protocol for both actuators to decide into which slot element Ipo can write new setpoints for one ipo cycle, and Servo can read the setpoints for one ipo cycle, which Ipo has produced in the previous ipo cycle.

3. AN ASYNCHRONOUS BOUNDED WAIT-FREE PROTOCOL

3.1 Idea

The idea is that both tasks must come to an agreement about the slot which contains the latest data. Our protocol does not need any helping schemes or CAS-based retry loops typical for some wait-free protocols [13, 11, 2]. We only require atomic TAS operations, which test and conditionally write to a memory location and returns the old value.

In the following, we describe our mechanism of making a chronologically asynchronous process consensus between the Ipo and Servo tasks. The mechanism returns the slot index that the caller can use to access its slot element for one ipo cycle. Therefore, we use a shared three-slot array for both actuators. The following different roles can be adopted by every slot of the three-slot array:

Consented slot (C): Both tasks make an agreement about this slot element that holds the latest setpoints from the Ipo task. The Servo task reads only from this buffer slot, which is called the consented slot. When the Servo task has finished reading and requests new setpoints, this buffer slot becomes 'safe'.

Last written slot (LW): Here the latest setpoints from the Ipo task are located. If the Servo task asks for new setpoints at the next ipo cycle, the state of this slot element is changed to 'consented'.

Safe slot (S): This is the free slot element, where the Ipo task can write to.

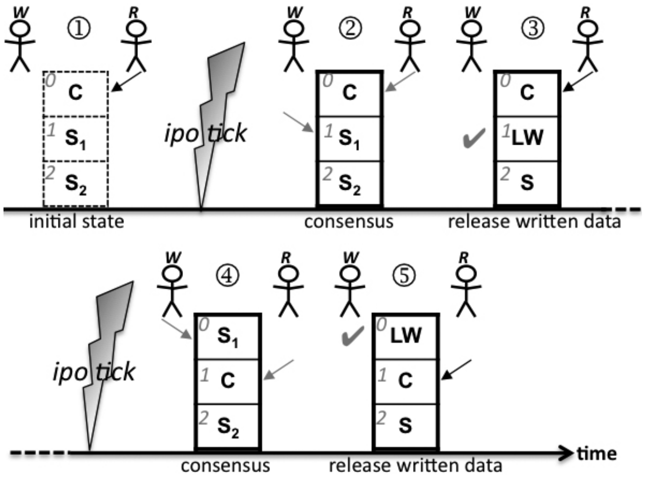

Figure 4: Consensus between Ipo and Servo task. Arrows indicate actuator's affinity to one specific slot, grey arrows symbolize a changed affinity related to the previous state.

If we talk about the state of a buffer slot, we mean the role of this slot element dependent on contained data, illustrated as letters (C, LW, S) in Fig. 4 inside the buffer slots. The arrows symbolize the affinity from actuators to a specific buffer slot. If an actuator changes its affinity to a slot element, it must also change the role of its previous slot.

The time frame for state transitions mentioned above is ensured by time-triggered tasks of the underlying RTOS. As shown in Fig. 4 there is an initial state of the C buffer slot (actually the LW slot element must be initialized so that R can decide which is the C element). The two safe buffer slots S_1 and S_2 arise out of C implicitly. In the next ipo cycle both tasks make a new asynchronous decision about the new affinity to an appropriate slot. Because there is an initial state of the C slot element at the beginning ①, it looks like R does not change the state of its affinity in ②. W chooses, e.g., the first safe slot element. After consensus[2] is made, W must release its written data at the end of this ipo cycle and hence changes the state of its slot from S_1 to LW ③. Once again, at the next ipo cycle W chooses, e.g., the first safe slot and R chooses this slot element, which contains the latest written data ④. At the end of this ipo cycle the writer actuator releases its written data ⑤. (*And so on for further ipo cycles.*)

The consensus procedure illustrated in Fig. 4 shows a scenario, where the reader consensus always happens before the writer consensus. Hence, there are two available safe slots S_1 and S_2 for the writer. If the writer consensus happens before the reader consensus, then there is only one available safe slot S for the writer, because the writer must not use C (still used by the reader) or its last written slot LW (which the reader has not fetched yet).

[2]The consensus actually is made about the C buffer slot. Both actuators must have knowledge about C to perform further steps.

3.2 Implementation

```
volatile T    Buffer[3];
volatile bool Sync = false;
volatile char LastWritten = 0;
volatile char ReaderPref/*erence*/;
volatile char WriterPref/*erence*/;
char          Consented;    /* reader's state */
```

Listing 1: shared data needed by our protocol

In the following, we take a look at the shared data needed by our protocol. Apart from the three-slot buffer, we need five control variables (see Listing 1), one synchronization bit (*Sync*) for deciding which of the two actuators has passed through the protocol first. *LastWritten* contains the slot index of the homonymous buffer slot; *ReaderPref* and *WriterPref* propose the slot index which the respective task would prefer. Finally, we need the state of the Servo task (reader) represented by the *Consented* control variable.

LW index	***C index*** **0**	**1**	**2**
0	1, 2	2	1
1	2	0, 2	0
2	1	0	0, 1

Table 1: retrieving the safe slot element

Listing 2 shows our asynchronous process consensus protocol. Both actuators call the corresponding consensus function and try to set the control variable *Sync*. If the reader task needs new data it resets *Sync*. On concurrent execution of both functions, every actuator can directly identify the consented buffer slot by using its own preference (if it was the first actuator, which sets *Sync*). As both functions return the consented buffer slot, both functions also set their preference always to the *LW* slot element. The point thereby is, that *W* can decide on basis of *C* and *LW* which is the corresponding safe slot element *S* using the static permutator array inside *writer*() (see Table 1).

After the writer has finished, it must release its slot by setting *LastWritten* to the safe slot element which it has used.

```
 1 char getWriterConsensus() {
 2   WriterPref = LastWritten;
 3   if(TAS(&Sync, true)==false)
 4     Consented = WriterPref;
 5   else Consented = ReaderPref;
 6   return Consented;
 7 }
 8
 9 char getReaderConsensus() {
10   Sync = false;
11   ReaderPref = LastWritten;
12   if(TAS(&Sync, true)==false)
13     return ReaderPref;
14   else return WriterPref;
15 }
16
17 void reader() { /* Servo task */
18   char ConsentedIdx = getReaderConsensus();
19   READ_FROM Buffer[ConsentedIdx];
20 }
21
22 void writer() { /* Ipo task */
23   static const char Permutator[3][3] = { {1,2,1},
       {2,2,0}, {1,0,0} };
24   char ConsentedIdx = getWriterConsensus();
25   char Safe = Permutator[ConsentedIdx][LastWritten];
26   WRITE_TO Buffer[Safe];
27   LastWritten = Safe;   /* release written data */
28 }
```

Listing 2: our asynchronous process consensus protocol

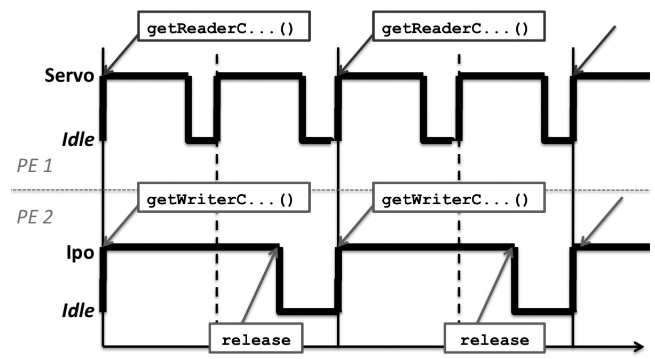

Figure 5: protocol integration in our domain

There are serveral further points of interest: Firstly, what happens if *W* executes line 5 of our protocol simultaneously while *R* executes the C-statement in line 11? At first glance, there is a potential race condition. At a second view, a race condition cannot happen, because a character variable to encode the decimal numbers 0, 1 or 2 (which are the only possible values for our control variables – apart from *Sync*) is always one byte. A store instruction can always do that atomically; that is either *R* had stored the value of *LastWritten* to *ReaderPref* before *W* executes its store instruction of line 5 or not. Both scenarios are consistent per definition of our asynchronous protocol. This statement is equally valid for concurrent execution of line 27 (release written data) and line 11.

Moreover, what happens while *R* executes line 10 and the writer actuator executes its atomic test-and-set operation using a memory barrier? Also an assignment of a boolean value to a processor register is an atomic instruction. If *W* executes its *TAS* operation *R* cannot make further progress because of the memory barrier of the *TAS* implementation. Hence no race condition can occur. It also does not matter if *R* executes line 10 before or after *W* executes line 3. The reader actuator is responsible for resetting *Sync* by the next call.

Finally, if both actuators try to execute their TAS operation in parallel, then one *TAS* will return *true* because the other actuator had set *Sync* before. Then this actuator returns the slot index, which the other actuator prefers.

3.3 Integration

The integration of our protocol into the RC is straightforward. Both, the Ipo and the Servo task must call their consensus function at the beginning of every ipo cycle, depicted in Fig. 5. Both functions return the consented index slot which is the slot element where Servo read from. The reader actuator can directly use this index to read its buffer slot. The Ipo task must perform an indirection over the static permutator array to decide which is the safe slot element. At the end of every Ipo run, Ipo must release its slot element by setting *LW* to the used buffer index.

Because Servo calls the *getReaderConsensus*() function only every second servo cycle, as illustrated in Fig. 5, two consecutive Servo runs introduce two different runtimes. But, every Servo run based on even- or odd-numbered servo cycles are strictly deterministic on C-statement level. If the Ipo task releases its buffer slot always in the second servo cycle related to one ipo cycle, as illustrated in Fig. 5, then Servo can call its *getReaderConsensus*() function every servo cycle to eliminate this kind of jitter. Such a second function call does not change the read consensus if LW did not change in the meantime. Otherwise this procedure would introduce a time restriction for the Ipo task as opposed to our goals defined in Sec. 2.3.

3.4 Evaluation

For an evaluation of our protocol, we compared it to a standard implementation using a semaphore to signal the Servo task that Ipo had produced new blocks and had set the slot index for Servo, respectively. Therefore, we additionally need a spinlock to protect the slot index, which contains the last written data for the next ipo cycle. Additionally, we measured the status quo implementation.

As hardware environment we used an octacore PC with two Xeon E5440 quadcore processors, at 2.83 GHz clock frequency, 256 KB L1 cache per core for instructions and data, 6 MB L2 cache per core pair (that is 12 MB per CPU) and 1333 MHz FSB. Today, our RC kernel running on different powerful Intel processors. Hence, we are using two of Intel's E5440 for an appropriate test enviroment, even if a Xeon processor creates too much heat to integrate it to our embedded system.

On the mentioned hardware platform we implemented a test environment on Windows XP, in which we trigger two highest priority threads on the basis of the multimedia timer with a minimal resolution of 1 ms. We distribute the two threads to different idle cores with a common 2nd level cache and validated our implementation with Intel's Thread Profiler. Only the multimedia timer and the semaphore functions P and V use corresponding system calls.

To interpret the results shown in the next Section, we have to keep in mind that the cores, which execute our periodic threads, do not have to deal with incoming IRQs, because we have set the *IntAffinity* boot option in *boot.ini* to force interrupts to the highest numbered execution core. The jitter arises from cache effects. Furthermore, it depends on the strategy how changes in the cores' caches are written back (e.g., write-through, write-back). We analyzed the influence by Windows in appropriate sequential test scenarios, which is insignificant compared to a hard real-time operating system. Furthermore, because of the out-of-order execution of the E5440 processors, we had to flush the processor pipeline before reading out RDTSC [9] to get suitable measured values. The CPUID instruction was used to flush the pipeline, which introduces some jitter ($min = 220$, $max = 284$, $\sigma = 12$, $c_v =\sim 0.05$) against what is remaining in the pipeline.

For collecting our measured values, we used the 64-bit model-specific register RDTSC. The summary of 1,000 two-way protocol calls is shown in Table 2. We measured C++ implementations of the protocols mentioned above. On the reader's side the measured values represent line 18 and on writer's side line 24–25 of Listing 2, respectively.

As shown in Table 2, the status quo implementation executes the double buffering procedure completely on Servo side. Here, the two threads run on one execution core. This implementation has a little dispersion about the mean (see coefficient of variation $c_v = \sigma/avg * 100\% =\sim 5\%$). The results also show that we have reduced the CPU cycles (based on average or median) on Servo side compared to the status quo. But, what is much more interesting on real-time systems, we also increased the cache-based outlier dramatically (see coefficient of variation c_v). On Ipo side, our protocol introduces much less jitter.

		status quo	our protocol	impl. with semaphore
Ipo side:	min	./.	416	5.955
	max	./.	1.062	39.138
	avg	./.	920	9.848
	med	./.	917	8.199
	σ	./.	61	5.449
	c_v	./.	$\sim 7\%$	$\sim 55\%$
Servo side:	min	709	305	2.430
	max	1.041	1.461	22.244
	avg	713	586	5.288
	med	709	390	4.156
	σ	34	322	4.346
	c_v	$\sim 5\%$	$\sim 55\%$	$\sim 82\%$

Table 2: runtimes of our protocol based on needed clock cycles

The standard semaphore implementation is not very efficient relating to costs for CPU cycles, priority awareness and in particular jitter. Moreover, any semaphore implementation is not leading to a fully concurrent execution of both tasks and may be expensive, because of context switches introduced by needed system calls. Furthermore, the measured values of this implementation highly depend on operating system details. This makes the results not fully comparable.

The costs in time for our protocol are really small compared to the whole runtime of the Servo and the Ipo task. The converted clock cycles in Table 2 show that on Ipo side our protocol takes $\sim$ 375 nsec. at maximum on the mentioned hardware platform, that is a rise of 100 percent. On Servo side, we have increased the costs in time from $\sim$ 368 nsec. at maximum to $\sim$ 516 nsec (rise of $\sim$ 40%). However, we have approximately doubled the utilizable processor power of the RC by cutting it into two nearly independent parts.

4. RELATED WORK

Some former work on the area of asynchronous communication from Chen and Burns [1] caches the writer consensus to improve situations, where the writer is faster than the reader. Such a situation does not happen in our field of application. Furthermore, such an approach does not work for our scenario, because if the actuators overtake each other by calling their consensus functions, it leads to Servo loosing data or Ipo overwriting data, respectively. This means, we need consistent state transitions. Chen and Burns use the state, which was last written until new data has been written. Or they overwrite data, which the reader actuator has not fetched yet. However, our protocol was inspired by their work.

A generalized $(n + 2)$-slot mechanism for n reader actuators is presented in [2] using atomic CAS instructions. The mechanism uses a process helping scheme, which induces more overhead and jitter. As mentioned, Chen and Burns [2, 1] have a different understanding of data sharing and hence their work is not suitable in the same manner for our scenario. Similarly, Peterson [10] proposes a wait-free process helping schema. In [11] Huang et al. presented a transformation mechanism that takes advantage of temporal characteristics (e.g., slower and faster reader) of the system to reduce both time and space overhead of some single-writer, multiple-reader algorithms.

Kane presented in [5] a nonblocking buffer mechanism for real-time event message communication. However, the reader must possibly check the update counter multiple times, which is modified by the writer actuator. Therefore, Kane's protocol is generally lock-free, but not wait-free.

5. SUMMARY AND FUTURE WORK

We have shown an approach to coordinate and synchronize access to a shared three-slot buffer by using an asynchronous, bounded wait-free process consensus protocol. The wait-free property ensures that our protocol does not suffer from priority inversion, because no locks are used. It is also resistant to common pitfalls of lock composition such as deadlocks, starvation or convoy effect. Furthermore, we did not use any kernel objects or operating system-dependent system calls in our protocol, hence no expensive context switches are introduced and the protocol is independent of the underlying RTOS. In Sec. 3.4 we have shown that a standard implementation with semaphore introduces much more jitter than our wait-free protocol.

The implementation and usage of our protocol is very simple compared to a standard implementation with semaphore and does not limit the parallel execution of the writer and reader task, because the protocol consensus functions can be called without presettings about timing which is asynchronous.

In our application domain of RC, buffered processing of shared data takes place in many places. Latency and jitter that arise by buffered processing on shared resources are mostly much smaller than synchronous access to shared resources. We will investigate more of the domain-specific problems that arise by parallelization of such a mid-size software project. Many nonblocking methods, in particular generic ones, such as software transactional memory, have been blamed for being only research toys [3]. Such an approach only displaces the common pitfall of resource contention in a SMP system on a lower level. Resource contention introduces latency and jitter, which is not suitable for real-time application. In the field of robotic control kernel used in industrial production robots in, e.g., automobile industry, there is no range for using such approaches, yet.

We will concentrate our research on pragmatic nonblocking coordination and synchronization methods to ensure valid and efficient data sharing in a multi-core environment for such real-time applications. In our point of view, high-grade application knowledge is needed to minimize resource contention. This leaves room for further research activities.

6. REFERENCES

[1] J. Chen; A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. *10th Euromicro Workshop on Real-Time Systems*, December 1997.

[2] J. Chen; A. Burns. A fully asynchronous reader/writer mechanism for multiprocessors real-time systems. Technical report, YCS-288, Department of Computer Science, University of York, 1997.

[3] C. Cascaval; C. Blundell; M. Michael; H. W. Cain; P. Wu; S. Chiras; S. Chatterjee. Software transactional memory: why is it only a research toy? *ACM Queue*, 6(5), September 2008.

[4] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[5] K. H. (Kane) Kim. A non-blocking buffer mechanism for real-time event message communication. *Real-Time Systems - The International Journal of Time-Critical Computing Systems*, 32(3):197–211, March 2006.

[6] R. Rajkumar; L. Sha; J. P. Lehoaky. Real-time synchronization protocols for multiprocessor. *in Proc. of Real-Time Systems Symposium*, pages 259–269, 1988.

[7] M. M. Michael. ABA prevention using single-word instructions. Technical report, IBM Research Division, RC23089 (W0401-136), January 2004.

[8] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[9] J. Muir. Using the rdtsc instruction for performance monitoring. Technical report, Intel Corporation, 1997.

[10] G. L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[11] H. Huang; P. Pillai; K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. *USENIX Annual Technical Conference*, pages 303–316, 2002.

[12] H. Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), March 2005.

[13] H. Sundell; P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. *in Proc. of RTCSA 2000 in Cheju Island (South Korea)*, December 2000.

[14] V. Yodaiken. Against priority inheritance. Technical report, FSMLabs, July 2002.

[15] P. Tsigas; Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. *in Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143, 2001.