Mechanisms for Guaranteeing Data Consistency and Flow Preservation in AUTOSAR Software on Multi-core Platforms

Haibo Zeng
General Motors R&D, haibo.zeng@gm.com

Marco Di Natale
Scuola Superiore S. Anna, marco@sssup.it

***Abstract*—The implementation of AUTOSAR runnables as a set of concurrent tasks requires the protection of shared communication and state variables implementing interface and internal ports. In addition, in a model-based design flow, the results of the model validation and verification are retained only if the code implementation preserves the semantic properties of interest. Since AUTOSAR does not support the modeling of the internal behavior of runnables, the most likely candidate for the development of the functions behavior is Simulink, which is based on a Synchronous Reactive semantics. Commercial code generation tools offer solutions for preserving the signal flows exchanged among model blocks allocated to the same core, but do not scale to multicore systems. In this paper, we summarize the possible options for the implementation of communication mechanisms that preserve signal flows, and discuss the tradeoff in the implementation of AUTOSAR models on multicore platforms.**

I. Introduction

The AUTOSAR (AUTomotive Open System ARchitecture) development partnership, which includes several OEM manufacturers, car electronics (Tier 1) suppliers, tool and software vendors, has been created to develop an open industry standard for automotive software architectures. The current version includes a reference architecture, a common software infrastructure, and specifications for the definition of components and their interfaces. The latest metamodel (version 4.0) [15] has an operational communication and synchronization semantic and has been given a timing model for the definition of event chains. However, similar to the Unified Modeling Language (UML), the AUTOSAR metamodel is mature in its static or structural part, but only offers an incomplete behavioral description which lacks a formal model of computation and a complete timed event model.

A. SW Components, Runnables, and Tasks

In AUTOSAR, the *functional architecture* of the system is a collection of *SW components* cooperating through their interfaces (Figure 1). The conceptual framework providing support for component communications is called *Virtual Functional Bus or VFB*. Component interfaces are defined as a set of ports for data-oriented or service-oriented communication. In the case of data-oriented communication, the port represents (asynchronous) access to a shared storage which one component may write into and others may read from. In the case of service-oriented communication, a client may invoke the services of a server component. These communications occur over the VFB, for which an implementation can be automatically generated depending on the placement of the components in the physical architecture. The definition of abstract components and the VFB nicely abstracts functionality from the physical architecture. The two are bound later in a process supported by tools for automatic code generation.

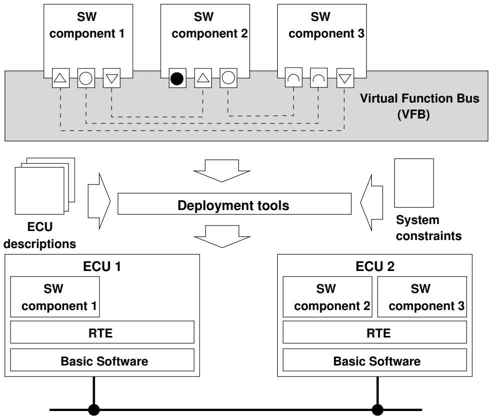

Figure 1. AUTOSAR components, interfaces and runnables.

The tools that are responsible for the automatic generation of the VFB implementation take as input the hardware platform description and placement constraints defined by the user. They produce as a result the implementation of the task model, the placement of tasks on the ECUs (Electronic Control Units), and the communication and synchronization layers. In AUTOSAR, this activity provides the configuration of the *Basic Software* (the operating system and the device drivers) and the generation of the *Run-Time Environment* (RTE) realizing communications, including intra- and inter-task, intra- and inter-ECU, and also event generation, forwarding, and dispatching (synchronization).

The *behavior* of each AUTOSAR component is represented by a set of *runnables*, that is, procedures that can be executed in response to events generated by the RTE, such as timer activations (for periodic runnables), data writes on ports, and the reception of a service call. Runnables may need to update as well as use state variables for their computations. This often requires exclusive access (write/read) to such state variables. In AUTOSAR these variables are labeled as *InterRunnableVariables* and can only be shared among runnables belonging to the same SW component. Data interactions among components occur when runnables write into and read from interface ports. Runnables may read and write *explicitly* (by calling API functions provided by the RTE) or *implicitly*. In this case, the reading runnable reads the port contents when it starts executing and writes the port data at the end of its execution. The reading and writing code is automatically generated by the AUTOSAR tools. Inside the generated RTE, when communicating runnables are mapped into different tasks that can preempt each other, the variables implementing the communication port need to be suitably protected to ensure consistency of the data.

With respect to scheduling, the code of the runnables is executed by a set of threads in a task and resource model. Runnables from different components may be mapped into the same task and must be mapped in such a way that ordering relations and causal dependencies are preserved.

In multi-core architectures, tasks can be scheduled with *partitioned* or *global* scheduling. Under *partitioned scheduling*, tasks are statically assigned to processors and each processor is scheduled separately. Under *global scheduling*, all tasks are scheduled using a single shared task-queue. They are allocated dynamically and inter-processor migration is allowed. Global scheduling algorithms can achieve higher utilization bounds. However, the required job migration can incur significant overheads [11]. Also, partitioned scheduling is adopted and supported not only by AUTOSAR, but also by commercial real-time operating systems (e.g. VxWorks, LynxOS, and ThreadX). Due to these reasons, in this work we assume partitioned scheduling.

B. Data Consistency

In multicore architectures, the protection of shared communication buffers implementing AUTOSAR ports can be performed in several ways.

- Lock-based: when a task wants to access the communication data while another task holds the lock, it blocks. When the lock is released, the task is restored in the ready state and can access the data. In multicore architectures with global locks, two options are possible. The blocked task can release the CPU on which it executes and be transferred on a (global) waiting list, or it may spin on the CPU (busy-waiting).
- Lock-free: each reader accesses the communication data without blocking. At the end of the operation, it performs a check. If the reader realizes there was a possible concurrent operation by the writer and it has read an inconsistent value, it repeats the operation. The number of retries can be upper bounded [8] [1].
- Wait-free: readers and writer are protected against concurrent access by replicating the communication buffers and by leveraging information about the time instants when they access the buffer or other information that constrains the access (such as priorities or other scheduling related information) [6] [9].

The mapping of runnables into tasks, the configuration of the task model, and the selection of the right mechanisms for the implementation of the communication over ports have a large impact on the performance of the system. Of course, context switch overheads should also be considered when defining the mapping.

Among lock-based mechanisms, the multiprocessor extension of the Priority Ceiling Protocol (MPCP) was developed in [13] to deal with the mutual exclusion problem in the context of shared-memory multiprocessors. In MPCP, tasks that fail to lock on a resource shared with remote tasks (global resource) are *suspended*, which allows other local (and possibly lower priority) tasks to execute. MSRP, the Multiprocessor extension to Stack Resource Policy [2], has been proposed and compared with MPCP in terms of worst-case blocking times [7]. A task that fails to lock on a global resource keeps *spinning* instead of suspending as in MPCP, thus keeping its processor busy. The Flexible Multiprocessor Locking Protocol (FMLP) [4] combines the strengths of the two approaches. In FMLP, short resource requests use a busy-wait mechanism, while long resource requests are handled using a suspension approach.

When the global shared resource is a communication buffer (the case of interest for our study), another possibility is wait-free methods. In [5], an asynchronous protocol is proposed for preserving data consistency with execution-time freshest value semantics in the single-writer to single-reader communication on multiprocessors. A hardware-supported Compare-And-Swap (CAS) instruction (required by any mechanism for atomic access in multicores [8]) is used to guarantee atomic reading position assignments and pointer updates.

In the case of multiple reader tasks, the buffer size can be defined by the **reader instance method**, which relies on the computation of an *upper bound for the maximum number of buffers that can be used at any given time by reader tasks* [5], or by the **lifetime bound method** based on the computation of an *upper bound on the number of times the writer can produce new values while a given data item is used by at least one reader* [10] [6].

A combination of the lifetime bound and reader instance methods can be used to obtain a better buffer sizing [16]. The implementation of a wait-free communication method that preserves Synchronous Reactive (SR, see Section I-C) flows using the buffer sizing in [16] is presented in [17] for the case of single-core OSEK implementations. However, an implementation of wait-free communication methods with the preservation of communication flows of SR models in *multicore platforms*, while not particularly difficult, has not been proposed until now. We present an outline of the characteristics and tradeoffs of such an implementation as opposed to other options including lock-based mechanisms or the enforcement of an execution order or time synchronization among tasks.

C. Preserving communication flows when implementing synchronous (Simulink) models

AUTOSAR is agnostic with respect to the development of runnable functionality. However, model-based design is very popular in the automotive domain, because of the possibility to verify the functionality of controls by simulation and formal methods. Typically the function model is first defined according to the semantics of Synchronous Reactive models like those created in MathWorks Simulink [12], and later mapped into AUTOSAR components. In AUTOSAR, behavior modeling is not mandatory to be complete, since runnables are only required to be entry points to a program.

The Simulink functional model is defined as a network of communicating blocks. Each block operates on a set of input signals and produces a set of output signals. The domain of the input function can be a set of discrete points (discrete-time signal) or it can be a continuous time interval (continuous-time signal). Continuous blocks are implemented by a solver, executing at the base rate. Eventually, every block has a sample time, with the restriction that the discrete part is executed at the same rate or at an integer fraction of the base rate.

A fundamental part of the model semantics is the rules dictating the evaluation order of the blocks. Any block for which the output is directly dependent on its input (i.e., any block with direct feedthrough) cannot execute until the block driving its input has executed. The set of topological dependencies implied by the direct feedthrough defines a partial order of execution among blocks. Before Simulink simulates a model, it orders all blocks based upon their topological dependencies, and (arbitrarily) chooses one total execution order that is compatible with the partial order imposed by the model semantics. Then, the virtual time is initialized at zero. The simulator engine scans the precedence list in order and executes all the blocks for which the value of the virtual time is an integer multiple of the period of their inputs. Executing a block means computing the output function, followed by the state update function. When the execution of all the blocks that need to be triggered at the current instant of the virtual time is completed, the simulator advances the virtual clock by one base rate cycle and resumes scanning the block list.

The code generation framework follows the general rule set of the simulation engine and must produce an implementation with the same behavior (preserving the semantics). The Real-Time Workshop/Embedded Coder (RTW/EC) code generator of MathWorks allows two different code generation options: single task and fixed-priority multitask. Single task implementations are guaranteed to preserve the simulation-time execution semantics. However, this comes at the cost of a very strong condition on task schedulability and a single-task implementation is terribly inefficient in multicore systems. In multitask implementations, the run-time execution of the model is performed by running the code in the context of a set of threads under the control of a priority-based real-time operating system (RTOS). The RTW code generator assigns each block a task priority according to the Rate Monotonic scheduling policy.

Because of preemption and scheduling, in a multirate system, the signal flows of the implementation can significantly differ from the model flows. In case a high priority block/task τ_1 drives a low priority block/task τ_2 (left side of Figure 2), there is uncertainty about which instance of τ_1 produces the data consumed by τ_2 (τ_2 should read the values produced by the first instance of τ_1, not the second). Furthermore, the shared variables implementing the communication channel must be protected for data consistency.

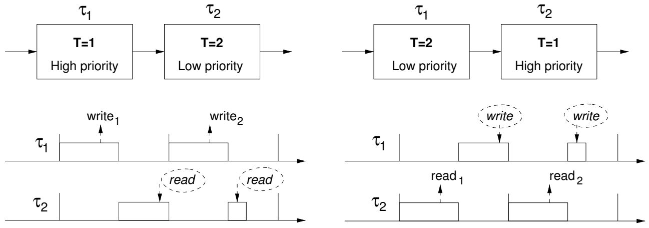

Figure 2. Possible problems when blocks with different rates interact.

In case a low priority block/task drives a high priority block/task, when the Rate Monotonic priority assignment is used, there is the additional problem that reads are executed before the corresponding writes, therefore a delay buffer and an initial value are required. The right side of Figure 2 represents this case, where the high rate task τ_2 is executed before τ_1 despite being its successor in the functional graph.

As a special case of wait-free mechanisms, the Simulink solution for these problems (in a single-core implementation) consists in the *Rate Transition* (RT) block, providing consistency of shared buffers, flow preservation, and time determinism [12]. In the case of high-to-low rate/priority transitions, the RT block output update function executes at the rate of the receiver block (left side of Figure 3), but within the task and at the priority of the sender block. In low-to-high priority transitions (right side of Figure 3), the RT block state update function executes in the context of the low rate task. The RT block output update function runs in the context of the high rate task, but at the rate of the sender task, feeding the high rate receiver. The output function uses the state of the RT that was updated in the previous instance of the low rate task. RT blocks can only be applied in a more restricted way than generic wait-free methods [6]: the periods of the writer and reader need to be harmonic, meaning one is an integer multiple of the other;

also, they only apply to one-to-one communication links. One-to-n communication is regarded as n links and each of them is buffered independently.

We refer to RT blocks associated to high priority to low priority as *direct feedthrough (DF)* and those associated to low priority to high priority transitions as *unit delay (UD)*. Both types result in additional buffer requirements, but RT blocks of type UD also result in an additional delay. The Rate Transition block of type DF behaves like a Zero-Order Hold block, and RT block of type UD composes a Unit Delay block plus a Hold block (Sample and Hold).

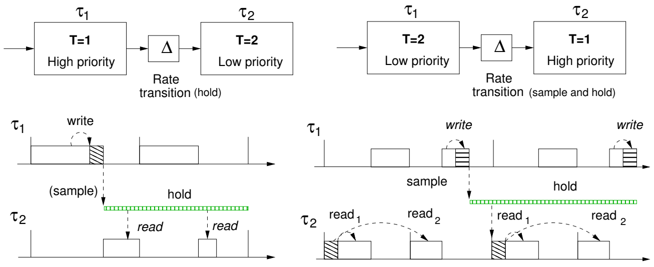

Figure 3. The effect of the introduction of Rate Transition blocks on single-CPU architectures.

The basic RT mechanism is clearly not sufficient when the reader and the writer execute on different cores. In the following we will analyze the mechanisms for guaranteeing data consistency in inter-core communication and define the possible tradeoffs. Lock-based mechanisms are sufficient when the functionality is robust with respect to possible time jitter in the communication data, as is assumed in most cases of development based on handwritten code. However, there are cases (most often when the development is based on formal models) when the flow semantics among blocks (runnables) should be preserved.

In the following, we first discuss the mechanisms for data consistency in multicore systems in Section II, and analyze the subset of the mechanisms that can also guarantee flow preservation in Section III. In Section IV we use a motivating example to discuss some of the tradeoffs and options, and point out an opportunity to define an optimization problem to find the best combination of these mechanisms. Finally, conclusion and future work are discussed in Section V.

II. Mechanisms for Guaranteeing Data Consistency in Multicore Systems

The mechanisms that can be used to guarantee data consistency in multicore systems when the writer and at least one of the readers are allocated on different cores are:

1) Explicit synchronization between the writer and the reader, possibly supplemented by timing analysis, to ensure that no read (write) can be performed while a write (read) is in progress.
2) Wait free methods: communication buffers are replicated and no simultaneous access to one buffer instance by the writer and any of the readers.
3) Semaphore locks: multiprocessor versions of the immediate priority ceiling semaphores, along the line of MPCP, or MSRP.

In the following, we discuss the applicability/extension of these mechanisms, including an analysis of the timing and memory overheads associated to each of them. However, first we need to shortly introduce the task model and the foundation of timing analysis.

Unless otherwise specified, we will assume that each subsystem/runnable ρ_i is defined to be activated by a periodic event with period T_i. A runnable is characterized by a worst-case execution time (when executed in isolation) γ_i, and it is mapped into a task τ_j in such a way that the task period is harmonic with the runnable period (the task period must be an integer divisor). We denote the period of τ_j by T_j. The mapping of runnables into tasks defines a sequential execution order. We will use two indexes for a runnable as in $\rho_{j,k}$, meaning that the runnable is the j-th among those mapped to task τ_k. Finally, C_i denotes the worst-case execution time of τ_i. Clearly, $C_i = \sum_k \gamma_{k,i}$.

In a system where tasks are independent and scheduled by priority with preemption, the worst case response time of task τ_i is

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

where $hp(i)$ indicates the indexes of all tasks allocated to the same processor, with priority higher than τ_i. Similarly, the worst case response time of runnable $\rho_{k,i}$ is

$$R_{k,i} = \sum_{p \leq k} \gamma_{p,i} + \sum_{j \in hp(i)} \left\lceil \frac{R_{k,i}}{T_j} \right\rceil C_j \qquad (2)$$

The response time of a runnable will also be indicated with a single index whenever the runnable has a single index (without reference to the task into which it is mapped).

A. Explicit synchronization between writer and reader

For any pair of runnables ρ_i and ρ_j mapped to different cores, we are interested in knowing whether the execution of ρ_i and ρ_j can overlap. We denote the minimum offset from the activation of ρ_i to the following activation of ρ_j that is released later than ρ_i as $o_{i,j}$, and the minimum offset from ρ_j to ρ_i as $o_{j,i}$. If the worst case response time R_i of ρ_i is no greater than $o_{i,j}$, then the finish time F_i of ρ_i is $F_i = O_i + R_i \leq O_i + o_{i,j} = O_j \leq S_j$, where O_i and O_j are the release times of ρ_i and ρ_j respectively. Likewise, if the response time R_j of ρ_j is no greater than $o_{j,i}$, then the finish time F_j of ρ_j is $F_j = O_j + R_j \leq O_j + o_{j,i} = O_i \leq S_i$. In summary, if the following (sufficient) condition is satisfied

$$R_i \leq o_{i,j} \bigwedge R_j \leq o_{j,i} \qquad (3)$$

there is no overlap in the execution window of ρ_i and ρ_j, as shown in Figure 4. Communication variables do not need to be protected thus there is no timing and memory overhead.

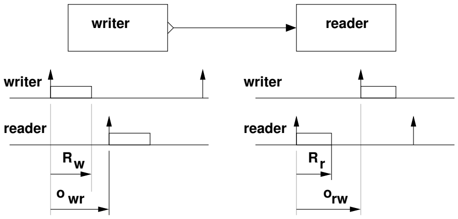

Figure 4. Absence of overlap in the execution of the writer and the reader.

B. Wait-free communication buffers

The objective of wait-free methods is to avoid blocking by ensuring that each time a writer needs to update the communication data, it is reserved a new buffer area. At the same time, readers are free to use other dedicated buffers. Of course, as in the SR semantics, a reader that preempts a writer (only possible if the reader has higher priority) will access the latest buffer for which a write has been completed (not the one currently being written).

Figure 5 shows the typical stages performed by the writer and the reader in a wait-free protocol implementation.

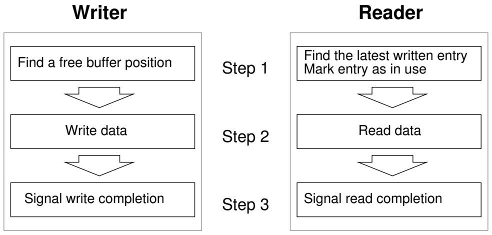

Figure 5. Stages for Writer and Readers in a wait-free protocol.

These stages have been implemented in [5] by means of an atomic Compare-And-Swap (CAS) operation. The CAS takes three operands

```
Compare-and-Swap(mem, v1, v2)
```

where the value `v2` is written into the memory location `mem` only if the current value of `mem` is equal to `v1`. Otherwise, the value at `mem` is left unchanged. The algorithm makes use of three global sets of data. An array of buffers (BUFFER) is sized so that there is always an available buffer for the writer to write new data. An array READING that keeps track in the i-th position what is the buffer index in use by the i-th reader. When the reader is in the process of updating this information, a value 0 is stored in this entry. Finally, a variable LATEST keeps track of the latest BUFFER entry that has been updated by the writer. The code for the writer is shown in Algorithm 1. The writer updates the view of what buffers are used by readers and then picks a free buffer using the procedure GetBuf(). Then, it uses the buffer item to write the newly produced data item and updates LATEST. Finally, it uses CAS operation to ensure the consistency in the updates of the READING indexes.

On the reader side, the code is in Algorithm 2. The reader first looks for the latest entry updated by the writer, stores it in the local variable ridx and then reads its contents.

Algorithm 1: Modified Chen's Protocol for Writer

```
Data: BUFFER [1,...,NB]; NB: Num of buffers
Data: READING [1,...,n]; n : Num of readers
Data: LATEST
GetBuf();
begin
    bool InUse [1,...,NB];
    for i=1 to NB do InUse [i]=false;
    InUse[LATEST ]=true;
    for i=1 to n do
        j = READING [i];
        if j !=0 then InUse [j]=true;
    end
    i=1;
    while InUse [i] do ++i;
    return i;
end
Writer();
begin
    integer widx, i;
    widx = GetBuf();
    Write data into BUFFER [widx];
    LATEST = widx;
    for i=1 to n do CAS(READING [i],0,widx);
end
```

Algorithm 2: Modified Chen's Protocol for Readers

```
Data: BUFFER [1,...,NB]; NB: Num of buffers
Data: READING [1,...,$n_r$]; $n_r$: Num of readers
Data: LATEST
Reader();
begin
    constant id; – Each reader has its unique id;
    integer ridx;
    READING [id]=0;
    ridx = LATEST;
    CAS(READING [id],0,ridx);
    ridx = READING [id];
    Read data from BUFFER [ridx];
end
```

This mechanism ensures data consistency with some runtime overhead (its complexity is $O(n_r)$ where n_r is the number of readers), but it avoids the introduction of blocking times. However, it requires a number of buffer replicas, thus additional memory, if compared with lock-based methods. The number of required buffers is $n_r + 2$ in [5], which has been improved in following researches [14] [16]. Finally, the wait-free mechanism presented in [5] does not preserve the flow semantics of SR models, given that the value being read is simply the latest one updated by the writer and therefore depends on the scheduling of the writer and the reader.

C. Lock-based mechanisms

MPCP (Multiprocessor Priority Ceiling Protocol) [13] is the multiprocessor extension of the priority ceiling protocol (PCP). Tasks use assigned priorities for normal executions, and inherit the ceiling priority of the resource whenever they execute a critical section on a shared resource. The ceiling priority of a local resource is defined as the highest priority of any task that can possibly use it. For a global resource, its remote priority ceiling is required to be higher than any task priority in the system. For this purpose, a base priority offset which is higher than the priority of any task is applied to all global resources. Jobs are suspended when they try to access a locked global critical section and added to a priority queue. The suspension of a higher priority task blocked on a global critical section allows other local tasks (possibly lower priority ones) to be executed and may even try a lock on local or global critical sections. The worst-case remote blocking time of a job is bounded as a function of the duration of critical sections of other jobs, and does not depend on the non-critical sections (see Equations (6)-(8)).

MSRP (Multiprocessor Stack Resource Policy) [7] is a multiprocessor synchronization protocol, derived by extension from the Stack Resource Policy (SRP). For local resources, the algorithm is the same as the SRP algorithm, where tasks are allowed to access local resource through nested critical sections. However, global critical sections cannot be nested. A task that fails to lock on a global resource keeps *spinning* instead of suspending as in MPCP, thus keeping its processor busy. To minimize the spin lock time (which is a wasted CPU time), tasks cannot be preempted when executing a critical section on a global resource to ensure that the resource be freed as soon as possible. MSRP uses a First-Come-First-Serve queue (as opposite to a priority-based queue in MPCP) for each global resource to manage the tasks that fail to lock the resource.

FMLP (Flexible Multiprocessor Locking Protocol) [4] is a flexible approach which combines the strengths of MPCP and MSRP. It manages short resource requests by a busy-wait mechanism (as in MSRP), and long resource requests using a suspension approach (as in MPCP). The threshold to determine whether a resource should be considered long or short is specified by the user. Since FMLP allows resource requests be nested, deadlock is prevented by grouping resources and allowing only one job to access resources in any given group at any time. Each group contains either only short (protected by a non-preemptive queue lock) or only long (protected by a semaphore) resources.

The mixture of spinning with short critical section and suspension with long critical section makes the analysis of FMLP very complex. For simplicity, in the following we only summarize the timing analysis of MSRP and MPCP. We assume periodic tasks under partitioned static priority scheduling policies. With respect to memory, MPCP is a suspension-based locking mechanism and does not allow sharing the stack space of tasks. When using MSRP or any spin-lock based policy, the execution of all tasks allocated to the same processor can be perfectly nested (once a task starts execution it cannot be blocked, but only be preempted by a higher priority task which completes before it resumes execution), therefore all tasks can share the same stack.

1) Timing Analysis of MPCP [11]: The execution of task τ_i is defined as a set of alternate code sections in which the task executes without the need of a (global or local) shared resources, defined as *normal execution segments*, and critical sections. The worst case execution time (WCET) can be defined by a tuple $\{C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,s(i)-1}, C_{i,s(i)}\}$, where $s(i)$ is the number of normal execution segments of τ_i, and $s(i) - 1$ is the number of critical sections. $C_{i,j}$ is the WCET of the j-th normal execution segment of τ_i and $C'_{i,j}$ the WCET of the j-th critical section. π_i denotes the normal priority of τ_i (the higher the number, the lower the priority), T_i the period, and E_i the CPU resource. The global shared resource associated to the j-th critical section of τ_i is denoted as $\mathcal{S}_{i,j}$. The remote priority ceiling for $\mathcal{S}_{i,j}$ is denoted as $\Pi_{i,j}$, which is defined globally and can be compared with critical sections for tasks on the same core as well as other cores. The WCET C_i of τ_i is

$$C_i = \sum_{1 \leq j \leq s(i)} C_{i,j} + \sum_{1 \leq j < s(i)} C'_{i,j} \tag{4}$$

The normal execution segment of a task can be blocked by the critical section of each lower priority task on the same core. For each of the $s(i)$ normal execution segment, the worst case local blocking time τ_i may suffer is the longest critical section among all the lower priority tasks. Thus the total local blocking time of τ_i is

$$B_i^l = s(i) \times \sum_{k:\pi_k > \pi_i \& E_k = E_i} \max_{1 \leq m < s(k)} C'_{k,m} \tag{5}$$

τ_i can only be interfered by critical sections with a higher remote priority ceiling once it enters its critical section. Also, since the critical section has a higher priority than the normal execution segment, in a critical section τ_i will only suffer one such interference for each task on the same core. Thus the response time of the j-th critical section is bounded by

$$W'_{i,j} = C'_{i,j} + \sum_{k \neq i: E_k = E_i} \max_{1 \leq m < s(k) \& \Pi_{k,m} < \Pi_{i,j}} C'_{k,m} \tag{6}$$

The remote blocking time $B^r_{i,j}$ suffered in the j-th critical section can be calculated by the following iterative formula

$$\begin{aligned} B^r_{i,j} \;&= \max_{\pi_k > \pi_i \& \mathcal{S}_{k,m} = \mathcal{S}_{i,j}} W'_{k,m} \\ &+ \sum_{\pi_h < \pi_i \& \mathcal{S}_{h,n} = \mathcal{S}_{i,j}} \left(\left\lceil \frac{B^r_{i,j}}{T_h} \right\rceil + 1 \right) W'_{h,n} \end{aligned} \tag{7}$$

where its initial value can be set as the first term on the right hand side. The total remote blocking time is

$$B_i^r = \sum_{1 \leq j < s(i)} B^r_{i,j} \tag{8}$$

The worst case response time R_i of τ_i can be calculated as the convergence of the following iterative formula

$$R_i = C_i + B_i^l + B_i^r + \sum_{\pi_h < \pi_i \& E_h = E_i} \left\lceil \frac{R_i + B_h^r}{T_h} \right\rceil C_h \tag{9}$$

2) Timing Analysis of MSRP [7]: The spin block time $L_{i,j}$ that a task τ_i needs to spend for accessing a global resource $\mathcal{S}_{i,j}$ can be bounded by

$$L_{i,j} = \sum_{E \neq E_i} \max_{\tau_k : E_k = E, 1 \leq m < s(k)} C'_{k,m} \qquad (10)$$

This is the time increment to the j-th critical section of τ_i, thus its actual worst case execution time C_i^* is

$$C_i^* = C_i + \sum_{1 \leq j < s(i)} L_{i,j} \qquad (11)$$

MSRP maintains the same basic property of SRP, that is, once a task starts execution it cannot be blocked. The local blocking time B_i^l and remote blocking time B_i^r are

$$B_i^l = \max_{k:\pi_k > \pi_i \& E_k = E_i} \max_{1 \leq m < s(k)} C'_{k,m} \qquad (12)$$

$$B_i^r = \max_{k:\pi_k > \pi_i \& E_k = E_i} \max_{1 \leq m < s(k)} C'_{k,m} + L_{k,m} \qquad (13)$$

The worst case response time R_i of τ_i can be calculated as the convergence of the following iterative formula

$$R_i = C_i^* + B_i^l + B_i^r + \sum_{\pi_h < \pi_i \& E_h = E_i} \left\lceil \frac{R_i}{T_h} \right\rceil C_h^* \qquad (14)$$

D. Timing analysis with release jitter

The scheduling solutions can be based on a slightly different task activation model. Besides the pure periodic activation model, response time analysis can be performed in the case when the activation of task instances can be deferred by a bounded quantity of time from the periodic activation signal. For task τ_i, the activation or release jitter J_i is defined as the worst case delay from the periodic activation signal to the true activation of the task instance. When the system tasks are activated with release jitter, the formula for computing the worst case response time changes. For example, (1) must be revised as

$$R_i = R_i^* + J_i, \qquad R_i^* = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^* + J_j}{T_j} \right\rceil C_j \quad (15)$$

Similarly, Equations (9) and (14) must be updated to account for the release jitter.

Lock-based mechanisms do not guarantee the preservation of SR flows and are not meant to. A discussion of mechanisms for flow preservation in addition to data consistency is the subject of the next section.

III. Mechanisms for preserving SR Communication Flows in Multicore systems

Of the previous general categories of protection mechanisms, only two can be adapted to also provide flow preservation. Lock-based mechanisms are indeed based on the assumption that the execution order of the writer and reader is unknown and there is the possibility of one to preempt the other while operating on the shared resource. Therefore the possible options are:

1) enforcing synchronization in the scheduled execution of the writer and its readers.
2) wait free methods with flow preservation.

A. Enforcing synchronization in the execution of the writer and its readers

This case requires little changes to the timing conditions for demonstrating the absence of preemption between the writer and its readers. The main addition is the enforcement of an execution order among writer and reader runnables and the definition of a deadline for the reader task.

B. Wait-free communication buffers

Rate Transition blocks are in the general category of wait-free communication methods (although they are a special implementation for a restricted case). Therefore, their extension to the communication of runnables executing on different cores belongs to this section and is discussed first.

Local priorities of the writer and the reader are of course not sufficient to enforce an execution order as in the single-core case. Since the runnables are executed with partitioned scheduling, there is generally no notion of global priority.

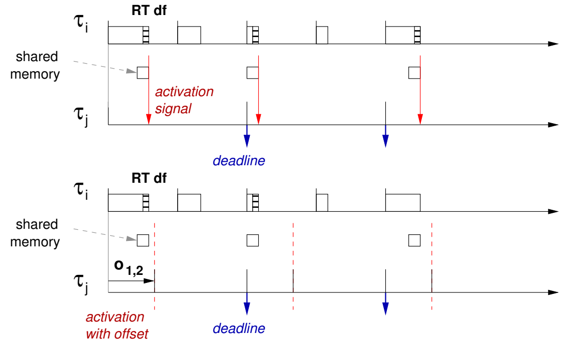

Figure 6. Direct feedthrough RT blocks in multi-cores with two activation options for the reader: event-triggered (top) and time-triggered (bottom)

For direct feedthrough RT block, the writer and reader runnables (tasks) need to be scheduled as shown in Figure 6 for a communication link from τ_i and τ_j. The reader needs to execute after the output update portion of the RT block (following the first writer instance), and before the third instance of the writer completes and the RT output update function is executed again. This can be obtained in two ways depending on the support provided by the operating system.

After the execution of the output update part of the RT block which is on the same core as the writer runnable, an activation signal can be sent to the receiver task (typically realized as an inter-processor interrupt signal) as shown in the upper part of the figure. As in the single processor case, the implementation of the RT block consists of the update function only. The variables shared between cores are the output variable of the RT block. Since the receiver must terminate before the next execution of the update function of the RT block, the receiver is characterized by a base period T_j, an activation jitter J_j equal to the worst case response time of the writer runnable (including the RT block portion) R_i, and a deadline T_j. This implementation does not require any clock synchronization among the two cores (and their schedulers). The problem lies in the activation jitter of the receiver task, which impacts on the timing analysis of the lower priority tasks executing on the same core.

A different option consists in the synchronized activation of the writer and reader as in the bottom part of Figure 6. In this case, the reader is activated by a periodic signal, synchronized with the activation of the writer, with offset $o_{i,j}$ no less than the worst-case response time R_i of the writer τ_i. The absolute deadline is the same as in the previous case, and the reader must be completed within $T_j - o_{i,j}$ time units from its activation. The two conditions on the offset and the receiver deadline can be summarized by the following formula (regardless of whether $T_i \geq T_j$ or $T_i < T_j$)

$$R_i \leq o_{i,j} \bigwedge R_j + o_{i,j} \leq T_j \qquad (16)$$

This implementation requires the synchronization of the two schedulers. The feasibility of the receiver execution within the deadline is similar to the previous case, but with the advantage of activations without jitter.

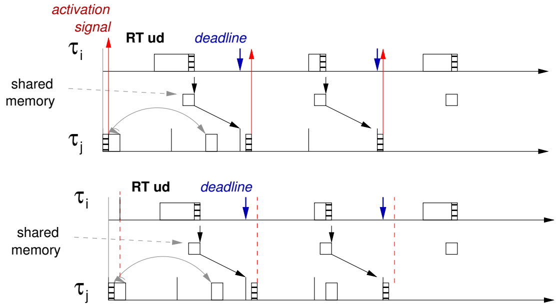

Figure 7. Unit delay RT blocks in multi-cores with two activation options for the writer: event-triggered (top) and time-triggered (bottom)

For unit-delay RT block, there are similar requirements between the execution of the reader and the writer runnables and the state update and output update parts of the RT block. The situation is summarized in Figure 7. The global shared variable is the state variable of the RT block. As in the case of direct feedthrough blocks, there are two scheduling options for the activation of the writer task, including the state update part of the RT block. The writer task can be activated by an interrupt signal, generated right after the completion of the output update part of the RT block in the receiver task. The other option is to be activated with a set of periodic signals, synchronized with the activation of the receiver task, with an offset equal to the worst case response time of the output update function of the RT block. The deadlines are similarly computed as the writer period, or the difference between the writer period and the offset.

In the implementation of more general wait-free mechanisms providing a correct implementation of SR flows in the case that periods are not harmonic, an additional effort is required. Any correct implementation of an SR model requires that the data item used by the reader be defined based on the writer and reader task activation times. Both tasks, however, are not guaranteed to start their execution at their release times because of scheduling delays. Therefore, the selection of the data that is written or read must be delegated to the operating system (or a hook procedure that is executed at the task activation time). At execution time, the writer and the readers will use the buffer positions defined at their activation times. Of course, as in the case of the previously considered wait-free mechanisms, the writer may produce multiple outputs before the reader finishes reading the data. In this case, the implementation must consist of an array of buffers in which pointers (indices) are assigned to writers and readers to find the right entry.

In the general case of multiple reader tasks, the buffer sizes can be defined by analyzing the relationship between the writer and its reader task instances, as shown in [16]. With reference to the algorithm presented in the previous sections, the writer and reader protocols need to be partitioned in two sections, one executed at task activation time, the other at runtime. The management of the buffer positions (READINGLP[], READINGHP[], PREVIOUS and LATEST) are delegated to the operating system or a hook procedure that is executed at the task activation time. The write and read operations are executed at the runtime time, using the buffer positions defined at their activation time.

Readers are divided in two sets according to different requirements for flow preservation. Lower priority readers read the value produced by the latest writer instance activated before their activation (the communication link is direct through). Higher priority readers read the value produced by the previous writer instance (the communication link has a unit delay). The two corresponding buffer entries are indicated by the LATEST and PREVIOUS variables. Two separate arrays, READINGLP[] and READINGHP[], contain one entry for each low and high-priority readers respectively, even if they are managed in the same way. The writer updates all zero-valued elements of READINGHP[] and READINGLP[] with the value of PREVIOUS and LATEST respectively (lines 24 and 25 in Algorithm 3). The pseudo code for the writer and readers is shown in Algorithms 3 and 4, respectively.

When the reader executes on a different core than the writer, if the communication link is direct feedthrough, additional mechanisms need to be employed to ensure the reader starts execution after the data is written. This is because of the assumption that tasks (runnables) on different cores are scheduled separately thus there is no guarantee

Algorithm 3: Modified Chen's Protocol for SR flow preservation - Writer part

```
Data: BUFFER [1,...,NB]; NB: Num of buffers
Data: READINGLP [1,...,$n_{lp}$]; $n_{lp}$: Num of lower priority readers
Data: READINGHP [1,...,$n_{hp}$]; $n_{hp}$: Num of higher priority readers
Data: PREVIOUS, LATEST
GetBuf();
begin
    bool InUse [1,...,NB];
    for i=1 to NB do InUse [i]=false;
    InUse[LATEST]=true;
    for i=1 to $n_{lp}$ do
        j = READINGLP [i];
        if j !=0 then InUse [j]=true;
    end
    for i=1 to $n_{hp}$ do
        j = READINGHP [i];
        if j !=0 then InUse [j]=true;
    end
    i=1;
    while InUse [i] do ++i;
    return i;
end
Writer_activation();
begin
    integer widx, i;
    widx = GetBuf();
    PREVIOUS = LATEST;
    LATEST = widx;
    for i=1 to $n_{hp}$ do CAS(READINGHP [i], 0, PREVIOUS);
    for i=1 to $n_{lp}$ do CAS(READINGLP [i], 0, LATEST);
end
Writer_runtime();
begin
    Write data into BUFFER [widx];
end
```

Algorithm 4: Modified Chen's Protocol for SR flow preservation - Readers

```
ReaderLP_activation();
begin
    constant id; – Each lower priority reader has its unique id;
    integer ridx;
    READINGLP [id]=0;
    ridx = LATEST;
    CAS(READINGLP [id],0,ridx);
    ridx = READINGLP [id];
end
ReaderHP_activation();
begin
    constant id; – Each higher priority reader has its unique id;
    integer ridx;
    READINGHP [id]=0;
    ridx = PREVIOUS;
    CAS(READINGHP [id],0,ridx);
    ridx = READINGHP [id];
end
Reader_runtime();
begin
    Read data from BUFFER [ridx];
end
```

that the writer will finish execution before the reader starts. Like the case of RT blocks, this execution order can be enforced by an activation signal sent to the reader (typically realized as an inter-processor interrupt signal), or by the synchronized activation of the writer and reader.

IV. Optimization Opportunities: a Motivating Example

As stated in the previous section, several methods can be used to ensure data consistency, time determinism, and flow preservation. In some cases, it is possible to achieve safe operation without additional costs, by simply enforcing the synchronization of the writer and the reader and ensuring no overlap in their execution. This sometimes requires the careful selection of the mapping order of runnables into tasks and the allocation of tasks to cores. In other cases, several methods are available that offer tradeoffs between the amount of additional memory, runtime overheads, and additional blocking time imposed over tasks. These methods can be used in a combination. For example, by assigning an early execution order to runnables with large sets of communication and shared state variables, then by properly configuring the activation offsets of tasks to guarantee absence of preemption when possible. For the variables that still need to be protected, use spin-based mechanisms for runnables with short global critical sections and the other mechanisms to the remaining cases, possibly using wait-free communication when the memory overheads for the replicated buffers can be tolerated. By formalizing the memory cost and the amount of additional execution time or blocking time that is required by these methods, it should be possible to define an optimization problem to find the best system configuration.

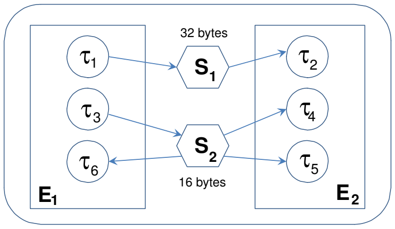

Figure 8. An example dual-core system with two global shared resources.

As a motivating example, consider the system of Figure 8. Each of the two cores is executing three tasks, which are indexed in priority order (τ_1 has the highest priority, τ_6 the lowest). The two global shared resources $\mathcal{S}_1$ and $\mathcal{S}_2$ are used by tasks that are mapped to different cores. The shared resources are the communication variables accessed by the tasks, and the sizes of the variables for $\mathcal{S}_1$ and $\mathcal{S}_2$ are 32 and 16 bytes, respectively. The global critical sections on $\mathcal{S}_1$ are assumed to have higher priorities than those on $\mathcal{S}_2$. The parameters for the tasks including the priority π_i, the period T_i, and the execution time C_i, are given in Table I. The deadlines are assumed to the be the same as the period.

Table I
TIMING ANALYSIS OF DIFFERENT MECHANISMS FOR DATA CONSISTENCY (TIME UNIT: MILLISECOND)

τ_i	E_i	π_i	T_i	C_i	**MPCP**			**MSRP**			**Wait-free**
					B_i^l	B_i^r	R_i	B_i^l	B_i^r	R_i	R_i
τ_1	E_1	1	5	{1, 0.5, 1}	2.0	0.5	5.0	0.5	1.0	4.5	2.5
τ_2	E_2	2	5	{1, 0.5, 1}	2.0	1.0	**5.5**	0.5	1.0	4.5	2.5
τ_3	E_1	3	10	{0.5,0.5,0.5}	1.0	1.5	9.0	0.5	1.0	9.5	4.0
τ_4	E_2	4	10	{0.5,0.5,0.5}	1.0	3.5	**13.0**	0.5	1.0	9.5	4.0
τ_5	E_2	5	10	{0.6,0.5,0.6}	0	3.5	**18.2**	0	0	**13.2**	8.2
τ_6	E_1	6	10	{0.6,0.5,0.6}	0	7.0	**24.2**	0	0	**13.2**	8.2

The timing analysis results for the tasks are shown in Table I, comparing the possible implications using MPCP (Equation (9)), MSRP (Equation (14)), and wait-free method (Equation (1)) to guarantee the data consistency. In this example, the length of the critical sections are all set to be $0.5ms$, which is relatively long. Because of the long remote blocking due to global critical sections, and possible back-to-back executions due to suspended tasks, the MPCP mechanism performs poorly: the remote blocking time can be as large as $7ms$, and 4 out of 6 tasks are not schedulable. MSRP mechanism can limit the local and remote blocking time, but the system is still unschedulable. Wait-free mechanism removes the blocking altogether at the price of additional buffer memory and (small) timing overheads for buffer access management. In the table, we show the result of wait-free method assuming no timing overhead, where all tasks are schedulable. In [17], the performance of the implementations using the ePICos18 OSEK RTOS is evaluated. On the 40 MHz-10 MIPS PIC18F452 processor, the procedures for wait-free mechanisms needs 77 instruction cycle (or 7.7 μs). The timing overhead associated to the management of the buffer pointers in the wait-free mechanism is expected to be very short compared to the duration of a critical section.

In our example, the memory overhead associated to the wait-free method can be upper bounded by $n_r + 2$ additional copies of the shared variable where n_r is the number of readers [5] (but results improving this bound are available, e.g. [16]), thus for this example the total memory overhead is $(1+2) \times 32 + (3+2) \times 16 = 176$ bytes.

V. CONCLUSION AND FUTURE WORK

In this paper, we described some of the issues in the implementation of a set of AUTOSAR runnables into a set of concurrent tasks on multicore platforms. The requirements for the consistency of communication and state variables and the possible additional requirements of flow preservation (time determinism on the communication) can be satisfied using several protection mechanisms. The available methods offer tradeoffs between time response (and time overheads) and demand for additional (RAM) memory. Memory costs need to be analyzed, in light of the time constraints of the application, to select the best mechanism for the application runnables or possibly a combination of them within the same application. As the future work, we plan to implement the proposed algorithms for wait-free methods along with lock based mechanisms (MPCP, MSRP, etc.) on an AUTOSAR operating system, to characterize the associated timing and memory overheads of these methods and compare them.

REFERENCES

[1] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15:134–165, May 1997.

[2] T.P. Baker. A stack-based resource allocation policy for realtime processes. In *RTSS '90: Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[3] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91, January 2003.

[4] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, 2007.

[5] J. Chen and A. Burns. A fully asynchronous reader/write mechanism for multiprocessor real-time systems. Technical Report YCS 288, Department of Computer Science, University of York, May 1997.

[6] J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Proc. of International Conference on Real-Time Computing Systems and Applications*, pages 236–246, 1999.

[7] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 73–83, 2001.

[8] M. Herlihy. A methodology for implementing highly concurrent structures. In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

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

[10] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Proc. of IEEE Real-Time Systems Symposium*, 1993.

[11] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS '09: Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 469–478, 2009.

[12] MathWorks. *The MathWorks Simulink and StateFlow User's Manuals*. web page: http://www.mathworks.com.

[13] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proc. of International Conference on Distributed Computing Systems*, pp. 116–123, 1990.

[14] C. Sofronis, S. Tripakis, and P. Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proc. of International conference on Embedded software*, pages 21–33, 2006.

[15] The AUTOSAR consortium. *The AUTOSAR Standard, specification version 4.0*. web page: http://www.autosar.org.

[16] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli. Improving the size of communication buffers in synchronous models with time constraints. *IEEE Transactions on Industrial Informatics*, 5(3):229–240, August 2009.

[17] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli. Optimal synthesis of communication procedures in real-time synchronous reactive models. *IEEE Transactions on Industrial Informatics*, 6(4):729–743, November 2010.