Efficient Implementation of AUTOSAR Components with Minimal Memory Usage

Haibo Zeng
McGill University, haibo.zeng@gmail.com

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

***Abstract*—The implementation of AUTOSAR runnables in a concurrent program executing as a set of tasks reveals several issues and trade-offs because of the need to protect communication and state variables, to guarantee deadlines and preserve the flow semantics of the model and the objective of using the least possible amount of memory. We discuss some of these tradeoffs and options and outline a problem formulation that can be used to compute the solution with minimum memory requirements executing within the time constraints.**

I. Introduction

The AUTOSAR development partnership has been created to develop an open industry standard for automotive software architectures, including the definition of components and their interface. In AUTOSAR, the *functional architecture* of the system is a collection of *SW Components* cooperating through their interfaces on a conceptual framework called *Virtual Functional Bus or VFB*. Components interfaces are ports for data-oriented or service-oriented communication. In the first case (of type Send-Receive), the port represents (asynchronous) access to a shared storage in which one component may write into and others may read from. In the case of service-oriented communication, a client component may invoke the services of a server component.

The *behavior* of each AUTOSAR component is represented by a set of *runnables*, procedures that can be executed in response to events, such as timer activations (for periodic runnables), or data writes on ports, or other application signals. In this work, we restrict to runnables that are activated in response to periodic timer events.

Runnables may need to update as well as use state variables for their computations, which 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 component. Of course, (data) interactions among components occur when runnables write into and read from interface ports. When communicating runnables are mapped into different tasks that can possibly preempt each other, the variables implementing the communication port need to be suitably protected to ensure consistency of the data.

The implementation of runnables consists of the code implementing the functionality. With respect to scheduling, the runnables code 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 are preserved.

In this paper, we deal with timing issues at the local level, that is, for components mapped into ***tasks executing on the same ECU***. The mapping of runnables into tasks, the configuration of the task model, and the selection of the mechanisms for the implementation of the communication over ports (protecting against data inconsistency and possibly flow semantics violations) have a large impact on the performance of the system. The selection of the communication mechanism and the protocol to protect state variables leverages tradeoffs between time overhead for the execution of the protocol, memory required for the implementation of the mechanism and possible blocking time. In this work, using the AUTOSAR model and definitions, we present a scheme for the optimal selection of

- the execution order of runnables mapped into a task
- the assignment of preemption thresholds to tasks
- the selection of the appropriate mechanism for protecting communication variables and state variables among a set of possible choices that includes preemption disabling, lock-based methods (priority ceiling semaphores), and wait-free methods.

within constraints defined on the application as

- deadlines for tasks and runnables
- the (optional) need to preserve the flow semantics on communication links

with the objective of minimizing the use of RAM memory for stack space and the implementation of communication.

II. System model: assumptions and notation

An AUTOSAR model of execution is represented by a *Directed Graph* $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where $\mathcal{V}$ is the set of vertices, representing the runnables, and $\mathcal{E}$ the set of edges or links between runnables. Such a graph will have inputs from sampling, source and constant blocks, representing the signals from the controlled system or plant. At the other end of the graph, the output signals are the result of the controller's computations. We assume an implementation on a single processor where concurrent tasks are scheduled by fixed priority. The notation is the following:

$\rho = \{\rho_1, \ldots, \rho_{|\rho|}\}$ is the set of *runnables*. A runnable ρ_i reads from a set of *input ports*, denoted as $\mathcal{E}_i^{\text{in}}$, and a set of *output ports*, denoted as $\mathcal{E}_i^{\text{out}}$. Each runnable is activated periodically, with *period* t_i, which is also the sampling period for the signals on the input ports. The signals are processed by the runnable and the result of the computation is a set of signal with the same rate, produced on the output ports. We also denote the set of data ports accessed by ρ_i as $\mathcal{E}_i = \mathcal{E}_i^{\text{in}} \bigcup \mathcal{E}_i^{\text{out}}$.

$\mathcal{E} = \{\varepsilon_1, \ldots, \varepsilon_{|\mathcal{E}|}\}$ is the set of *shared resources*. We consider the case of one-to-many communication: a shared resource ε_i has a writer runnable, denoted as $\rho^{\text{W}}(\varepsilon_i)$, and a set of reader runnables $\rho^{\text{R}}(\varepsilon_i)$. We also denote the set of readers with higher (lower) priority than the reader $\rho^{\text{W}}(\varepsilon_i)$ as $\rho^{\text{HR}}(\varepsilon_i)$ ($\rho^{\text{LR}}(\varepsilon_i)$). M_i denotes the size of the data communicated over ε_i.

The execution time of a runnable ρ_i is characterized by $(C_{i,0}, C_{i,1}, ..., C_{i,|\mathcal{E}_i^{\text{in}}|}, ..., C_{i,|\mathcal{E}_i^{\text{in}}|+|\mathcal{E}_i^{\text{out}}|})$, where

- $|\mathcal{E}_i^{\text{in}}|$ is the number of execution segments of ρ_i reading from input ports;
- $|\mathcal{E}_i^{\text{out}}|$ is the number of execution segments of ρ_i writing into output ports;
- $C_{i,0}$ is the total worst case execution time of the normal execution segments;
- $C_{i,j}, j = 1, ..., |\mathcal{E}_i^{\text{in}}|$ is the worst-case execution time of the critical section on the j-th input port;
- $C_{i,|\mathcal{E}_i^{\text{in}}|+j}, j = 1, ..., |\mathcal{E}_i^{\text{out}}|$ is the worst case execution time of the critical section on the j-th output port.

We also use $C_i(\varepsilon_k)$ to denote the worst case execution time of ρ_i accessing the input/output port ε_k, $\forall \varepsilon_k \in \mathcal{E}_i^{\text{in}} \bigcup \mathcal{E}_i^{\text{out}}$. The worst case execution time c_i of the runnable ρ_i also depends on the time overhead of the mechanism used to protect the shared resources.

$\mathcal{T} = \{\tau_1, \ldots, \tau_{|\mathcal{T}|}\}$ is the set of *tasks*. Each task τ_i has a priority Π_i and an activation period T_i. Each task is periodic with an offset equal to zero, thus all tasks start at the same time instant $t = 0$. The task τ_i is also assigned with a unique priority Π_i.

A mapping relation $m(\rho_i, \tau_j, k)$ may be defined between a runnable ρ_i and a task τ_j meaning that the code implementing the runnable ρ_i is executed in the context of task τ_j with ordering index k. A mapping relation is only possible if the execution period of ρ_i is an integer multiple of τ_j, i.e. $t_i = k_i \cdot T_j$ for some integer k_i. The deadline of the ρ_i is defined as the period of the task τ_j it is mapped to, thus $D_i = T_j$, which is no greater than the period t_i of ρ_i. The priority order of runnables is inherited from the priority order of the tasks they are mapped into, the priority π_i of runnable ρ_i is inherited from the priority of the task τ_h it is mapped to, i.e. $\pi_i = \Pi_h$. If two runnables are mapped to the same task, the mapping order index must match the partial order in the execution of the runnables.

Besides the normal priority π_i, a runnable ρ_i is also assigned a preemption threshold γ_i with $\pi_i \geq \gamma_i$ [4]. When the runnable is activated, it is inserted in the ready queue inside the task it is mapped to with the normal priority. When the runnable starts execution, its priority is raised to the preemption threshold level.

As summarized in [3], there are four different mechanisms, all of which can guarantee data consistency, but only two of them ensure flow preservation.

M1: Demonstrating absence of preemption. For any pair of runnables ρ_i and ρ_j mapped to different tasks, with priority $\pi_i > \pi_j$, we denote the minimum offset from the activation of ρ_i to the following activation of ρ_j as $o_{i,j}$. If the worst case response time r_i of ρ_i is no greater than $o_{i,j}$, then there can be no preemption from ρ_j to ρ_i. This can be applied for both data consistency and flow preservation.

M2: Disabling preemption. Preemption can be disabled for runnables with negligible time and memory overhead. However, this will result in a worst case blocking time (for other higher priority runnables) equal to the duration of the longest runnable. However, this mechanism alone cannot guarantee flow preservation as it has no awareness on the writer instance the reader is reading from (which is the key for flow preservation).

M3: Wait-free communication buffers. For a shared resource ε_i, suppose the number of lower priority reader tasks is n_i^{LR}. We denote the number of additional buffers needed for the wait-free communication implementation as n_i. As in [9] [10], the higher priority readers use one buffer, and all the others require, in the worst case, a total of $n_i^{LR} + 1$ buffers. Thus, if there is any higher priority reader, then $n_i = n_i^{LR} + 2$; otherwise $n_i = n_i^{LR} + 1$. This mechanism can be applied for both data consistency and flow preservation.

The implementation of the wait-free method also results in time overhead. At activation time, the writer needs to find a free buffer to store the data it will produce at runtime. In [10] a constant time implementation is presented. We denote this overhead as H_1. Since the buffer selection code is executed by the kernel at activation time, it provides interference to all tasks in the system. At execution time, the writer simply writes the data in the free buffer it has been assigned at activation time with no time overhead. Each reader is similarly assigned at activation time the buffer position from which it reads. The timing overhead is denoted as H_2. The time overhead at execution time is assumed to be negligible.

M4: (Immediate) Priority Ceiling semaphores. The other possibility is the use of immediate priority ceiling semaphores. In this case, the timing overhead is a constant H_3, and the memory overhead is zero. The use of priority ceiling semaphores also introduces blocking time in the measure of the largest critical section executed by a lower priority task on a resource also used by the task itself or a higher priority one. This mechanism does not apply to the purpose of flow preservation.

III. Definition of the Feasibility Region

The design space must be constrained to contain only the feasible solutions (for which runnables complete before their deadlines). This requires an efficient formulation of the feasibility region as well as other time constraints that apply to runnable completion times in the MILP framework.

The original response time analysis for task sets scheduled with preemption threshold was proposed in [4] and later corrected in [8]. It considers all q^* instances in the busy period of level π_i. This fact, together with the fact that the number q^* of such instances is not known a-priori, results in excessive complexity for our purposes. Thus, we look for lower and upper bounds to the region, corresponding, respectively, to sufficient-only (pessimistic) and necessary-only (optimistic) conditions for feasibility. We make use of a method for the efficient encoding of schedulability conditions in an MILP framework [11] [12].

A *sufficient condition* for the schedulability of τ_i is that τ_i is schedulable assuming it is fully preemptive, i.e., its preemption threshold is the same as its priority.

$$\bigwedge_{\tau_i \in \Gamma} \bigvee_{t \in \mathcal{I}_i} B_i + \sum_{j:\pi_j \leq \pi_i} rbf_j(t) \leq t \tag{1}$$

where $rbf_j(t)$ denotes the request bound function of τ_j within the interval of length t. The set of points $\mathcal{I}_i$ can be computed using the methods described in [11]. The blocking time B_i needs to account for the use of preemption thresholds and priority ceiling protocols.

A *necessary condition* for task τ_i to be schedulable is that the first instance in the busy period is schedulable. In this case, feasibility can be evaluated by computing the worst-case start and finish times of the first instance, respectively. Its linearization and simplification in MILP framework can be found in [12].

IV. Problem Formulation in MILP

In this formulation, we consider that the runnable to task mapping and task priority assignment are given. The designers still has the freedom to decide the execution order of runnables inside a task. We only focus on the problem of guaranteeing data consistency (thus all four mechanisms can be used) and leave the problem of flow preservation to future work. We make use of an integer- or binary-linear programming (MILP) formulation.

A. Constraints

We define a set of optimization variables associated to runnables and tasks.

Execution order relation among runnables The priority order of runnables is inherited from the priority order of the tasks they are mapped into, the priority π_i of runnable ρ_i is inherited from the priority of the task τ_h it is mapped to, i.e. $\pi_i = \Pi_h$. If two runnables are mapped to the same task, the mapping order index must match the partial order in the execution of the runnables. For each pair of runnables ρ_i and ρ_j mapped to the same task, we define an execution order relation $p_{i,j}$ between them. $p_{i,j}$ is 1 if ρ_i has a smaller execution index than ρ_j; otherwise, it is 0.

$$\begin{aligned} &\forall \rho_i \neq \rho_j, m(\rho_i, \tau_k, l) = m(\rho_j, \tau_k, n) = 1, \\ &p_{i,j} = \begin{cases} 1 & \text{if } l < n \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{2}$$

The execution order is subject to the antisymmetric and transitive properties of the execution order relation

$$\begin{aligned} &p_{i,j} + p_{j,i} = 1 \\ &p_{i,j} + p_{j,k} - 1 \leq p_{i,k} \end{aligned} \tag{3}$$

Preemption between runnables Once it starts execution, the preemption threshold of a runnable is used to check whether other runnables can preempt it. For each pair ρ_i, ρ_j, ρ_i cannot preempt ρ_j iff $\pi_i \geq \gamma_j$. A set of binary variables is used to encode this condition

$$q_{i,j} = \begin{cases} 1 & \text{if } \pi_i \geq \gamma_j \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Also, if a runnable ρ_i has priority higher than or equal to ρ_j, then ρ_j can not preempt ρ_i.

$$\forall j : \pi_j \geq \pi_i, \quad q_{j,i} = 1 \tag{5}$$

Obviously if ρ_i and ρ_j are mapped to the same task (thus $\pi_i = \pi_j$), they can not preempt each other.

If ρ_i cannot preempt ρ_j, then any runnable ρ_k with priority $\geq \pi_i$ cannot preempt ρ_j, too; conversely, if ρ_i can preempt ρ_j, any runnable with priority $\leq \pi_i$ can preempt ρ_j.

$$\begin{aligned} &\forall k : \pi_k \geq \pi_i, \quad q_{k,j} \geq q_{i,j} \\ &\forall k : \pi_k \leq \pi_i, \quad q_{k,j} \leq q_{i,j} \end{aligned} \tag{6}$$

Absence of preemption by timing analysis For any pair of runnables ρ_i and ρ_j mapped to different tasks (with different priority $\pi_i > \pi_j$), we use a binary variable to denote whether the minimum offset $o_{i,j}$ from the activation of ρ_i to the following activation of ρ_j allows to demonstrate that ρ_j cannot preempt ρ_i.

$$\begin{aligned} &\forall \rho_i, \rho_j \text{ with } \pi_i > \pi_j \\ &z_{i,j} = \begin{cases} 1 & \text{if } r_i \leq o_{i,j} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{7}$$

If $o_{i,j} \geq D_i$, then the feasibility of ρ_i implies the absence of preemption. In this case, we can set $z_{i,j}$ to be 1 and just enforce the schedulability of ρ_i with respect to its deadline.

$$\forall \rho_i, \rho_j \text{ with } \pi_i > \pi_j \text{ and } o_{i,j} \geq D_i, \quad z_{i,j} = 1 \tag{8}$$

No preemption between runnables Preemption cannot happen when:

- two runnables are mapped into the same task;
- preemption thresholds are assigned in such a way that they cannot preempt each other;
- time analysis shows there can be no preemption.

The first condition is a special case of the second. Both are captured by the binary variable $q_{i,j}$.

For each pair of runnables ρ_i and ρ_j with priority $\pi_i > \pi_j$, we use an additional set of binary variables to indicate that ρ_j does not preempt ρ_i because of: 1) timing analysis ($z_{i,j} = 1$); 2) disabling preemption by preemption thresholds ($q_{j,i} = 1$).

$$\forall \rho_i, \rho_j \text{ with } \pi_i > \pi_j$$
$$h_{i,j} = \begin{cases} 1 & \text{if } r_i \leq o_{i,j} \text{ or } q_{j,i} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$h_{i,j}$ should satisfy a set of constraints by definition

$$\begin{array}{l} h_{i,j} \leq z_{i,j} + q_{j,i} \\ h_{i,j} \geq z_{i,j}, \quad h_{i,j} \geq q_{j,i} \end{array} \tag{10}$$

Semaphore locks The set of shared resources can be protected by immediate priority ceiling semaphores. For each resource ε_k, we define a binary variable l_k to indicate whether or not it is guarded by a semaphore lock.

$$l_k = \begin{cases} 1 & \text{if } \varepsilon_k \text{ is protected by semaphore lock} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

Wait free methods For each link ε_k, we define a binary variable to indicate the use of wait-free communication

$$w_k = \begin{cases} 1 & \text{if } \varepsilon_k \text{ is protected by wait free method} \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

For each link between the writer $\rho_i \in \mathcal{E}_k^{\text{W}}$ and the low priority reader $\rho_j \in \mathcal{E}_k^{\text{LR}}$, the wait free buffer can be avoided if there is no preemption between ρ_i and ρ_j $(h_{j,i} = 1)$. We define the set of binary variables

$$\forall \rho_i \in \mathcal{E}_k^{\text{W}}, \rho_j \in \mathcal{E}_k^{\text{LR}}$$
$$f_{k,i,j} = \begin{cases} 1 & \text{if } (\rho_i, \rho_j) \text{ is protected by wait free method} \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

w_k and $f_{k,i,j}$ should be consistent with their definitions

$$f_{k,i,j} \leq 1 - h_{j,i}, \quad f_{k,i,j} \leq w_k \tag{14}$$

Providing data consistency and time determinism As discussed, there are four mechanisms to guarantee the data consistency and time determinism in the runnable to task implementation.

Thus for any shared resource $\varepsilon_k \in \mathcal{E}$, we have the following constraint

$$\begin{array}{ll} \forall \rho_i \in \rho^{\text{W}}(\varepsilon_k), \rho_j \in \rho^{\text{LR}}(\varepsilon_k), & f_{k,i,j} + l_k \geq 1 - h_{j,i} \\ \forall \rho_i \in \rho^{\text{W}}(\varepsilon_k), \rho_j \in \rho^{\text{HR}}(\varepsilon_k), & w_k + l_k \geq 1 - h_{i,j} \end{array} \tag{15}$$

For efficiency issues considering timing and overhead, we only need to choose one mechanism between wait-free and semaphore locks

$$w_k + l_k \leq 1 \tag{16}$$

If there is no preemption between the writer and any of the readers, then wait-free buffers or semaphore locks are not needed

$$w_k + l_k \leq \sum_{\rho_i \in \rho^{\text{W}}(\varepsilon_k), \rho_j \in \rho^{\text{LR}}(\varepsilon_k)} (1 - h_{j,i}) + \sum_{\rho_i \in \rho^{\text{W}}(\varepsilon_k), \rho_j \in \rho^{\text{HR}}(\varepsilon_k)} (1 - h_{i,j}) \tag{17}$$

Nonpreemption group The set of runnables can be partitioned into nonpreemption groups by assigning a preemption threshold or by proving that there is no preemption between them. For each pair of runnables ρ_i and ρ_j mapped into different tasks, we define a variable $g_{i,j}$ equal to 1 if ρ_i and ρ_j ar in the same non-preemption group, and 0 otherwise.

$$g_{i,j} = \begin{cases} 1 & \text{if } \rho_i \text{ and } \rho_j \text{ are in the same group} \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

ρ_i and ρ_j can only be in the same nonpreemption group if it is proven that there is no preemption between them or the preemption threshold is assigned in such a way that they cannot preempt each other.

$$\forall i, j \text{ with } \pi_i > \pi_j, \quad g_{i,j} \leq h_{i,j} \tag{19}$$

The nonpreemption group variable is subject to the symmetric and transitive properties

$$\begin{array}{l} g_{i,j} = g_{j,i} \\ g_{i,j} + g_{j,k} - 1 \leq g_{i,k} \end{array} \tag{20}$$

Execution time of runnables The worst case execution time of the runnable ρ_i is also dependent on the mechanism to protect the shared resources. Different mechanisms require different time overhead. For each runnable ρ_i, we define $c'_i(\varepsilon_k)$ as the execution time considering the time overhead on each link $\varepsilon_k \in \mathcal{E}_i$. It is

$$c'_i(\varepsilon_k) = C_i(\varepsilon_k) + H_3 \cdot l_k \tag{21}$$

The total execution time of ρ_i is now

$$\begin{aligned} c_i = & \; C_{i,0} + \sum_{\varepsilon_k \in \mathcal{E}_i} c'_i(\varepsilon_k) \\ = & \; C_{i,0} + \sum_{\varepsilon_k \in \mathcal{E}_i} C_i(\varepsilon_k) + H_3 \sum_{\varepsilon_k \in \mathcal{E}_i} l_k \end{aligned} \tag{22}$$

Blocking time Each runnable ρ_i can only block once, with a worst-case blocking time equal to the maximum execution time of a lower priority runnable ρ_j with a preemption threshold $\gamma_j \leq \pi_i$, and the largest critical section on a shared resource protected using priority ceiling and shared by a lower- and a higher-than-or-equal-priority tasks.

$$B_i = \max_{j:\pi_i < \pi_j} (q_{i,j} \cdot c_j, \max_{\varepsilon_k \in \mathcal{E}_i} l_k \cdot c'_j(\varepsilon_k)) \tag{23}$$

Note that $l_k \cdot l_k = l_k$, the second item $l_k \cdot c'_j(\varepsilon_k)$ in (23) can be linearized as $l_k \cdot C_j(\varepsilon_k) + l_k \cdot H_3$. However, the first item $q_{i,j} \cdot c_j$ needs to be linearized by adding an additional set of binary variables

$$\forall \rho_i, \rho_j \text{ with } \pi_i < \pi_j, \varepsilon_k \in \mathcal{E}_j$$
$$ql_{i,j,k} = \begin{cases} 1 & \text{if } q_{i,j} = 1 \text{ and } l_k = 1 \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

The variables $ql_{i,j,k}$, $q_{i,j}$ and l_k should satisfy

$$\begin{aligned} & q_{i,j} + l_k - 1 \leq ql_{i,j,k} \\ & ql_{i,j,k} \leq q_{i,j}, \;\; ql_{i,j,k} \leq l_k \end{aligned} \tag{25}$$

Thus (23) can be written in a set of MILP constraints as $\forall j : \pi_i < \pi_j$,

$$\begin{cases} B_i \geq q_{i,j} \cdot C_{j,0} + q_{i,j} \sum_{\varepsilon_k \in \mathcal{E}_j} c_j(\varepsilon_k) + H_3 \sum_{\varepsilon_k \in \mathcal{E}_j} ql_{i,j,k} \\ \forall \varepsilon_k \in \mathcal{E}_j, B_i \geq l_k \cdot c_j(\varepsilon_k) + l_k \cdot H_3 \end{cases} \tag{26}$$

Kernel level timing overhead Wait free methods require the execution of several procedure at task activation time, with the highest priority in the system. These procedures are executed at the activation time of the runnables, with their period.

The request bound function during the time interval t of these kernel level overhead for shared resource ε_k can be formulated as

$$\begin{aligned} rbf_0(\varepsilon_k, t) = & \sum_{\rho_i \in \rho^{\mathrm{W}}(\varepsilon_k)} (w_k \cdot \left\lceil \frac{t}{t_i} \right\rceil H_1 \\ & + \sum_{\rho_j \in \rho^{\mathrm{LR}}(\varepsilon_k)} f_{k,i,j} \cdot \left\lceil \frac{t}{t_j} \right\rceil H_2 + \sum_{\rho_j \in \rho^{\mathrm{HR}}(\varepsilon_k)} w_k \left\lceil \frac{t}{t_j} \right\rceil \cdot H_2) \end{aligned} \tag{27}$$

The total request bound function for all the shared resources is

$$rbf_0(t) = \sum_{\varepsilon_k \in \mathcal{E}} rbf_0(\varepsilon_k, t) \tag{28}$$

Real-time Schedulability To verify the schedulability of ρ_j, we check whether there exists a point $t \in \mathcal{I}_j$ such that the sum of the possible execution requests within the time interval t is no larger than the available CPU time. The possible execution requests include:

1) B_j: worst case blocking time;
2) $rbf_0(t)$: kernel-level timing overhead;
3) $rbf_j(t)$: the computation time c_j of ρ_j (as $t \leq T_j$);
4) $rbf_i(t), \forall i$ with $\pi_i < \pi_j$: the sum of the interferences from blocks ρ_i with higher priority, which is
$$\sum_{i:\pi_i < \pi_j} \lceil \frac{t}{t_i} \rceil \cdot c_i \tag{29}$$
5) $rbf_i(t), \forall i$ with $\pi_i = \pi_j$: the sum of the interferences from blocks ρ_i mapped to the same task, which is
$$\sum_{i:\pi_i = \pi_j} p_{i,j} \lceil \frac{t}{t_i} \rceil \cdot c_i \tag{30}$$

However, in (30) it contains the product of two variables $p_{i,j}$ and c_i. By (22), c_i is a linear function of y_k and l_k for each input and output link ε_k of ρ_i. We define the following two variables to make the constraint (30) linear:

$$\begin{aligned} & \forall \rho_j \neq \rho_i, \varepsilon_k \in \mathcal{E}_i^{\mathrm{in}} \bigcup \mathcal{E}_i^{\mathrm{out}} \\ & v_{i,j,k} = \begin{cases} 1 & \text{if } p_{i,j} = 1 \text{ and } y_k = 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{31}$$

$v_{i,j,k}$ should satisfy the following constraints:

$$\begin{aligned} & p_{i,j} + y_k - 1 \leq v_{i,j,k} \\ & v_{i,j,k} \leq p_{i,j}, \;\; v_{i,j,k} \leq y_k \end{aligned} \tag{32}$$

Similarly,

$$\begin{aligned} & \forall \rho_i \neq \rho_j, \varepsilon_k \in \mathcal{E}_i^{\mathrm{in}} \bigcup \mathcal{E}_i^{\mathrm{out}} \\ & w_{i,j,k} = \begin{cases} 1 & \text{if } p_{i,j} = 1 \text{ and } l_k = 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{33}$$

$w_{i,j,k}$ should satisfy the following constraints:

$$\begin{aligned} & p_{i,j} + l_k - 1 \leq w_{i,j,k} \\ & w_{i,j,k} \leq p_{i,j}, \;\; w_{i,j,k} \leq l_k \end{aligned} \tag{34}$$

Stack usage The stack usage of the system includes:

- the fixed stack usage S_i of each task τ_i;
- the maximum possible stack usage of runnables because of preemption.

We order the runnables according to their decreasing usage of stack:

$$o : \rho_i \to \mathbb{N}^+ \tag{35}$$

such that $o(\rho_i) < o(\rho_j) \Rightarrow S_i \geq S_j$.

We define the following binary variable

$$u_i = \begin{cases} 1 & \text{if } \rho_i \text{ has the largest stack size} \\ & \text{in the nonpreemption group} \\ 0 & \text{otherwise} \end{cases} \tag{36}$$

u_i is dependent on $g_{i,j}$ and should satisfy

$$\begin{aligned} & 1 - \textstyle\sum_{j:o(\rho_j) \leq o(\rho_i)} g_{i,j} \leq u_i \\ & u_i \leq 1 - g_{i,j}, \forall j : o(\rho_j) \leq o(\rho_i) \end{aligned} \tag{37}$$

The maximum stack usage is

$$s = \textstyle\sum_{\tau_i \in \mathcal{T}} S_i + \sum_{\rho_i \in \rho} S_i \cdot u_i \tag{38}$$

Memory constraints The memory cost of the additional wait free buffers for resource ε_k is

$$n_k = \begin{cases} \sum_{\rho_i \in \rho_k^{\mathrm{W}}, \rho_j \in \rho_k^{\mathrm{LR}}} f_{k,i,j} + 2w_k & \text{if } \rho_k^{\mathrm{HR}} \neq \emptyset \\ \sum_{\rho_i \in \rho_k^{\mathrm{W}}, \rho_j \in \rho_k^{\mathrm{LR}}} f_{k,i,j} + w_k & \text{if } \rho_k^{\mathrm{HR}} = \emptyset \end{cases} \tag{39}$$

When adding the base memory requirements of the application M_A, the overall required memory, including the stack used by runnables and tasks is

$$m = M_A + \sum_{\varepsilon_k \in \mathcal{E}} M_k \cdot n_k + s \tag{40}$$

B. Objective Function

In addition to satisfying the constraints, we can also minimize the memory usage considering stack and overhead introduced by mechanisms to ensure data consistency and timing determinism.

$$\text{minimize} \quad m \tag{41}$$

V. Experimental Results

We implemented our MILP approach in AMPL (A Mathematical Programming Language) and used CPLEX as the solver. The experiments are performed on an industrial case study consisting of a fuel injection embedded controller. The case study is a simplified version of the full control system (for confidentiality reasons) with 90 runnables (out of 200 in the real system).

The runnables are mapped into 16 tasks, as shown in Table I. The execution times of some functions are provided as part of the case study. The others are assigned to achieve a system utilization of 94.1%, which is close to the values found in real systems of this type.

Task	Period(ms)	Priority	$C_i(\mu s)$	NW	NLPR	NHPR	Stack (bytes)
τ_0	1000	6	1500	4	0	0	512
τ_1	1000	7	5000	4	3	0	704
τ_2	8	3	148	4	0	0	128
τ_3	4	0	208	4	0	1	256
τ_4	8	4	100	3	0	2	608
τ_5	1000	15	131100	3	2	0	640
τ_6	1000	11	150000	3	2	1	768
τ_7	8	1	340	4	1	12	608
τ_8	5	5	5	6	1	1	448
τ_9	1000	12	110000	3	14	2	768
τ_{10}	1000	14	110000	3	13	2	640
τ_{11}	4	2	39	2	4	18	288
τ_{12}	12	9	820	2	10	6	1024
τ_{13}	50	8	1000	0	0	0	160
τ_{14}	100	10	9846	1	11	6	544
τ_{15}	1000	13	110000	0	29	4	736

Table I
List of tasks in the automotive fuel injection application

The first three columns of Table I are task indices, periods and priorities. Periods and priorities are taken from the automotive application. The runnables are executing at 7 different periods (in ms) in the example: 4, 5, 8, 12, 50, 100 and 1000. Columns 5, 6, and 7 represent the numbers of writers (output ports), lower-priority readers (input ports connected with higher-priority writers), and higher-priority readers (input ports connected with lower-priority writers) respectively that the task implements. In the information available from the real application, the communication topology was only defined as communication flows among the components. Based on these, we made assumptions about the estimated communication among runnables and finally among tasks, thereby completing the definition of the communication topology. The communication link delays are assumed to be one from low-priority writers to high-priority readers and zero otherwise. There are 46 writers and 145 readers (90 lower-priority readers and 55 higher-priority readers) in the derived example.

Using the formulation corresponding to the reduced set presented in this paper, *the optimal solution can be found by the MILP solver in 14677 seconds, or about 4 hours.* Our optimization framework requires 69% less memory to guarantee data consistency compared to commercial tools such as [2]. The reason is that we selectively disable the preemption among runnables while still guarantee the system's real-time schedulability, which enables the sharing of stack space.

VI. Conclusion

We presented an algorithm for optimizing the implementation of AUTOSAR runnables in a concurrent program executing as a set of tasks. We showed that there is an opportunity for optimizing the memory requirements (including stack usage and communication buffers) when implementing a model. The solution is based on an MILP optimization framework that explores the design/implementation space while trying to share the stack and avoid additional communication buffers whenever possible. We plan to propose fast heuristics and demonstrate that they yield a solution with close to minimal memory usage while satisfying real-time schedulability constraints.

References

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

[2] *SystemDesk*, dSPACE Inc., web page: http://dspaceinc.com.

[3] A. Ferrari, M. D. Natale, G. Gentile, G. Reggiani, and P. Gai, "Time and memory tradeoffs in the implementation of autosar components," in *Proceedings of the DATE conference*, 2009.

[4] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, 1999.

[5] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proceedings of the 21rd IEEE Real-Time Systems Symposium*, 2000, pp. 25–34.

[6] R. Ghattas and A. G. Dean, "Preemption threshold scheduling: Stack optimality, enhancements and analysis," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, 2007, pp. 147–157.

[7] J. Chen, A. Harji, and P. Buhr, "Solution space for fixed-priority with preemption threshold," in *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, 2005, pp. 385–394.

[8] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002, pp. 315–326.

[9] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *Proceedings of the 6th International conference on Embedded software*, 2006.

[10] M. Di Natale, G. Wang, and A. Sangiovanni-Vincentelli, "Improving the size of communication buffers in synchronous models with time constraints," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 3, August 2009.

[11] H. Zeng and M. D. Natale, "Improving real-time feasibility analysis for use in linear optimization methods," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*.

[12] H. Zeng and M. Di Natale, "An Efficient Formulation of the Real-time Feasibility Region for Design Optimization," in *submission to IEEE Transactions on Computers*.