Efficient Implementation of AUTOSAR Components with Minimal Memory Usage

Haibo Zeng
McGill University, haibo.zeng@mcgill.ca

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

Abstract—The adoption of AUTOSAR in the development of automotive electronics can increase the portability and reuse of functional components. Inside each component, the behavior is represented by a set of runnables, defining reactions executed in response to an event or periodic computations. 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 to 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 the end, the mapping of runnables into tasks takes the shape of Figure 1.

In this paper, we deal with timing issues at the local level, that is, for components mapped into tasks executing on the same CPU. The mapping of runnables into tasks, the configuration of the task model, the selection of the right mechanisms for the implementation of the communication over ports (protecting against data inconsistency and possibly flow semantics violations) has a large impact on the performance of the system and defines its correct implementation. Current tools do not leverage any timing information in the definition of the mapping. As such, they often use general rules for the mapping of runnables and the definition of the communication mechanisms that may easily result in inefficient implementations. One example is a pipeline of runnables exchanging activation signals. Such runnables could be easily mapped into a single task, provided they are executed in the right order. However, existing tools [2] translate this model into a set of tasks, each one sending an activation event to its successor, with unnecessary context switch overhead.

Of course, context switch overheads should be considered when defining the mapping, and the reduction of such overheads is among the main drivers for the mapping scheme presented in [9], but many other design parameters come into play. An initial discussion of the possible options was provided in [6]. To summarize, the mapping of runnables into tasks and the assignment of priorities to tasks determine the task and runnable response times against the deadlines. However, preemption can be selectively disabled or preemption thresholds can be used for the purpose of improving schedulability of lower priority threads [13] [11] and, for task sets with arbitrary offsets, [7] to optimize the requirements of system memory for stack space. Algorithms to optimize the assignment of preemption thresholds for minimizing the use of stack space are presented in [11]. In [14] a functional model is considered in which runnables are already mapped into tasks and the priority of the tasks is given, and preemption thresholds can be
assigned to runnables. The maximum amount of blocking that can be tolerated by each runnable is computed and the stack requirement is minimized by iteratively increasing the preemption threshold of the runnables as much as possible, starting from those belonging to the second-highest priority task, as long as the task set remains schedulable, with an algorithm very similar to the one in [11]. When scheduling offsets are known, they can be exploited to further improve the analysis and the definition of the threshold levels, as discussed in [8] [3].

Of course, disabling preemption comes at the price of possible task blocking and must be carefully selected to allow high priority tasks to execute within their deadlines. Also, in the selection of the communication mechanism and the protocol to protect state variables several options are possible, which typically result in tradeoffs between time overhead for the execution of the protocol, memory required for the implementation of the mechanism and possible blocking times.

In this work, using the AUTOSAR model and definitions, we consider the case in which the runnable to task mapping and the task priority assignment are given, and present a scheme for the optimal selection of

- the execution order of runnables mapped into a task
- the assignment of preemption thresholds to runnables
- 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 $G = (V, E)$, where $V$ is the set of vertices, representing the runnables, and $E$ 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:

$\rho = \{\rho_1, \ldots, \rho_p\}$ is the set of runnables. A runnable $\rho_i$ reads from a set of input ports, denoted as $E_i^\text{in}$, and a set of output ports, denoted as $E_i^\text{out}$. 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 $\rho_i$ as $E_i = E_i^\text{in} \cup E_i^\text{out}$. Each runnable is characterized by a worst-case stack requirement $\sigma_i$ (in bytes) needed for its execution.

$E = \{e_1, \ldots, e_E\}$ is the set of shared resources. We consider the case of one-to-many communication: a shared resource $e_i$ has a writer runnable, denoted as $\rho^W(e_i)$, and a set of reader runnables $\rho^R(e_i)$. We also denote the set of readers with higher (lower) priority than the reader $\rho^W(e_i)$ as $\rho^{HR}(e_i)$ ($\rho^{LR}(e_i)$). $M_i$ denotes the size of the data communicated over $e_i$.

The execution time of a runnable $\rho_i$ is characterized by $(C_{i,0}, C_{i,1}, \ldots, C_{i,|E_i^\text{in}|}, \ldots, C_{i,|E_i^\text{in}|+|E_i^\text{out}|})$, where

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

We also use $C_i(e_k)$ to denote the worst case execution time of $\rho_i$ accessing the input/output port $e_k$, $\forall e_k \in E_i^\text{in} \cup E_i^\text{out}$. The worst case execution time $c_i$ of the runnable $\rho_i$ also depends on the time overhead of the mechanism used to protect the shared resources.

$T = \{\tau_1, \ldots, \tau_T\}$ is the set of tasks. Each task $\tau_i$ has a priority $\Pi_i$ and an activation period $T_i$. The lower the
number, the higher the priority. Each task is periodic with an offset (equal to zero, thus all tasks start at the same time instant \( t = 0 \)). It is also characterized by \( s_{i,0} \), a quantity of memory that is required for the stack space to switch between its runnables.

A mapping relation \( m(\rho_i, \tau_j, k) \) may be defined between a runnable \( \rho_i \) and a task \( \tau_j \) meaning that the code implementing the runnable \( \rho_i \) is executed in the context of task \( \tau_j \) with ordering index \( k \). A mapping relation is only possible if the execution period of \( \rho_i \) is an integer multiple of \( \tau_j \), i.e. \( t_i = k_i \cdot T_j \) for some integer \( k_i \). The deadline of the \( \rho_i \) is defined as the period of the task \( \tau_j \) it is mapped to, thus \( D_i = T_j \), which is no greater than the period \( t_i \) of \( \rho_i \).

The priority order of runnables is inherited from the priority order of the tasks they are mapped into, the priority \( \pi_i \) of runnable \( \rho_i \) is inherited from the priority of the task \( \tau_i \) 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 \( \pi_i \), a runnable \( \rho_i \) is also assigned a preemption threshold \( \gamma_i \) with \( \pi_i \geq \gamma_i \) \([13]\). 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 \([6]\), 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 \( \rho_i \) and \( \rho_j \) mapped to different tasks, with priority \( \pi_i > \pi_j \), we denote the minimum offset from the activation of \( \rho_i \) to the following activation of \( \rho_j \) as \( \alpha_{i,j} \). If the worst case response time \( \tau_i \) of \( \rho_i \) is no greater than \( \alpha_{i,j} \), then there can be no preemption from \( \rho_j \) to \( \rho_i \). 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 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 \( \epsilon_i \), suppose the number of lower priority reader tasks is \( n_i^{L,R} \). We denote the number of additional buffers needed for the wait-free communication implementation as \( n_i \). As in \([12, 5]\), the higher priority readers use one buffer, and all the others require, in the worst case, a total of \( n_i^{L,R} + 1 \) buffers. Thus, if there is any higher priority reader, then \( n_i = n_i^{L,R} + 2 \); otherwise \( n_i = n_i^{L,R} + 1 \). 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 \([5]\) 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 mixed integer linear programming (MILP) framework.

The original response time analysis for task sets scheduled with preemption threshold was proposed in \([13]\) and later corrected in \([10]\). It considers all \( q^* \) instances in the busy period of level \( \pi_i \). 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 \([15, 16]\).

Tasks are scheduled with two different modes. When ready, they are scheduled with their priority, but as soon as they start execution, they behave as if their priority is raised to the preemption threshold. Accordingly, the analysis is performed in two steps. First, the worst-case start time for a generic task \( \tau_i \) needs to be computed. According to \([13]\), the worst case start time \( S_i \) of the first instance of \( \tau_i \) can be calculated iteratively by the following,

\[
S_i = B_i + \sum_{j \in h well(i)} (1 + \left\lfloor \frac{S_i}{T_j} \right\rfloor) C_j
\] (1)
Once $S_i$ is computed, the worst case finish time $F_i$ of the first instance of $\tau_i$ is

$$F_i = S_i + C_i + \sum_{\tau_j > \tau_i} \left( \left\lceil \frac{F_j}{T_j} \right\rceil - 1 - \left\lfloor \frac{S_j}{T_j} \right\rfloor \right) C_j + qC_i \quad (2)$$

Unfortunately (as explained in [10]), the first instance is not necessarily the one resulting in the largest response time, but all instances in the busy period of level $\pi_i$ need to be considered. Hence, the previous (1) and (2) need to be changed as follows. The start time for the first $q$ instances in the busy period $q = 0, 1, \ldots, q^*$ is computed as

$$S_i(q) = B_i + \sum_{j \in h(p(i))} (1 + \left\lfloor \frac{S_i(q)}{T_j} \right\rfloor) C_j + qC_i \quad (3)$$

correspondingly, the finish time of the $q$-th instance is

$$F_i(q) = S_i(q) + C_i + \sum_{\tau_j > \tau_i} \left( \left\lceil \frac{F_i(q)}{T_j} \right\rceil - 1 - \left\lfloor \frac{S_i(q)}{T_j} \right\rfloor \right) C_j + qC_i \quad (4)$$

The worst-case response time of $\tau_i$ is computed as

$$R_i = \max_{q=0, \ldots, q^*} \{ R_i(q) = F_i(q) - qT_i \}$$

where the largest index $q^* = \left\lceil \frac{L_i}{T_i} \right\rceil$ to be considered is the number of instances in the level-$i$ busy period $L_i$, and $L_i$ can be calculated by the following iterative formula

$$L_i = B_i + \sum_{j \in h(p(i)) \cup \{ i \}} \left\lceil \frac{L_j}{T_j} \right\rceil C_j \quad (5)$$

From (3) and (4), the amount of workload from higher priority tasks before the completion of $\tau_i$ depends on the finish time $F_i$ as well as the start time $S_i$.

The need to consider the first $q^*$ instances, together with the fact that the number $q^*$ of such instances is not known a-priori results in excessive complexity for our purposes. Given that the linearization of the exact feasibility region is probably exceedingly complex, we look for lower and upper bounds to the region, corresponding, respectively, to sufficient-only (pessimistic) and necessary-only (optimistic) conditions for feasibility.

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

$$\bigwedge_{\tau_i \in T} \bigvee_{t \in T_i} B_i + \sum_{j: \pi_j \leq \pi_i} rbf_j(t) \leq t \quad (5)$$

where $rbf_j(t) = \left\lceil \frac{t}{T_j} \right\rceil C_j$ denotes the request bound function of $\tau_j$ within the interval of length $t$. The set of points $T_i$ can be computed using the methods described in [15]. The blocking time $B_i$ needs to account for the use of preemption thresholds and priority ceiling protocols.

A necessary condition for task $\tau_i$ 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 [16].

The sufficient condition provides a much easier formulation, but its use may easily result in a sub-optimal solution. The necessary condition is definitely more complex, but the small difference between the necessary and exact feasibility regions (the average difference is 0.021%, and the maximum is 6.1% for randomly generated task sets [16]) indicate a very good chance that the solution obtained using the necessary-only condition is also feasible with respect to the exact test. In this work, we employ the heuristic that we first use the necessary-only feasibility condition and check the returned result against the exact test. If the result is feasible, then it is also optimal. Otherwise, the sufficient condition is used in the formulation and we settle with a possibly sub-optimal solution.

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 a mixed integer linear programming (MILP) formulation. An MILP program in standard form is:

minimize \quad c^T x

subject to \quad Ax = b

\quad x \geq 0

where $x = (x_1, \ldots, x_n)$ is a vector of positive real, integer, or binary-valued decision variables. Constraints of the type $Ax \leq b$ can be handled by adding a suitable set of variables. MILPs can be solved very efficiently by a variety of solvers such as CPLEX.

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 $\pi_i$ of runnable $\rho_i$ is inherited from the priority of the task $\tau_i$ 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 $\rho_i$ and $\rho_j$ mapped to the same task, we define an execution order relation $p_{i,j}$ between them. $p_{i,j}$ is 1 if $\rho_i$
has a smaller execution index than \( \rho_j \); otherwise, it is 0.

\[
\forall i \neq j, m(p_i, \tau_k, l) = m(\rho_j, \tau_k, n) = 1, \\
p_{i,j} = \begin{cases} 
1 & \text{if } l < n \\
0 & \text{otherwise}
\end{cases}
\]  

(7)

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

\[
p_{i,j} + p_{j,i} = 1 \\
p_{i,j} + p_{j,k} - 1 \leq p_{i,k}
\]  

(8)

Preemption threshold assignment.

Once it starts execution, the preemption threshold of a runnable is used to check whether other runnables can preempt it. For each pair \( \rho_i, \rho_j, \rho_i \) cannot preempt \( \rho_j \) if and only if \( \pi_i \geq \gamma_j \). A set of binary variables is used to encode this condition

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

(9)

Also, if a runnable \( \rho_i \) has priority higher than or equal to \( \rho_j \), then \( \rho_j \) cannot preempt \( \rho_i \).

\[
\forall j : \pi_j \geq \pi_i, \quad q_{j,i} = 1
\]  

(10)

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

If \( \rho_i \) cannot preempt \( \rho_j \), then any runnable \( \rho_k \) with priority \( \pi_k \geq \pi_i \) cannot preempt \( \rho_j \), too; conversely, if \( \rho_i \) can preempt \( \rho_j \), any runnable with priority \( \pi_i \leq \pi_j \) can preempt \( \rho_j \).

\[
\forall k : \pi_k \geq \pi_i, \quad q_{k,j} \geq q_{i,j} \\
\forall k : \pi_k \leq \pi_i, \quad q_{k,j} \leq q_{i,j}
\]  

(11)

Absence of preemption by timing analysis.

For any pair of runnables \( \rho_i \) and \( \rho_j \) mapped to different tasks (with different priority \( \pi_i > \pi_j \)), we use a binary variable to denote whether the minimum offset \( \alpha_{i,j} \) from the activation of \( \rho_i \) to the following activation of \( \rho_j \) allows to demonstrate that \( \rho_j \) cannot preempt \( \rho_i \).

\[
\forall \rho_i, \rho_j \text{ with } \pi_i > \pi_j, \quad z_{i,j} = \begin{cases} 
1 & \text{if } r_i \leq \alpha_{i,j} \\
0 & \text{otherwise}
\end{cases}
\]  

(12)

If \( \alpha_{i,j} \geq D_{i} \), then the feasibility of \( \rho_i \) implies the absence of preemption. In this case, we can set \( z_{i,j} \) to be 1 and just enforce the schedulability of \( \rho_i \) with respect to its deadline.

\[
\forall \rho_i, \rho_j \text{ with } \pi_i > \pi_j \text{ and } \alpha_{i,j} \geq D_{i}, \quad z_{i,j} = 1
\]  

(13)

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

\[
\forall \rho_i, \rho_j \text{ with } \pi_i > \pi_j, \quad h_{i,j} = \begin{cases} 
1 & \text{if } r_i \leq \alpha_{i,j} \text{ or } q_{j,i} = 1 \\
0 & \text{otherwise}
\end{cases}
\]  

(14)

\( h_{i,j} \) should satisfy a set of constraints by definition

\[
h_{i,j} \leq z_{i,j} + q_{j,i} \quad h_{i,j} \geq z_{i,j}, \quad h_{i,j} \geq q_{j,i}
\]  

(15)

Semaphore locks.

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

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

(16)

Wait free methods.

For each link \( \varepsilon_k \), we define a binary variable to indicate the use of wait-free communication

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

(17)

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

\[
\forall \rho_i \in \varepsilon_k^{w}, \rho_j \in \varepsilon_k^{LR}, \quad f_{k,i,j} = \begin{cases} 
1 & \text{if } (\rho_i, \rho_j) \text{ is protected by wait free method} \\
0 & \text{otherwise}
\end{cases}
\]  

(18)

\( w_k \) and \( f_{k,i,j} \) should be consistent with their definitions

\[
f_{k,i,j} \leq 1 - h_{j,i}, \quad f_{k,i,j} \leq w_k
\]  

(19)

Providing data consistency.

As discussed, there are four mechanisms to guarantee the data consistency in the runnable to task implementation. Thus for any shared resource \( \varepsilon_k \in \mathcal{E} \), we have the following constraint

\[
\forall \rho_i \in \rho_k^{w}(\varepsilon_k), \rho_j \in \rho_k^{LR}(\varepsilon_k), \quad f_{k,i,j} + l_k \geq 1 - h_{j,i} \\
\forall \rho_i \in \rho_k^{w}(\varepsilon_k), \rho_j \in \rho_k^{HR}(\varepsilon_k), \quad w_k + l_k \geq 1 - h_{j,i}
\]  

(20)

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

\[
w_k + l_k \leq 1
\]  

(21)
If there is no preemption between the writer and any of the readers, then wait-free buffers or semaphore locks are not needed

\[ w_k + l_k \leq \sum_{\rho_i \in \rho^w(\varepsilon_k) \cap \rho^h(\varepsilon_k)} (1 - h_{j,i}) + \sum_{\rho_i \in \rho^w(\varepsilon_k) \cap \rho^h(\varepsilon_k)} (1 - h_{i,j}) \]  

(22)

**Non-preemption group.**

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

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

(23)

\( \rho_i \) and \( \rho_j \) can only be in the same non-preemption 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.

\[ \forall i, j \text{ with } \pi_i > \pi_j, \quad g_{i,j} \leq h_{i,j} \]  

(24)

The non-preemption group variable is subject to the symmetric and transitive properties

\[ g_{i,j} = g_{j,i} \quad g_{i,j} + g_{j,k} - 1 \leq g_{i,k} \]  

(25)

**Execution time of runnables.**

The worst case execution time of the runnable \( \rho_i \) is also dependent on the mechanism to protect the shared resources. Different mechanisms require different time overhead. For each runnable \( \rho_i \), 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

\[ c'_i(\varepsilon_k) = C_i(\varepsilon_k) + H_3 \cdot l_k \]  

(26)

The total execution time of \( \rho_i \) is now

\[ c_i = C_{i,0} + \sum_{\varepsilon_k \in \mathcal{E}_i} c'_i(\varepsilon_k) \]

\[ = C_{i,0} + \sum_{\varepsilon_k \in \mathcal{E}_i} C_i(\varepsilon_k) + H_3 \sum_{\varepsilon_k \in \mathcal{E}_i} l_k \]  

(27)

**Blocking time.**

Each runnable \( \rho_i \) can only block once, with a worst-case blocking time equal to the maximum execution time of a lower priority runnable \( \rho_j \) 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.

\[ B_i = \max_{j: \pi_i < \pi_j} \left( g_{i,j} \cdot c_j, \max_{\varepsilon_k \in \mathcal{E}_i} l_k \cdot c'_i(\varepsilon_k) \right) \]  

(28)

Note that \( l_k \cdot l_k = l_k \), the second item \( l_k \cdot c'_i(\varepsilon_k) \) in (28) can be linearized as \( l_k \cdot C_j(\varepsilon_k) + l_k \cdot H_3 \). However, the first item \( g_{i,j} \cdot c_j \) needs to be linearized by adding an additional set of binary variables

\[ \forall \rho_i, \rho_j \text{ with } \pi_i < \pi_j, \varepsilon_k \in \mathcal{E}_j \]

\[ q_{i,j,k} = \begin{cases} 1 & \text{if } g_{i,j} = 1 \text{ and } l_k = 1 \\ 0 & \text{otherwise} \end{cases} \]  

(29)

The variables \( q_{i,j,k}, \rho_i, \rho_j \) and \( l_k \) should satisfy

\[ g_{i,j} + l_k - 1 \leq q_{i,j,k} \]

\[ q_{i,j,k} \leq q_{i,j}, \quad q_{i,j,k} \leq l_k \]  

(30)

Thus (28) can be written in a set of MILP constraints as

\[ \forall j : \pi_i < \pi_j, \]

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

(31)

**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 \( \varepsilon_k \) can be formulated as

\[ rb_{f_0}(\varepsilon_k, t) = \sum_{\rho_i \in \rho^w(\varepsilon_k)} (w_k \cdot \left\lceil \frac{t}{l_i} \right\rceil H_1) \]

\[ + \sum_{\rho_j \in \rho^h(\varepsilon_k)} f_{k,i,j} \cdot \left\lceil \frac{t}{l_j} \right\rceil H_2 + \sum_{\rho_j \in \rho^h(\varepsilon_k)} w_k \left\lceil \frac{t}{l_j} \right\rceil H_2 \]  

(32)

The total request bound function for all the shared resources is

\[ rb_{f_0}(t) = \sum_{\varepsilon_k \in \mathcal{E}} rb_{f_0}(\varepsilon_k, t) \]  

(33)

**Real-time Schedulability.**

To verify the schedulability of \( \rho_j \), 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) \( rb_{f_0}(t) \): kernel-level timing overhead;
3) \( rb_{f_j}(t) \): the computation time \( c_j \) of \( \rho_j \) (as \( t \leq T_j \));
4) \( rb_{f_i}(t), \forall i \) with \( \pi_i < \pi_j \): the sum of the interferences from blocks \( \rho_i \) with higher priority, which is

\[ \sum_{i:m_i < \pi_j} \left\lceil \frac{t}{l_j} \right\rceil \cdot c_i \]  

(34)

5) \( rb_{f_i}(t), \forall i \) with \( \pi_i = \pi_j \): the sum of the interferences from blocks \( \rho_i \) mapped to the same task, which is
However, in (35) it contains the product of two variables \( p_{i,j} \) and \( c_i \). By (27), \( c_i \) is a linear function of \( y_k \) and \( l_k \) for each input and output link \( \xi_k \) of \( p_i \). We define the following two variables to make the constraint (35) linear:

\[
\forall \rho_j \neq \rho_i, \xi_k \in E_{\text{in}} \cup E_{\text{out}}
\]

\[
v_{i,j,k} = \begin{cases} 1 & \text{if } p_{i,j} = 1 \text{ and } y_k = 1 \\ 0 & \text{otherwise} \end{cases}
\]

\[
v_{i,j,k} \text{ should satisfy the following constraints:}
\]

\[
p_{i,j} + y_k - 1 \leq v_{i,j,k} \leq yi_{j,k} \quad v_{i,j,k} \leq p_{i,j} \quad v_{i,j,k} \leq y_k
\]

Similarly,

\[
\forall \rho_j \neq \rho_i, \xi_k \in E_{\text{in}} \cup E_{\text{out}}
\]

\[
w_{i,j,k} = \begin{cases} 1 & \text{if } p_{i,j} = 1 \text{ and } l_k = 1 \\ 0 & \text{otherwise} \end{cases}
\]

\[
w_{i,j,k} \text{ should satisfy the following constraints:}
\]

\[
p_{i,j} + l_k - 1 \leq w_{i,j,k} \leq w_{i,j,k} \leq p_{i,j} \leq l_k
\]

**Stack usage.**

The stack usage of the system includes:

- the fixed stack usage \( s_{i,0} \) of each task \( \tau_i \);
- the maximum possible stack usage of runnables because of preemption.

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

\[
o : \rho_i \rightarrow \mathbb{N}^+
\]

such that \( o(\rho_i) < o(\rho_j) \Rightarrow \sigma_i \geq \sigma_j \).

We define the following binary variable

\[
u_i = \begin{cases} 1 & \text{if } \rho_i \text{ has the largest stack size} \\ 0 & \text{in the non-preemption group} \end{cases}
\]

\[
u_i \text{ is dependent on } g_{i,j} \text{ and should satisfy}
\]

\[
1 - \sum_{j : o(\rho_j) \leq o(\rho_i)} g_{i,j} \leq u_i
\]

\[
u_i \leq 1 - g_{i,j} \forall j : o(\rho_j) \leq o(\rho_i)
\]

The maximum stack usage is

\[
s = \sum_{\tau_i \in T} s_{i,0} + \sum_{\rho_i \in \rho} \sigma_i \cdot u_i
\]

**Memory constraints.**

The memory cost of the additional wait free buffers for resource \( e_k \) is

\[
n_k = \begin{cases} \sum_{\rho_j \in \rho^k} f_{k,i,j} + 2w_k & \text{if } \rho_k^H \neq \emptyset \\ \sum_{\rho_j \in \rho^k} f_{k,i,j} + w_k & \text{if } \rho_k^H = \emptyset \end{cases}
\]

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

\[
m = M_A + \sum_{e_k \in E} M_k \cdot n_k + s
\]

**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.

\[
\text{minimize } m
\]

**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 (for confidentiality reasons) with 90 runnables (out of 200 in the real system).

<table>
<thead>
<tr>
<th>Task</th>
<th>Period(msec)</th>
<th>Priority</th>
<th>( C_{1}[\mu s] )</th>
<th>NW</th>
<th>NLPR</th>
<th>NHP</th>
<th>Stack (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_0 )</td>
<td>1000</td>
<td>0</td>
<td>5000</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>512</td>
</tr>
<tr>
<td>( t_1 )</td>
<td>1000</td>
<td>3</td>
<td>150000</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>704</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>8</td>
<td>3</td>
<td>148</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>128</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>4</td>
<td>0</td>
<td>208</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>8</td>
<td>4</td>
<td>100</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>608</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>1000</td>
<td>15</td>
<td>131100</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>640</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>1000</td>
<td>11</td>
<td>150000</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>768</td>
</tr>
<tr>
<td>( t_7 )</td>
<td>8</td>
<td>1</td>
<td>340</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>608</td>
</tr>
<tr>
<td>( t_8 )</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>448</td>
</tr>
<tr>
<td>( t_9 )</td>
<td>1000</td>
<td>12</td>
<td>110000</td>
<td>3</td>
<td>14</td>
<td>2</td>
<td>768</td>
</tr>
<tr>
<td>( t_{10} )</td>
<td>1000</td>
<td>14</td>
<td>110000</td>
<td>3</td>
<td>12</td>
<td>2</td>
<td>640</td>
</tr>
<tr>
<td>( t_{11} )</td>
<td>4</td>
<td>0</td>
<td>39</td>
<td>2</td>
<td>4</td>
<td>18</td>
<td>288</td>
</tr>
<tr>
<td>( t_{12} )</td>
<td>12</td>
<td>9</td>
<td>620</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>1024</td>
</tr>
<tr>
<td>( t_{13} )</td>
<td>50</td>
<td>8</td>
<td>10000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>160</td>
</tr>
<tr>
<td>( t_{14} )</td>
<td>100</td>
<td>10</td>
<td>9846</td>
<td>1</td>
<td>11</td>
<td>6</td>
<td>544</td>
</tr>
<tr>
<td>( t_{15} )</td>
<td>1000</td>
<td>13</td>
<td>110000</td>
<td>0</td>
<td>29</td>
<td>4</td>
<td>736</td>
</tr>
</tbody>
</table>

**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 ns) 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. We also plan to consider the requirement that semantic properties of the functional model need to be preserved.

REFERENCES