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 pre-emption 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:

$\rho = \{\rho_1, \ldots, \rho_n\}$ 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 $C_i = E_i^{\text{in}} \cup E_i^{\text{out}}$.

$$E = \{ e_1, \ldots, e_{|E|} \}$$

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 $\rho^H(e_i)$ ($\rho^L(e_i)$). $\mathcal{M}_i$ denotes the size of the data communicated over $e_i$.

The execution time of a runnable $\rho_i$ is characterized by $(C_{i,0}, C_{i,1}, \ldots, C_{i,|E_i^{\text{in}}|}, C_{i,|E_i^{\text{in}}|+|E_i^{\text{out}}|})$, 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 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. Therefore, 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 assigned the time overhead as $H_2$. The time overhead at execution time is assumed to be negligible.

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 $q^t$ instances in the busy period of level $\pi_i$. This fact, together with the fact that the number $q^t$ 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 $\pi_i$ is that $\tau_i$ is schedulable assuming it is fully preemptive, i.e., its preemption threshold is the same as its priority.

$$\forall \tau_i \in \tau_{\pi_i} \exists t \in T_i \exists rbf_j(t) \leq t$$

where $rbf_j(t)$ 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 [11]. 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 [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 $\pi_i$ of runnable $\rho_i$ is inherited from the priority of the task $\tau_i$ it is mapped to, i.e., $\pi_i = \Pi_i$. 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.

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

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

$$p_{i,j} + p_{j,i} = 1$$

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 $\rho_i, \rho_j$, $\rho_i$ cannot preempt $\rho_j$ iff $\pi_i \geq \pi_j$. A set of binary variables is used to encode this condition

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

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

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

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 $\rho_j$; conversely, if $\rho_i$ can preempt $\rho_j$, any runnable with priority $\pi_i \leq \pi_k$ can preempt $\rho_j$.

$$\forall k : \pi_k \geq \pi_i, \quad q_{k,j} \geq q_{i,j}$$

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 $o_{i,j}$ from the activation of $\rho_i$ to the following activation of $\rho_j$ allows to demonstrate that $\rho_j$ cannot preempt $\rho_i$.

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

If $o_{i,j} \geq D_{i,j}$, 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.

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

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 \((q_{j,i} = 1)\).

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

(9)

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

\[
\begin{align*}
    h_{i,j} &\leq z_{i,j} + q_{j,i} \\
    h_{i,j} &\geq z_{i,j}, \quad h_{i,j} \geq q_{j,i}
\end{align*}
\]

(10)

**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.

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

(11)

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

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

(12)

For each link between the writer \( \rho_i \in \mathcal{E}_k^W \) and the low priority reader \( \rho_j \in \mathcal{E}_k^{LR}, \) 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

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

(13)

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

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

(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

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

(15)

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

\[
w_k + l_k \leq 1
\]

(16)

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

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

(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 \( \rho_i \) and \( \rho_j \) mapped into different tasks, we define a variable \( g_{i,j} \) equal to 1 if \( \rho_i \) and \( \rho_j \) ar in the same non-preemption group, and 0 otherwise.

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

(18)

\( \rho_i \) and \( \rho_j \) 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.

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

(19)

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

\[
\begin{align*}
    g_{i,j} &= g_{j,i} \\
    g_{i,j} + g_{j,k} - 1 &\leq g_{i,k}
\end{align*}
\]

(20)

**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 \( \varepsilon_k \in \mathcal{E}_i \).

\[
c_i(\varepsilon_k) = C_i(\varepsilon_k) + H_3 \cdot l_k
\]

(21)

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

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

(22)

**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.

\[
B_i = \max_{j : \pi_j, c_j, \max_{\varepsilon_k \in \mathcal{E}_i} l_k \cdot c_i'(\varepsilon_k) + l_k \cdot H_3}
\]

(23)

Note that \( l_k \cdot l_k = l_k \), the second item \( l_k \cdot c_i'(\varepsilon_k) \) 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

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

(24)
The variables $q_{i,j,k}$, $q_{i,j}$ and $l_k$ should satisfy
\[ q_{i,j} + l_k - 1 \leq q_{i,j,k} \]
\[ q_{i,j,k} \leq q_{i,j} \quad q_{i,j,k} \leq l_k \] 
(25)

Thus (23) can be written in a set of MILP constraints as
\[
\forall j: \pi_i < \pi_j, \\
\begin{cases}
B_i \geq q_{i,j} \cdot C_{j,0} + q_{i,j} \sum_{c_k \in E_j} c_j(\varepsilon_k) + H_3 \sum_{c_k \in E_j} q_{i,j,k} \\
\forall \varepsilon_k \in E_j, B_i \geq l_k \cdot c_j(\varepsilon_k) + l_k \cdot H_3
\end{cases}
\] 
(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 $\varepsilon_k$ can be formulated as
\[
rbf_0(\varepsilon_k, t) = \sum_{\rho_i \in \rho(\varepsilon_k)} \left( u_k \cdot \left\lfloor \frac{t}{t_i} \right\rfloor \right) H_1 + \sum_{\rho_j \in \rho(\varepsilon_k)} f_{k,i,j} \left( \frac{t}{t_j} \right) H_2 + \sum_{\rho_j \in \rho(\varepsilon_k)} w_k \left( \frac{t}{t_j} \right) \cdot H_2
\] 
(27)

The total request bound function for all the shared resources is
\[
rbf_0(t) = \sum_{\varepsilon_k \in E} rbfo(\varepsilon_k, t)
\] 
(28)

**Real-time Schedulability** To verify the schedulability of $\rho_j$, we check whether there exists a point $t \in T_j$ 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. $rbfo_0(t)$: kernel-level timing overhead; 
3. $rbf_j(t)$: the computation time $c_j$ of $\rho_j$ (as $t \leq T_j$); 
4. $rbf_j(t)$, $\forall i$ with $\pi_i < \pi_j$: the sum of the interferences from blocks $\rho_i$ with higher priority, which is
\[
\sum_{i: \pi_i < \pi_j} \left\lfloor \frac{t}{t_i} \right\rfloor \cdot c_i
\] 
(29)

5. $rbf_j(t)$, $\forall i$ with $\pi_i = \pi_j$: the sum of the interferences from blocks $\rho_i$ mapped to the same task, which is
\[
\sum_{i: \pi_i = \pi_j} p_{i,j} \left\lfloor \frac{t}{t_i} \right\rfloor \cdot c_i
\] 
(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 $\varepsilon_k$ of $\rho_i$. We define the following two variables to make the constraint (30) linear:
\[
\forall \rho_j \neq \rho_i, \varepsilon_k \in E_i^{in} \cup E_i^{out}
\]
\[ v_{i,j,k} = \begin{cases} 1 & \text{if } p_{i,j} = 1 \text{ and } y_k = 1 \\ 0 & \text{otherwise} \end{cases} \] 
(31)

$v_{i,j,k}$ should satisfy the following constraints:
\[ p_{i,j} + y_k - 1 \leq v_{i,j,k} \]
\[ v_{i,j,k} \leq p_{i,j}, \quad v_{i,j,k} \leq y_k \] 
(32)

Similarly,
\[
\forall \rho_i \neq \rho_j, \varepsilon_k \in E_i^{in} \cup E_i^{out}
\]
\[ w_{i,j,k} = \begin{cases} 1 & \text{if } p_{i,j} = 1 \text{ and } l_k = 1 \\ 0 & \text{otherwise} \end{cases} \] 
(33)

$w_{i,j,k}$ should satisfy the following constraints:
\[ p_{i,j} + l_k - 1 \leq w_{i,j,k} \]
\[ w_{i,j,k} \leq p_{i,j}, \quad w_{i,j,k} \leq l_k \] 
(34)

**Stack usage** The stack usage of the system includes:

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

We order the runnables according to their decreasing usage of stack:
\[ \alpha: \rho_i \rightarrow \mathbb{N}^+ \] 
(35)

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

We define the following binary variable
\[ u_i = \begin{cases} 1 & \text{if } \rho_i \text{ has the largest stack size} \\ 0 & \text{otherwise} \end{cases} \] 
(36)

$u_i$ is dependent on $g_{i,j}$ and should satisfy
\[ 1 - \sum_{j: o(\rho_j) \leq o(\rho_i)} g_{i,j} \leq u_i \]
\[ u_i \leq 1 - g_{i,j}, \forall j: o(\rho_j) \leq o(\rho_i) \] 
(37)

The maximum stack usage is
\[ s = \sum_{i \in T} S_i + \sum_{\rho_i \in \rho} S_i \cdot u_i \] 
(38)

**Memory constraints** The memory cost of the additional wait free buffers for resource $\varepsilon_k$ is
\[ n_k = \sum_{\rho_i \in \rho(\varepsilon_k), \varepsilon_k \in E_i^{in} \cup E_i^{out}} f_{k,i,j} + 2w_k \quad \text{if } \rho_k^{HR} \neq \emptyset \]
\[ n_k = \sum_{\rho_i \in \rho(\varepsilon_k), \varepsilon_k \in E_i^{in} \cup E_i^{out}} f_{k,i,j} + w_k \quad \text{if } \rho_k^{HR} = \emptyset \] 
(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
\[ m = M_A + \sum_{\varepsilon_k \in E} n_k + s \] 
(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.

minimize $m$ 
(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.

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.

<table>
<thead>
<tr>
<th>Task</th>
<th>Period (ms)</th>
<th>Priority</th>
<th>C(_i) (\mu s)</th>
<th>NW</th>
<th>NLPR</th>
<th>NHPR</th>
<th>Stack (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_1</td>
<td>1000</td>
<td>0</td>
<td>1000</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>512</td>
</tr>
<tr>
<td>c_2</td>
<td>1000</td>
<td>1</td>
<td>5000</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>604</td>
</tr>
<tr>
<td>c_3</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>c_4</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>c_5</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>c_6</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>c_7</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>c_8</td>
<td>8</td>
<td>1</td>
<td>340</td>
<td>4</td>
<td>1</td>
<td>12</td>
<td>608</td>
</tr>
<tr>
<td>c_9</td>
<td>5</td>
<td>5</td>
<td>56</td>
<td>5</td>
<td>1</td>
<td>448</td>
<td></td>
</tr>
<tr>
<td>c_{10}</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>c_{11}</td>
<td>1000</td>
<td>14</td>
<td>110000</td>
<td>3</td>
<td>13</td>
<td>2</td>
<td>640</td>
</tr>
<tr>
<td>c_{12}</td>
<td>4</td>
<td>2</td>
<td>39</td>
<td>2</td>
<td>4</td>
<td>18</td>
<td>288</td>
</tr>
<tr>
<td>c_{13}</td>
<td>12</td>
<td>9</td>
<td>820</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>1024</td>
</tr>
<tr>
<td>c_{14}</td>
<td>50</td>
<td>8</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>c_{15}</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>c_{16}</td>
<td>1000</td>
<td>13</td>
<td>1100000</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 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