Stack memory requirements of AUTOSAR/OSEK-compliant scheduling policies

Technical Report

Bril, R.J.; Altmeyer, S.J.; Gai, P.

Citation for published version (APA):
Stack memory requirements of AUTOSAR/OSEK-compliant scheduling policies

Technical Report

Reinder J. Bril$^1$, Sebastian Altmeyer$^2$ and Paolo Gai$^3$

$^1$Technische Universiteit Eindhoven (TU/e), Eindhoven, The Netherlands
$^2$University of Amsterdam (UvA), Amsterdam, The Netherlands
$^3$Evidence Srl, Pisa, Italy

Abstract—Stack sharing between tasks may significantly reduce the amount of memory required in resource-constrained real-time embedded systems. Existing work on stack sharing mainly focused on stack sharing between tasks that neither leave any data on the stack from one instance to another nor suspend themselves, i.e. tasks with a so-called single-shot execution.

In this paper, we consider stack memory requirements of AUTOSAR/OSEK-compliant scheduling policies for a mixed task set, consisting of so-called basic and extended tasks. Unlike basic tasks, that have a single-shot execution, extended tasks are allowed to leave data on the stack from one instance to another and to suspend themselves. We prove that minimizing the shared stack requirement for such a mixed task set is an NP-hard problem. We subsequently provide an heuristic-based algorithm to minimize stack usage of a mixed task set, and evaluate the algorithm through a case study of an implementation of an unmanned aerial vehicle.

I. INTRODUCTION

Real-time embedded systems are typically resource-constrained. To reduce the amount of memory (RAM) for such systems, many real-time operating systems (RTOSs) provide means for stack sharing between tasks with a single-shot execution, such as Erika Enterprise [10] and Rubus [21]. The theoretical foundation for stack sharing between such tasks in systems with priority-based scheduling has been laid in [3, 7, 20, 4], amongst others. Stack sharing may give rise to a variation within the address space of tasks, however, which may prohibit the use of static timing verification and/or reduce the precision of execution bounds [23]. In [2], an efficient and effective method for predictable stack sharing (EMPRESS) was therefore presented, where the stack of a task is always located in the very same memory area. As described in [2], the results of EMPRESS can be realized within the Erika Enterprise RTOS without additional overheads.

These existing approaches for stack sharing exclusively considered tasks with a single-shot execution. Operating system standards, such as OSEK/VDX [19] and AUTOSAR-OS [1], support tasks with a single-shot execution, termed basic tasks, as well as tasks that may leave data on the stack from one instance to another or may suspend themselves, termed extended tasks, however. We are aware of only a single, patented approach that supports stack sharing of a mixed set of basic and extended tasks. That approach has been implemented in the RTA-OSK RTOS from ETAS [8]. Although the tasks execute on a single shared stack, additional memory buffers are used to temporarily store data of extended tasks upon suspension and completion of instances. In this paper, we aim at minimizing the stack memory requirements of a mixed set of basic and extended tasks without the need for additional buffers and for copying data back and forth between buffers and the stack, whilst providing predictability of stack sharing, as provided by EMPRESS. We focus on AUTOSAR/OSEK-compliant scheduling policies in general, and fixed-priority scheduling with preemption thresholds (FPTS) [22] in particular.

This paper presents four major contributions. Firstly, existing work on stack sharing of basics tasks is revisited and several novel insights are presented. As examples, it is shown that (i) an optimal set of non-preemptive groups [11] does not necessarily yield the minimal stack requirement and (ii) reducing stack requirements through possible preemption paths [4] is at the cost of a loss of predictability of a system. Secondly, a proof is presented that minimizing the shared stack requirement of a mixed task set is an NP-hard problem. Thirdly, an heuristic-based algorithm is presented, termed EMPRESS$^{Ext}$, aiming at a minimal stack requirement for a mixed task set with predictable stack sharing. Finally, an evaluation of EMPRESS$^{Ext}$ is presented using a case study.

The paper is structured as follows. In Section II, we introduce our system model and the required technical background. Section III revisits the related work on stack sharing of basic tasks. Section IV subsequently introduces the implications of having extended tasks next to basic tasks on stack sharing. In Section V, we prove that minimizing the stack requirements of a set of basic and extended tasks in an NP-hard problem. Thirdly, an heuristic-based algorithm is presented, termed EMPRESS$^{Ext}$, aiming at a minimal stack requirement for a mixed task set with predictable stack sharing. Finally, an evaluation of EMPRESS$^{Ext}$ is presented using a case study. The paper is concluded in Section IX.

II. BACKGROUND

In this section, we introduce our system model and the required technical background.

a) Scheduling Model: We assume a single-processor system, a set $T$ of $n$ tasks $\tau_1, \tau_2, \ldots, \tau_n$, and fixed-priority scheduling with preemption thresholds (FPTS). Every task $\tau_i$ has a priority $\pi_i$ and a preemption threshold $\theta_i$, where higher values for priorities represent higher priorities and $\theta_i \geq \pi_i$. Under FPTS, a task $\tau_i$ is only allowed to preempt a task $\tau_j$ when $\pi_i > \theta_j$. Tasks with the same priority are executed in first-in-first-out (FIFO) order, and when they arrive simultaneously they
are executed based on their index, lowest index first. We use \( \Pi(T) \) and \( \Theta(T) \) to denote the set of priorities and preemption thresholds associated with the tasks in \( T \), respectively, i.e.

\[
\Pi(T) = \{ \pi \mid \exists \tau_i \in T, \pi_i = \pi \},
\]

\[
\Theta(T) = \{ \theta \mid \exists \theta_i = \theta \}.
\]

Tasks may share mutually exclusive resources using an early blocking resource access protocol, such as the stack resource policy (SRP) [3]. The set of tasks \( T \) is partitioned in two sets, a set \( T^B \) of basic tasks and a set \( T^E \) of extended tasks. Basic tasks are not allowed to either suspend themselves or leave any data on the stack from one instance of the task to the next, whereas extended tasks are allowed to do both. Each task is characterized by a worst-case execution demand \( C \), a period (or minimal inter-arrival time) \( T \) and a relative deadline \( D \).

**b) System Model:** The system does not support memory address translation (as common within memory-management units or virtual memory) and facilitates a direct addressing from cache to main memory. Such a mapping is common amongst many embedded architectures and embedded operating systems [16] and often preferable over virtual memory for performance reasons.

We assume that the stacks of all tasks are mapped to the same memory space, starting at a system-wide static stack pointer. Without loss of generality, we set the memory address of this system-wide static stack pointer to 0, and only provide stack addresses relative to this static stack pointer.

**c) Maximal Stack Usage:** As stack overflows are a common source of system failures, techniques exist to upper-bound the stack-usage [6, 15] and hence to prevent stack overflows. These techniques are in particular important for hard real-time systems, where correctness is a primary concern and has to be validated statically [17].

Using these techniques, we can derive for each task \( \tau_i \) its maximum stack usage \( SU_i \in \mathbb{N}^\theta \). For the sake of simplicity, we assume that \( SU_i \) provides the maximum stack usage of task \( \tau_i \) including the size of the stack frame. The stack memory needed by any two pre-empting tasks \( \tau_i \) and \( \tau_j \) is therefore bounded by \( SU_i + SU_j \).

For an extended task \( T^E_i \), part of its stack is potentially shared and thus called shared stack. The part of the stack which is not shared, but remains on the stack in between jobs and upon suspension, is called dedicated stack. We assume the worst-case size of the dedicated stack \( SU^D_i \) can be determined, or at least safely bounded. The size of the shared stack \( SU^S_i \) can subsequently be derived using \( SU^S_i = SU_i - SU^D_i \).

**d) Pre-emption Relation and Pre-emption Graph:** We assume a binary pre-emption relation \( \preceq \) of allowed preemptions on tasks [4], which is derived from the priority levels and/or pre-emption levels of the tasks. In particular, we ignore the fact that extended tasks may suspend themselves. The relation \( \tau_j \preceq \tau_i \) holds if and only if task \( \tau_j \) can be pre-empted by task \( \tau_i \). For common real-time scheduling policies, such as fixed-priority pre-emptive scheduling (FPPS), fixed-priority non-pre-emptive scheduling (FPNS), fixed-priority threshold scheduling (FPTS), and earliest deadline first (EDF), such a relation is a strict partial order (SPO), i.e. both irreflexive (\( \lnot \tau \preceq \tau \)) and transitive (\( \tau_k \preceq \tau_j \land \tau_j \preceq \tau_i \Rightarrow \tau_k \preceq \tau_i \)). Given \( \preceq \), we can derive a directed acyclic graph (DAG) of allowed preemptions on tasks, where the nodes represent the tasks and an edge from a task \( \tau_i \) to \( \tau_j \) represents that \( \tau_j \preceq \tau_i \) holds.

Without loss of generality, we assume \( \tau_j \preceq \tau_i \Rightarrow j > i \), i.e. when task \( \tau_j \) can be pre-empted by task \( \tau_i \), \( \tau_j \) has a higher index than \( \tau_i \).

**e) Bounding a task’s shared, dedicated and total stack usage:** In this paper, we assume that we are provided with bounds on the stack usage of each task. We even consider the question on the derivation of stack bounds as out-of-scope of the paper and as largely solved. AbsInt, for instance, provides a static stack analyzer [13] able to derive safe bounds on a task’s stack usage. Concerning the separation of dedicated and shared stack requirements, however, the static stack analyzer provides, to the best of our knowledge, no built-in feature. It simply computes a stack bound starting from a user-defined program point, typically the main-function of the task. To derive bounds on the shared and/or dedicated stack usage of a task, manual annotations are needed to correctly configure the analysis. This includes, amongst others, the appropriate selection of starting points or the classification of function calls to either contribute only to the shared or to the dedicated stack usage. A measurement-based stack analysis represents one of course an alternative solution. The task is simply executed with varying inputs and the maximum dedicated and shared stacks needs are recorded. A safety margin is often added to the stack bound to increase the reliability. In both cases, there is no fundamental problem in deriving the dedicated and shared stack usage. Nevertheless, it is important to err on the safe side. In this setting, it means that we should rather over-approximate the portion of the task’s total stack which we consider dedicated, and under-approximate the portion which can potentially be shared with other tasks – of course under the assumption that the total stack need is a safe upper bound.

### III. Related work on basic task sets revisited

Focus of existing work is on stack sharing of basic tasks. In this section, we revisit existing approaches to determine the maximum stack usage \( SU(T^B) \) of a set of basic tasks \( T^B \). We consider approaches based on partitioning and on possible preemption chains. Next, we briefly summarize work on predictable stack sharing. We conclude with a summary of our findings.

#### A. Approaches based on partitioning

In [3, 7, 20, 11], approaches are described to determine the maximum stack usage of a set \( T^B \) of basic tasks based on partitioning. A partitioning \( G \) of a set of tasks \( T^B \) is a set of \( m \) subsets \( G_1, \ldots, G_m \) of \( T^B \), where the union of the subsets is equal to \( T^B \), i.e. \( \bigcup_{1 \leq i \leq m} G_i = T^B \), and the intersection of any two subsets is the empty set, i.e. \( \forall 1 \leq i < j \leq m \) \( G_i \cap G_j = \emptyset \). For each of the approaches, the elements in a subset \( G_k \) are pairwise non-pre-emptive, i.e.

\[
\forall 1 \leq i \neq j \leq m \quad \lnot \tau_i \preceq \tau_j \land \lnot \tau_j \preceq \tau_i.
\]
These approaches allocate a dedicated stack area to each subset of \( G \). The stack usage \( SU(T^B) \) of the set \( T^B \) of basic tasks is therefore given by the sum of the maximum stack usage of the tasks of each subset \( G_\tau \), i.e.

\[
SU(T^B) = \sum_{G_\tau \in \Gamma} \max_{\tau \in G} SU_\tau. \tag{1}
\]

The criteria used for partitioning differ per approach, however.

\( a) \) FPPS and SRP [3]: The original motivation for the development of SRP (Stack Resource Policy) [3], and in particular for the choice of early blocking, was to support stack sharing between tasks. For FPPS and SRP, tasks with the same priority are mutually non-preemptive. Partitioning of \( T^B \) in [3] is therefore based on task priorities, i.e.

\[
G(\pi, T^B) = \{ \tau_i \in T^B \mid \pi_i = \pi \},
\]

\[
\mathcal{G}(T^B) = \{ G(\pi, T^B) \mid \pi \in \Pi(T^B) \}.
\]

\( b) \) Non-preemption groups [7]: In [7], the notion dispatch priority is introduced, which is essentially the same as the notion preemption threshold for FPTS [22]. Partitioning of \( T^B \) is based on preemption thresholds in [7], i.e.

\[
G(\theta, T^B) = \{ \tau_i \in T^B \mid \theta_i = \theta \},
\]

\[
\mathcal{G}(T^B) = \{ G(\theta, T^B) \mid \theta \in \Theta(T^B) \}.
\]

To illustrate the result of this approach we take an example from [9]. The priorities, pre-emption thresholds, and stack usage of this task set \( T_1 \) are given in Table I. A directed acyclic graph (DAG) of allowed pre-emptions and the stack usage of the tasks of \( T_1 \) are illustrated in Figure 1. The number of non-preemption groups \( |\mathcal{G}(T_1)| \) for this partitioning of \( T_1 \) is equal to the number of distinct preemption thresholds of \( T_1 \), i.e. \( |\mathcal{G}(T_1)| = |\Theta(T_1)| = 6 \). Using (1), the resulting stack usage of \( T_1 \) based on non-preemption groups becomes 204, whereas it would have been 206 when stack usage is based on priorities.

\( c) \) FPTS and non-preemptive groups [20]: Partitioning in [20] is based on so-called non-preemptive groups, where a non-preemptive group is a set of tasks that are pairwise mutually non-preemptive based on priorities and preemption thresholds; see next proposition.

Proposition ([20]). Two tasks \( \tau_i \) and \( \tau_j \) are mutually non-preemptive if \( (\pi_i \leq \theta_j) \land (\pi_j \leq \theta_i) \).

There are many ways a set of tasks can be partitioned in non-preemptive groups, and we observe that non-preemption groups in [7] are just a specific partitioning in terms of non-preemptive groups [20].

The following claim is made in [20]

"By minimizing the number of non-preemptive groups we minimize the stack space requirement."

and an algorithm OPT-Partition is presented that partitions a set of tasks in a minimum number of non-preemptive groups.

The minimal number of non-preemptive groups of tasks of \( T_1 \) using this algorithm is 3, which is smaller than the number 6 of non-preemption groups as found using [7]; see also Figure 2. Using (1), the resulting stack usage of \( T_1 \) becomes 201, which is indeed smaller than a stack usage of 204 found using [7].

\( d) \) Optimal set of non-preemptive groups [11]: In [11], it is observed that by minimizing the number of non-preemptive groups, the stack usage, as determined by (1) is not necessarily minimized. An algorithm is therefore presented in [11] to find a so-called optimal set of non-preemptive groups that minimizes the stack usage.

An optimal set of 4 non-preemptive groups of tasks of \( T_1 \) is illustrated in Figure 3. From this figure and using (1), a stack usage of \( T_1 \) of 103 is found, refuting the claim in [20].
e) Non-optimality of partitioning: In [5, 14], it is observed that the maximum stack usage can be found by determining the longest path in the DAG of allowed preemptions with stack usage as weight. Applying this approach to $T_1$ yields a value of 102, i.e. a value that is smaller than the value of 103 found for an optimal set of non-preemptive groups.

From our example, we therefore conclude that (i) an optimal set of non-preemptive groups of $T$ does not necessarily minimize the stack usage $SU(T)$ of $T$, and (ii) $SU(T)$ is independent of the partitioning of $T$ in non-preemptive groups.

To better understand why this is the case, consider the set $T_{II} = \{\tau_3, \tau_5, \tau_7, \tau_8\} \subseteq T_1$; see also Figure 4. For this set $T_{II}$, we can construct 5 partitions into non-preemptive groups; see Table II. Considering Figure 5, the minimal stack usage of $T_{II}$ is 101, i.e. there is no partitioning of $T_{II}$ in non-preemptive groups yielding this stack usage. This is an immediate consequence of the fact that existing approaches do not assume (or allow) stack sharing between non-preemptive groups, whereas the minimal stack usage can only be found through such stack sharing.

![Fig. 4. A DAG of allowed pre-emptions and stack memory requirements of the tasks of subset $T_{II}$ of $T_1$.](image)

**TABLE II**

<table>
<thead>
<tr>
<th>partition $G_i$</th>
<th>approach $SU(T_{II})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_1$ ${\tau_3, \tau_5, \tau_7, \tau_8}$</td>
<td>[20] 200</td>
</tr>
<tr>
<td>$G_2$ ${\tau_3, \tau_7, \tau_5, \tau_8}$</td>
<td>[11] 102</td>
</tr>
<tr>
<td>$G_3$ ${\tau_3, \tau_7, \tau_5, \tau_8}$</td>
<td>[7] 202</td>
</tr>
<tr>
<td>$G_4$ ${\tau_3, \tau_5, \tau_7, \tau_8}$</td>
<td>201</td>
</tr>
<tr>
<td>$G_5$ ${\tau_3, \tau_5, \tau_7, \tau_8}$</td>
<td>201</td>
</tr>
</tbody>
</table>

![Fig. 5. A stack layout for $T_{II}$ shown in Figure 4.](image)

**B. Approach based on possible preemption chains [4]**

In [4], response time analysis and release jitter analysis is applied to prune the DAG of allowed preemptions of a set of tasks with offsets and precedences, resulting in a DAG of possible preemptions, and allowing a reduction of the minimal stack usage of that set. We will illustrate their approach based on an example presented in [4]; see Figure 6. Note that we have taken the notation from [4], where the first digit of the index of a task reflects the transaction to which the task belongs and the second digit the index of the task within the transaction. Hence, the index does not correspond to the task’s priority. Observe that the precedence relation $\prec$ of possible preemptions is no longer transitive, e.g. $\tau_{13} \prec \tau_{23}$ and $\tau_{23} \prec \tau_{11}$ but $\neg (\tau_{13} \prec \tau_{11})$.

A so-called possible preemption chain (PPC) is a sequence of tasks if and only if the relation $\prec$ holds transitively between all tasks in the sequence. A maximal stack usage PPC is a PPC for which no other PPC has a higher stack usage. An algorithm is presented in [4] to determine a maximal stack usage PPC. In Table III, 5 PPCs of Figure 6 are given with their stack usage, where PPC $\{\tau_{11}, \tau_{12}, \tau_{23}\}$ is a maximal PPC. The stack usage of the task set is therefore 8, which is less than 10, which would be found by determining the longest path in the DAG of possible preemptions with stack usage as weight. Stack layouts for the PPCs are shown in Figure 7.

![Fig. 6. Directed acyclic graph (DAG) of possible preemptions and stack usage of an example in [4].](image)

**TABLE III**

<table>
<thead>
<tr>
<th>PPC</th>
<th>$SU$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${\tau_{23}, \tau_{12}, \tau_{11}}$</td>
<td>$3 + 1 + 4 = 8$</td>
</tr>
<tr>
<td>${\tau_{21}, \tau_{12}, \tau_{11}}$</td>
<td>$2 + 1 + 4 = 7$</td>
</tr>
<tr>
<td>${\tau_{13}, \tau_{21}}$</td>
<td>$2 + 2 = 4$</td>
</tr>
<tr>
<td>${\tau_{23}, \tau_{12}, \tau_{22}}$</td>
<td>$1 + 1 + 3 = 5$</td>
</tr>
<tr>
<td>${\tau_{13}, \tau_{23}, \tau_{22}}$</td>
<td>$1 + 3 + 2 = 6$</td>
</tr>
</tbody>
</table>

![Fig. 7. Stack layouts of the PPCs of an example in [4].](image)
C. Predictable stack sharing [2]

In [2], an Efficient and effective Method for Predictable Stack Sharing (EMGRESS) is presented, i.e. the stack of every task is always located in the very same memory location, even for tasks sharing a stack. The method combines the predictability of dedicated stack spaces with the reduced memory needs of a shared stack. Algorithm 1, to determine the stack address of a task, is based on the same principle as an algorithm to determine the maximum stack usage of a set of tasks [5, 14].

We will refer to Algorithm 1 as EMPRESS. The algorithm starts with the task with the highest index, i.e. a task that can be pre-empted by other tasks but cannot pre-empt any task. The maximum stack address of task \( \tau_i \) is given by the maximum sum of the stack address \( \text{SA}_j \) and the stack usage \( \text{SU}_j \), where \( \tau_j \) is potentially pre-empted by task \( \tau_i \). The derived stack address of each task is relative to the system-wide static stack pointer, which is set to memory address 0. \( \text{SA}_i \) therefore does not provide an absolute address.

Based on the stack addresses \( \text{SA}_i \) for each task \( \tau_i \in \mathcal{T} \) determined by EMPRESS, the stack usage \( \text{SU}^{\text{EMPRESS}}(\mathcal{T}) \) of \( \mathcal{T} \) can be derived by

\[
\text{SU}^{\text{EMPRESS}}(\mathcal{T}) = \max_{\tau_i \in \mathcal{T}} (\text{SA}_i + \text{SU}_i). \tag{2}
\]

As illustrated in [4], \( \prec \) need not be transitive. Assuming the preemption graph is a directed acyclic graph, EMPRESS equally well applies for a relation \( \prec \) representing the possible preemptions between tasks.

Figure 8 shows the stack layout of \( \mathcal{T}_1 \) using EMPRESS, i.e. tasks use fixed ranges of memory locations for their stack.

Applying EMPRESS to the example in Section III-B yields a stack layout as illustrated in Figure 9. From this layout, we conclude that the reduction of the stack usage in [4] is at the cost of a loss of predictability of the system.

D. A summary of findings

Below, we summarize our findings:

1. The non-preemption groups in [7] are a special case of partitioning in non-preemptive groups in [20].
2. As mentioned in [11], the stack usage of a partitioning in non-preemptive groups is not necessarily minimized for a minimal number of non-preemptive groups, refuting a claim in [20].
4. EMPRESS will equally well work when the preemption relation \( \prec \) is not transitive.
5. Although the approach of possible preemption chains [4] may reduce the stack usage, that reduction is at the cost of a loss of predictability of a system.

To the best of our knowledge, only finding 2) was explicitly reported upon in the literature before.

IV. A mixed set of basic and extended tasks

In this section, we explore the consequences of a mixed task set for stack sharing. We start by considering the consequence for the binary relation \( \prec \) in Subsection IV-A and subsequently present observations on stack sharing and bounds on stack usage in Subsection IV-B.

A. Binary relation \( \prec \) revisited

In Section II, we assumed a binary relation \( \prec^a \) on tasks, which is derived from the priority levels and/or pre-emption levels of the tasks. As shown in Section III-B, the DAG of
allowed preemptions can be pruned to a DAG of possible preemptions for a set of basic tasks with offsets and precedences. Viewed as a set of pairs, the set of possible preemptions \( \prec \) is therefore a subset of \( \prec_a \), i.e. \( \prec \subseteq \prec_a \).

We now consider the influence of a mixed task set on stack sharing in general and how to reflect that influence on \( \prec \). Consider a set \( T_{III} \) of two independent tasks, a basic task \( \tau^B \) and an extended task \( \tau^E \), with characteristics as given in table IV, which are scheduled using FPTS.

**TABLE IV**

<table>
<thead>
<tr>
<th>task</th>
<th>( \pi )</th>
<th>( \theta )</th>
<th>( C )</th>
<th>( T = D )</th>
<th>( SU^B )</th>
<th>( SU^E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau^B )</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( \tau^E )</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Based on their priorities and preemption thresholds, tasks \( \tau^B \) and \( \tau^E \) are mutually non-preemptive, i.e. \( \neg \tau^B \prec \tau^E \wedge \neg \tau^E \prec \tau^B \). As illustrated in Figure 10(a), however, \( \tau^B \) may execute when a job of \( \tau^E \) suspends itself, which looks similar to a preemption of \( \tau^E \) by \( \tau^B \). Because \( \tau^E \) and \( \tau^B \) are mutually non-preemptive based on priorities and preemption thresholds, task \( \tau^E \) may still share its shared stack area with \( \tau^B \), as illustrated in Figure 10(b).

![A timeline for \( \tau^B \) and \( \tau^E \).](image)

![A stack layout for \( \tau^B \) and \( \tau^E \).](image)

As a result, we ignore the type of tasks when determining the preemption relation \( \prec \) between tasks for stack sharing and therefore have to revert to another, complementary approach to address stack sharing between basic and extended tasks.

**B. Observations on stack sharing for a mixed task set**

Regarding stack sharing of a mixed task set, we make four observations. We start with an observation for basic tasks [3, 7, 20, 4]; see also Section III.

**Observation 1.** Basic tasks that are mutually non-preemptive can share a stack.

Next we summarize the consequence of a dedicated stack area for an extended task:

**Observation 2.** The dedicated stack of an extended task cannot be shared with any other task, i.e. neither with another extended task nor with a basic task.

As a result, an extended task cannot share a stack with another extended task, i.e.

**Observation 3.** An extended task cannot share a stack with another extended task, irrespective whether or not they are mutually non-preemptive.

As illustrated in the previous subsection, the shared stack area of an extended stack can be shared with a basic task, as long as the tasks are mutually non-preemptive:

**Observation 4.** Whenever an extended task and a basic task are mutually non-preemptive, the shared stack area of the extended task can be shared with the stack of the basic task.

Based on these observations, we can derive both lower bounds as well as an upper bounds on the stack usage of a mixed task set \( T \).

A first lower bound \( SU_{min}(T) \) can be found through a simple summation of the stack usage of extended tasks by assuming that all basic tasks can share their stack with the shared stacks of the extended tasks such that no additional stack space is required, i.e.

\[
SU_{min}(T) = \sum_{\tau \in T^E} SU_{\tau}.
\]

Another lower bound \( SU_{lwb}(T) \) is found by ignoring the restrictions imposed on stack sharing by the extended tasks of \( T \), applying EMPRESS (Algorithm 1), and using Equation (2), i.e.

\[
SU_{lwb}(T) = SU_{EMPRESS}(T).
\]

We merely observe that \( SU_{min}(T) \) and \( SU_{lwb}(T) \) are incomparable.

We can find an upper bound \( SU_{upb}(T) \) by assuming that none of the basic tasks can share their stack with extended tasks, i.e. by combining Equations (3) and (2):

\[
SU_{upb}(T) = SU_{min}(T) + SU_{EMPRESS}(T^B).
\]

Finally another upper bound \( SU_{max}(T) \) can be found through a simple summation of the stack usage of all tasks by assuming no sharing of stacks, i.e.

\[
SU_{max}(T) = \sum_{\tau \in T} SU_{\tau}.
\]

We observe that \( SU_{upb}(T) \leq SU_{max}(T) \).

From these four bounds, \( SU_{lwb}(T) \) and \( SU_{max}(T) \) are independent of the number of extended tasks in \( T \) and the partitioning of their stacks in a shared and dedicated stack.

**V. Complexity Analysis**

In this section, we prove the following theorem.

**Theorem 1.** The minimization problem \( \mathcal{P} \) of finding a minimum stack layout for a mixed set \( T \) of basic and extended tasks scheduled by FPTS is NP-hard.

To this end, we provide an NP-completeness proof by restriction [12] for our problem \( \mathcal{P} \) by showing that \( \mathcal{P} \) contains a known NP-complete problem \( \mathcal{P}' \) as a special case. In our case, the known NP-complete problem \( \mathcal{P}'' \) is the bin-packing decision problem. The heart of our proof is in the additional restrictions to be placed on the instances of \( \mathcal{P} \) so that the resulting restricted problem will be identical to the bin-packing decision problem, i.e. that there exists an “obvious” one-to-one correspondence between their instances that preserves “yes” and “no” answers.
A. Bin-packing decision problem

Given a set \( \mathcal{B} \) of \( n \) bins \( \{B_1, \ldots, B_n\} \), each of size \( S_B \), and a set \( \mathcal{I} \) of \( m \) items \( \{a_1, \ldots, a_m\} \) with variable sizes \( S'_a \). Does there exist a mapping \( M \) of the \( m \) items to the \( n \) bins:

\[
M : \mathcal{I} \rightarrow \mathcal{B}
\]

such that:

\[
\forall B \in \mathcal{B} : \sum_{a_0 \in I, M(a_0) = B} S'_a \leq S_B?
\]

In other words, we have to find a mapping which distributes the \( m \) items to the \( n \) bins such that no bin overflows.

B. Restricting the minimization problem \( \mathcal{P} \) to a decision problem

Consider a set \( \mathcal{T} \) of \( n + m \) tasks \( \tau_1, \tau_2, \ldots, \tau_{m+n} \), that are scheduled by FPTS and can be partitioned in a set \( \mathcal{T}^E \) of \( n \) extended tasks \( \tau_{m+1}, \tau_{m+2}, \ldots, \tau_{m+n} \) and a set \( \mathcal{T}^B \) of \( m \) basic tasks \( \tau_1, \tau_2, \ldots, \tau_m \). Let each extended task \( \tau_i^E \in \mathcal{T}^E \) have a dedicated stack with a (fixed) stack usage \( SU^D_i \) and a shared stack with a (fixed) stack usage \( SU^S \). Moreover, let each basic task \( \tau_i^B \in \mathcal{T}^B \) have a (variable) stack usage \( SU_i \). Finally, let

- the basic tasks have unique priorities, the highest \( m \) priorities, and preemption thresholds equal to priorities;
- the extended tasks have the same, lowest priority, and preemption thresholds equal to the highest priority level.

Hence, each extended task is mutually non-preemptive with all other \( n - 1 \) extended tasks and with each of the \( m \) basic tasks, while there is no pair of basic tasks for which the task are mutually non-preemptive.

For such a set-up, none of the basic tasks can share a stack; see Observation 1. Moreover, none of the \( m \) extended tasks can share a stack; see Observation 3. Finally, each extended task can share the memory locations of its shared stack with the stack of every basic task; see Observation 4.

From Equation (3), we know that the sum of the stack usages of the extended tasks \( n \times SU^S + n \) is a lower bound on the total stack usage of the task set \( \mathcal{T} \). Whenever all the basic tasks can share their stack with the shared stacks of the extended tasks such that no additional stack space is required, this lower bound is assumed.

We now formulate our minimum stack layout decision problem as follows:

Given our set of mixed tasks \( \mathcal{T} \), does there exist a stack layout such that the stack usage of \( \mathcal{T} \) is equal to \( n \times SU^S + n \)?

In other words, does there exist a mapping \( M' \) of the stacks of the \( m \) basic tasks onto the \( n \) shared stacks of the extended tasks:

\[
M' : \mathcal{T}^B \rightarrow \mathcal{T}^E
\]

such that:

\[
\forall \tau^E \in \mathcal{T}^E : \sum_{\forall \tau_i^B \in \mathcal{T}^B : M(\tau_i^B) = \tau^E} SU_i \leq SU^S?
\]

**Lemma 1.** The minimum stack layout decision problem is NP-complete.

This lemma will be proven in the next subsection.

C. One-to-one correspondence between decision problems

The one-to-one correspondence between the bin-packing decision problem and the minimum stack layout decision problem is as follows:

- the set \( \mathcal{B} \) of \( n \) bins corresponds to the set \( \mathcal{T}^E \) of \( n \) extended tasks, where the size \( S_B \) of each bin is equal to the fixed shared stack usage \( SU^S \) of each extended task;
- the set \( \mathcal{I} \) of \( m \) items corresponds to the set \( \mathcal{T}^B \) of basic tasks, where the variable size \( S'_a \) of item \( a_i \) is equal to the stack usage \( SU_i \) of basic task \( \tau_i \).

With this correspondence between sets and their properties, the mappings \( M \) and \( M' \) correspond one-to-one, and the instances of the decisions problems preserve “yes” and “no” answers.

Hence, we conclude that the minimum stack layout decision problem is an NP-complete problem, which completes our proof of Lemma 1.

The relation between minimum stack layout and the bin-packing decision problem is exemplified in Figure 11.

\[
\begin{align*}
\text{(a) Feasible} & \quad \text{(b) Infeasible} \\
0 & \quad 0 \\
1 & \quad 1 \\
2 & \quad 2 \\
(n-1) \times S_B + (n-1) & \quad (n-1) \times S_B + (n-1) \\
(n) \times S_B + n & \quad (n) \times S_B + n \\
2 \times S_B + 2 & \quad 2 \times S_B + 2 \\
1 \times S_B + 1 & \quad 1 \times S_B + 1 \\
\end{align*}
\]

Fig. 11. Relation between minimum stack layout and bin-packing decision problem. Figure (a) shows a feasible bin-packing problem where the stack of every basic task (or every item) can be mapped onto a shared stack of an extended task (or on a bin). Figure (b) shows an infeasible bin-packing problem, where the minimum stack layout exceeds the lower bound of \( n \times S_B + n \). In the figure, the stack of the basic task \( \tau_1 \) (or item \( a_1 \)) cannot be mapped onto the shared stack of extended task \( \tau_{m+1} \) (or on bin \( B_i \)).

D. The minimization problem \( \mathcal{P} \) is NP-hard

Given that the restriction of the minimization problem \( \mathcal{P} \) to the minimum stack layout decision problem is an NP-complete problem, we conclude that the minimization problem is an NP-hard problem, which concludes our proof of Theorem 1.
VI. Minimizing the stack usage of a mixed task set

From Section V, we know that unless \( P = NP \), no polynomial-time algorithm can exist to compute a stack layout with minimal stack usage. We therefore resort to an heuristic, in which we aim to maximize the overlap between the shared stacks of extended tasks with stacks of basic tasks. The rationale behind this heuristic is as follows: dedicated stacks of extended tasks can never be shared with any other task; see Observation 2. We thus have a lower bound of the minimal stack usage given by the sum of the stack usages of the extended tasks; see Equation (3). If we can map the stacks of all basic tasks onto the shared stacks of extended tasks, then we have achieved the overlap, which in turn reduces the total stack requirement.

Equation (3). If we can map the stacks of all basic tasks onto the shared stacks of extended tasks, then we have achieved the minimal stack usage of the task set. If such a mapping of the stacks is not fully possible, then we at least try to maximize the overlap, which in turn reduces the total stack requirement.

Algorithm 2: TaskStackAddress-Ext(\( \mathcal{T} \), \(<\), \( SU.i \))

Input: A set of tasks \( \mathcal{T} \), a pre-emption relation \(<\), and for each task \( \tau_i \in \mathcal{T} \) the max. stack usage \( SU.i \).

Output: The static stack address \( SA_i \in \mathbb{N} \) for each task \( \tau_i \).

1: \( T^{ws} \leftarrow \mathcal{T} \setminus \{ \tau_i | \tau_i \in \mathcal{T} \) is an extended task\};
2: \( SA^e \leftarrow 0; \)
3: for each \( \tau_i^E \) (from highest to lowest index) do
4: \( T^c \leftarrow \{ \tau_i | \tau_i \in T^{ws} \land \tau_i^E \not\prec \tau_i \land \tau_i \not\prec \tau_i^E \}; \)
5: \( SA_i = SA^e; \)
6: \( SA^e = SA_i + SU_i; \)
7: for each \( \tau_i \in T^c \) (from highest to lowest index) do
8: \( SA_i \leftarrow SA_i + SU_i; \)
9: for each \( \tau_j \in T^c \) with \( j > i \land \tau_j \not\in T^{ws} \) do
10: if \( \tau_j \prec \tau_i \) then
11: \( SA_i \leftarrow \max(SA_i, SA_j + SU_j); \)
12: end if
13: end for
14: if \( SA_i < SA_i + SU_i \) then
15: \( T^{ws} \leftarrow T^{ws} \setminus \{ \tau_i \}; \)
16: \( SA^e \leftarrow \max(SA_i + SU_i, SA^e); \)
17: end if
18: end for
19: for each \( \tau_i \in T^{ws} \) (from highest to lowest index) do
20: \( SA_i \leftarrow SA^e; \)
21: for each \( \tau_j \in T \) with \( j > i \land j \not\in T^{ws} \) do
22: \( SA_i \leftarrow \max(SA_i, SA_j + SU_j); \)
23: if \( \tau_j \prec \tau_i \) then
24: \( SA_i \leftarrow \max(SA_i, SA_j + SU_j); \)
25: end if
26: end if
27: end for
28: end for

Algorithm 2 uses two global variables, a global stack address \( SA^e \) which is initialized with 0, and a working-set \( T^{ws} \) of tasks, initially containing all basic tasks.

The first outer loop (line 3 to 19) iterates over all extended tasks. For each extended task \( \tau_i^E \), we set the stack address of \( \tau_i^E \) to the global stack address \( SA^e \). Next, we construct a set \( T^c \) by selecting all basic tasks from the working-set \( T^{ws} \) that are mutually non-preemptive with \( \tau_i^E \) and hence can share the stack with the shared stack \( SU_i^c \) of \( \tau_i^E \); see Observation 4. For each such mutually non-preemptive basic task \( \tau_i \), we initially set the stack address to the stack address of \( \tau_i^E \) incremented by the dedicated stack size \( SU_i^c \) of \( \tau_i^E \). The inner-most loop (line 9 to 13) iterates over all tasks in \( T^c \) that may be preempted by \( \tau_i \) and updates the stack address \( SA_i \) to avoid illegal stack sharing with a potentially preempted task. The if-statement in line 14 now checks if the basic task \( \tau_j \) shares parts of its stack with the extended task \( \tau_i^E \). If so, then the basic task is removed from the working-set and the global stack address \( SA^e \) is updated.

Finally, the second outer loop (line 20 to 28) selects the stack addresses of all remaining basic tasks that have not yet been assigned a stack address. Line 23 prevents any overlap between the stack of tasks with run-to-completion semantics and the basic stack of tasks without. We note that this loop and the first inner loop (line 7 to 18) both perform the stack address computation from EMPRESS [2], just for different task sets. We will therefor refer to Algorithm 2 as EMPRESS\textsuperscript{Ext}.

We merely observe that EMPRESS\textsuperscript{Ext} is a “first-fit”-like algorithm, where a basic task shares (parts of) its stack with an extended task with the highest index that still has parts of its shared stack to share and with which it is mutually non-preemptive.

The stack usage using EMPRESS\textsuperscript{Ext} is given by

\[
SU^{EMPRESS\textsuperscript{Ext}}(\mathcal{T}) = \max_{\tau_i \in \mathcal{T}} (SA_i + SU_i). \tag{7}
\]

VII. Evaluation

In this section, we exemplify and evaluate EMPRESS\textsuperscript{Ext} using the case study from [2]. The case study is based on PapaBench [18] a free real-time benchmark implementing the control software of an unmanned aerial vehicle (UAV). It contains two disjoint task sets, \textit{Fly-by-wire} and \textit{Autopilot}. The pre-emption graphs for both task sets are shown in Figure 12. We have analyzed stack usage of PapaBench with AbsInt’s

![Fig. 12. Pre-emption constraints for PapaBench based on the precedence constraints, task frequencies and task priorities.](image-url)
that tasks T1, T3, T6, T11, and T12 are extended tasks. The results are shown in Figure 13 for Fly-by-wire, and Figure 14 for Autopilot. Despite the dedicated stack areas

![Stack layout and relative stack addresses for Predictable Stack Sharing: Fly-By-Wire Case Study.](image1)

Fig. 13. Stack layout and relative stack addresses for Predictable Stack Sharing: Fly-By-Wire Case Study.

of task T1 and T3, EMPRESS\textsuperscript{Ext} is able to achieve the same, minimal stack usage of EMPRESS for Fly-by-wire, which is optimal. Of course, a different selection of the extended tasks could have resulted in a different mapping. For the second case study, Autopilot, however, EMPRESS\textsuperscript{Ext} is not able to achieve the minimal stack usage of EMPRESS, but instead requires an additional 92 Byte (21%). Next, we derive more insights into the behavior of EMPRESS\textsuperscript{Ext}. To this end, we have computed the stack need using (i) EMPRESS [2], (ii) EMPRESS\textsuperscript{Ext} and (iii) dedicated stacks for all combinations of n extended tasks (with n ∈ \{1, 2, 3, 4, 5\} for Fly-by-wire and n ∈ \{1, 2, 3, 4, 5, 6, 7\} for Autopilot). We have assumed a percentage of p ∈ \{10, 20, …, 90, 100\} dedicated and 100 − p shared stack per extended task (compared to the task’s original complete stack need from Table V).

The average stack needs are shown in Figure 15 and 16.

![Stack need of Fly-by-wire, assuming n extended tasks with a dedicated stack of p% and a shared stack of (100 − p)% per extended task.](image2)

Fig. 15. Stack need of Fly-by-wire, assuming n extended tasks with a dedicated stack of p\% and a shared stack of (100 − p)\% per extended task.

As expected, the behavior of the original EMPRESS, which ignores extended tasks, and of the dedicated stacks do not depend on the number of extended tasks or on the percentage p. These solutions, which are reflected by Equations (4) and (6) in Section IV-B, respectively, provide a safe lower and upper bound on the stack need of EMPRESS\textsuperscript{Ext}. From Figure 15 and 16, we see that EMPRESS\textsuperscript{Ext} provides a stack configuration with a stack need reasonably close the theoretical lower bound of EMPRESS, as long as the number of extended tasks and/or the percentage of the dedicated stack remains below 50%.

![Stack need of Autopilot, assuming n extended tasks with a dedicated stack of p\% and a shared stack of (100 − p)\% per extended task.](image3)

Fig. 16. Stack need of Autopilot, assuming n extended tasks with a dedicated stack of p\% and a shared stack of (100 − p)\% per extended task.

VIII. DISCUSSION

As mentioned in the introduction, stack sharing between basic tasks and extended tasks is also facilitated by the RTA-OSEK RTOS from ETAS [8]. For the approach of ETAS,
additional buffer space is reserved for the dedicated stack area of each extended task. The dedicated stack area of an extended task is copied from the stack to its buffer upon self-suspension of an instance of the task as well as upon completion of an instance. Similarly, the dedicated stack is copied from the buffer to the shared stack upon resumption of an instance as well as upon the start of an instance. The amount of memory required is therefore equal to the stack memory requirement when EMPRESS is applied, see Equation (2), plus the sum of the dedicated stack usage of the extended tasks, i.e.,

\[ SU_{ETAS}(T) = SU_{EMPRESS}(T) + \sum_{i \in E} SU_{D(i)}. \]

This may result in significant memory savings compared to an approach where none of the basic tasks is allowed to share their stack with an extended task; see Equation (5). However, these memory savings come at the cost of additional overhead for copying the dedicated stacks of the extended tasks between the shared stack area and the buffers, and therefore at the cost of reduced schedulability. Hence, the approach in [8] effectively trades memory space for schedulability.

We now illustrate through the specific configuration of the case study presented in Section VII that the patented approach of ETAS and EMPRESSExt are incomparable from a stack memory requirements perspective; see also Table VI. Whereas the approach of ETAS needs less memory space than EMPRESSExt for Autopilot, EMPRESSExt needs less memory space than ETAS for Fly-by-wire. For systems with very demanding memory requirements and sufficient computational power, a hybrid approach combining the approach of ETAS with our novel approach denoted by EMPRESSExt may therefore be desirable. Further investigation falls outside the scope of this paper, however, and is future work.

### IX. Conclusion

In this paper, we considered stack memory requirements of AUTOSAR/OSek-compliant scheduling policies for a mixed set of basic tasks, i.e. tasks with a single-shot execution, and extended tasks, i.e. tasks that may leave data on the stack between instances and may suspend themselves. We started by revisiting the literature on stack sharing between basic tasks, and presented several novel insights. We subsequently proved that the problem of finding a stack layout with a minimal stack requirement for a mixed set of basic and extended tasks is NP-hard. We therefore presented an heuristic-based algorithm, termed EMPRESSExt, that aims at maximizing the overlap of basic tasks and extended tasks. Similar to EMPRESS [2], the resulting stack layout provides predictable stack sharing. We evaluated EMPRESSExt using a case study of an unmanned aerial vehicle, PapaBench. Finally, we briefly compared EMPRESSExt with the patented approach from ETAS, and found that both approaches are incomparable.

### References


### Table VI

<table>
<thead>
<tr>
<th></th>
<th>Fly-by-wire</th>
<th>Autopilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUETAS</td>
<td>44 + 36 = 180</td>
<td>424 + 76 = 500</td>
</tr>
<tr>
<td>SUEMPRESSExt</td>
<td>144</td>
<td>516</td>
</tr>
</tbody>
</table>