Stack memory requirements of AUTOSAR/OSEK-compliant scheduling policies

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.

An extended version of the paper is available as technical report [5].

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 [11] and Rubus [22]. The theoretical foundation for stack sharing between such tasks in systems with priority-based scheduling has been laid in [3, 8, 21, 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 [24]. 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 [20] 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-OSEK RTOS from ETAS [9]. 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) [23] 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 [12] 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$^{\text{Ext}}$, aiming at a minimal stack requirement for a mixed task set with predictable stack sharing. Finally, an evaluation of EMPRESS$^{\text{Ext}}$ 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 is an NP-hard problem. Section VI presents EMPRESS$^{\text{Ext}}$. The evaluation of EMPRESS$^{\text{Ext}}$ is the topic of Section VII. A brief comparison between the patented approach of ETAS and EMPRESS$^{\text{Ext}}$ is presented in Section VIII. 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 $\mathcal{T}$ of n tasks τ_1, τ_2, … τ_n, and fixed-priority scheduling with preeption thresholds (FPTS). Every task τ_i has a priority π_i and a preemption threshold θ_i, where higher values for priorities represent higher priorities and $\theta_i \geq \pi_i$. Under FPTS, a task τ_i is only allowed to preempt a task τ_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 $\Pi(\mathcal{T})$ and $\Theta(\mathcal{T})$ to denote the set of priorities and preemption thresholds associated with the tasks in $\mathcal{T}$, respectively, i.e.

$$\begin{aligned}\Pi(\mathcal{T}) &= \{\pi \mid \underset{\tau_i \in \mathcal{T}}{\exists} \pi_i = \pi\},\\ \Theta(\mathcal{T}) &= \{\theta \mid \underset{\tau_i \in \mathcal{T}}{\exists} \theta_i = \theta\}.\end{aligned}$$

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 $\mathcal{T}$ is partitioned in two sets, a set $\mathcal{T}^{\mathrm{B}}$ of *basic* tasks and a set $\mathcal{T}^{\mathrm{E}}$ 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 address-mapping from cache to main memory. Such a mapping is common amongst many embedded architectures and embedded operating systems [17] 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 [7, 16] 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 [18].

Using these techniques, we can derive for each task τ_i its maximum stack usage $SU_i \in \mathbb{N}^0$. For the sake of simplicity, we assume that SU_i provides the maximum stack usage of task τ_i including the size of the stack frame. The stack memory needed by any two pre-empting tasks τ_i and τ_j is therefore bounded by $SU_i + SU_j$.

For an extended task τ_i^{E}, 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_i^{D} can be determined, or at least safely bounded. The size of the shared stack SU_i^{S} can subsequently be derived using $SU_i^{\mathrm{S}} = SU_i - SU_i^{\mathrm{D}}$.

d) Pre-emption Relation and Pre-emption Graph: We assume a binary pre-emption relation $<^{\mathrm{a}}$ 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 $\tau_j <^{\mathrm{a}} \tau_i$ holds if and only if task τ_j can be pre-empted by task τ_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* ($\neg\tau <^{\mathrm{a}} \tau$) and *transitive* ($\tau_k <^{\mathrm{a}} \tau_j \wedge \tau_j <^{\mathrm{a}} \tau_i \Rightarrow \tau_k <^{\mathrm{a}} \tau_i$). Given $<^{\mathrm{a}}$, 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 τ_i to τ_j represents that $\tau_j <^{\mathrm{a}} \tau_i$ holds.

Without loss of generality, we assume $\tau_j <^{\mathrm{a}} \tau_i \Rightarrow j > i$, i.e. when task τ_j can be pre-empted by task τ_i, τ_j has a higher index than τ_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 [14] 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 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

In this section, we briefly summarize (i) work on predictable stack sharing and (ii) our findings regarding existing approaches to determine the maximum stack usage $SU(\mathcal{T}^{\mathrm{B}})$ of a set of basic tasks $\mathcal{T}^{\mathrm{B}}$. We consider approaches based on *partitioning* [3, 8, 21, 12] and on *possible preemption chains* [4].

A. Predictable stack sharing [2]

In [2], an Efficient and effective Method for Predictable Stack Sharing (EMPRESS) 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 [6, 15]. 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

```
Algorithm 1: TaskStackAddress($\mathcal{T}$, $\prec$, $SU$)
Input: A set of tasks $\mathcal{T}$, a pre-emption relation $\prec$ (SPO),
    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 $\tau_i$.
for each $\tau_i$ (from highest to lowest index $i$) do
    $SA_i \leftarrow 0$;
    for each $\tau_j$ with $j > i$ do
        if $\tau_j \prec \tau_i$ then
            $SA_i \leftarrow \max(SA_i, SA_j + SU_j)$;
        end if
    end for
end for
```

task. The maximum stack address of task τ_i is given by the maximum sum of the stack address SA_j and the stack usage SU_j, where τ_j is potentially pre-empted by task τ_i. The derived stack address of each task is relative to the system-wide static stack pointer, which is set to memory address 0. SA_i therefore does not provide an absolute address.

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

$$SU^{\text{EMPRESS}}(\mathcal{T}) = \max_{\tau_i \in \mathcal{T}} (SA_i + SU_i) . \quad (1)$$

B. Our findings on determining the maximum stack usage

Below, we summarize our findings regarding existing approaches to determine the maximum stack usage. Further explanations and justifications of our findings can be found in the technical report [5].

1) The *non-preemption groups* in [8] are a special case of partitioning in *non-preemptive groups* in [21].
2) As mentioned in [12], 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 [21].
3) An *optimal set of non-preemptive groups* [12] need not yield the minimal stack usage.
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

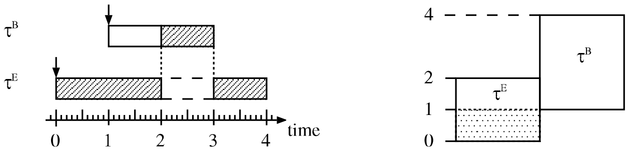
In Section II, we assumed a binary relation $\prec^{\text{a}}$ on tasks, which is derived from the priority levels and/or pre-emption levels of the tasks. As shown in [4], the DAG of allowed preemptions can be pruned to a DAG of possible preemptions for a set of basic tasks with offsets and precedences. Viewed as a set of pairs, the set of possible preemptions $\prec^{\text{p}}$ is therefore a subset of $\prec^{\text{a}}$, i.e. $\prec^{\text{p}} \subseteq \prec^{\text{a}}$.

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 $\mathcal{T}_{\text{III}}$ of two independent tasks, a basic task τ^{B} and an extended task τ^{E}, with characteristics as given in table I, which are scheduled using FPTS.

TABLE I
Characteristics of a basic task τ^{B} and an extended task τ^{E}.

task	π	θ	C	$T = D$	SU^{S}	SU^{D}
τ^{B}	2	2	1	5	3	0
τ^{E}	1	2	2	7	1	1

Based on their priorities and preemption thresholds, tasks τ^{B} and τ^{E} are mutually non-preemptive, i.e. $\neg\tau^{\text{B}} \prec^{\text{a}} \tau^{\text{E}} \wedge \neg\tau^{\text{E}} \prec^{\text{a}} \tau^{\text{B}}$. As illustrated in Figure 1(a), however, τ^{B} may execute when a job of τ^{E} suspends itself, which looks similar to a preemption of τ^{E} by τ^{B}. Because τ^{E} and τ^{B} are mutually non-preemptive based on priorities and preemption thresholds, task τ^{E} may still share its shared stack area with τ^{B}, as illustrated in Figure 1(b).



(a) A timeline for τ^{B} and τ^{E}.

(b) A stack layout for τ^{B} and τ^{E}.

Fig. 1. Despite the fact that the basic task τ^{B} may execute while the extended task τ^{E} is suspended (a), τ^{E} can share its shared stack area with τ^{B} (b). The box with the dashed border in (a) denotes the self-suspension of τ^{E} and the dotted box in (b) denotes the dedicated stack area of τ^{E}.

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, 8, 21, 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 and upper bounds on the stack usage of a mixed task set $\mathcal{T}$.

A first lower bound $SU^{\text{min}}(\mathcal{T})$ 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.

$$SU^{\text{min}}(\mathcal{T}) = \sum_{\tau_i \in \mathcal{T}^{\text{E}}} SU_i. \tag{2}$$

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

$$SU^{\text{lwb}}(\mathcal{T}) = SU^{\text{EMPRESS}}(\mathcal{T}). \tag{3}$$

We observe that $SU^{\text{min}}(\mathcal{T})$ and $SU^{\text{lwb}}(\mathcal{T})$ are incomparable.

We can find an upper bound $SU^{\text{upb}}(\mathcal{T})$ by assuming that none of the basic tasks can share their stack with extended tasks, i.e. by combining Equations (2) and (1):

$$SU^{\text{upb}}(\mathcal{T}) = SU^{\text{min}}(\mathcal{T}) + SU^{\text{EMPRESS}}(\mathcal{T}^{\text{B}}). \tag{4}$$

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

$$SU^{\text{max}}(\mathcal{T}) = \sum_{\tau_i \in \mathcal{T}} SU_i. \tag{5}$$

We observe that $SU^{\text{upb}}(\mathcal{T}) \leq SU^{\text{max}}(\mathcal{T})$.

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

V. Complexity Analysis

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

We provide an NP-completeness proof by restriction [13] 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.

The proof of Theorem 1 has the following structure. We restrict our original minimization problem $\mathcal{P}$ to a decision problem. Next, we show that there exists a one-to-one correspondence between the instances of the bin-packing decision problem and the restriction of the minimization problem $\mathcal{P}$ to a decision problem that preserves "yes" and "no" answers, proving that the latter is an NP-complete problem. We then conclude that the original minimization problem $\mathcal{P}$ is NP-hard.

The proof of Theorem 1 is in the technical report [5].

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 (2). 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($\mathcal{T}$, $\prec$, SU)

```
Input: A set of tasks $\mathcal{T}$, a pre-emption relation $\prec$,
    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 $\tau_i$.
$\mathcal{T}^{ws} \leftarrow \mathcal{T} \setminus \{\tau_l^{\text{E}} | \tau_l^{\text{E}} \in \mathcal{T}$ is an extended task$\}$;
$SA^g \leftarrow 0$;
for each $\tau_l^{\text{E}}$ (from highest to lowest index $l$) do
    $\mathcal{T}^c \leftarrow \{\tau_x | \tau_x \in \mathcal{T}^{ws} \wedge \tau_l^{\text{E}} \not\prec \tau_i \wedge \tau_i \not\prec \tau_l^{\text{E}}\}$;
    $SA_l = SA^g$
    $SA^g = SA_l + SU_l$
    for each $\tau_i \in \mathcal{T}^c$ (from highest to lowest index $i$) do
        $SA_i \leftarrow SA_l + SU_l^{\text{D}}$;
        for each $\tau_j \in \mathcal{T}^c$ with $j > i \wedge \tau_j \notin \mathcal{T}^{ws}$ do
            if $\tau_j \prec \tau_i$ then
                $SA_i \leftarrow \max(SA_i, SA_j + SU_j)$;
            end if
        end for
        if $SA_i < SA_l + SU_l$ then
            $\mathcal{T}^{ws} \leftarrow \mathcal{T}^{ws} \setminus \{\tau_i\}$;
            $SA^g \leftarrow \max(SA_i + SU_i, SA^g)$
        end if
    end for
end for
for each $\tau_i \in \mathcal{T}^{ws}$ (from highest to lowest index $i$) do
    $SA_i \leftarrow SA^g$;
    for each $\tau_j \in \mathcal{T}$ with $j > i \wedge j \notin \mathcal{T}^{ws}$ do
        $SA_i \leftarrow \max(SA_i, SA_j + SU_j^{\text{D}})$;
        if $\tau_j \prec \tau_i$ then
            $SA_i \leftarrow \max(SA_i, SA_j + SU_j)$;
        end if
    end for
end for
```

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

The first outer loop (line 3 to 19) iterates over all extended tasks. For each extended task τ_l^{E}, we set the stack address of τ_l^{E} to the global stack address SA^g. Next, we construct a set $\mathcal{T}^c$ by selecting all basic tasks from the working-set $\mathcal{T}^{ws}$ that are mutually non-preemptive with τ_l^{E} and hence can share the stack with the shared stack SU_l^{S} of τ_l^{E}; see Observation 4. For each such mutually non-preemptive basic task τ_i, we initially set the stack address to the stack address of τ_l^{E} incremented by the dedicated stack size SU_l^{D} of τ_l^{E}. The inner-most loop (line 9 to 13) iterates over all tasks in $\mathcal{T}^c$ that may be preempted by τ_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 τ_i shares parts of its stack with the extended task τ_l^{E}. If so, then the basic task is removed from $\mathcal{T}^{ws}$ and the global stack address SA^g 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$^{\text{Ext}}$.

We merely observe that EMPRESS$^{\text{Ext}}$ 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$^{\text{Ext}}$ is given by

$$SU^{\text{EMPRESS}^{\text{Ext}}}(\mathcal{T}) = \max_{\tau_i \in \mathcal{T}} (SA_i + SU_i)\,. \tag{6}$$

VII. Evaluation

In this section, we exemplify and evaluate EMPRESS$^{\text{Ext}}$ using the case study from [2]. The case study is based on PapaBench [19] a free real-time benchmark implementing the control software of an unmanned aerial vehicle (UAV). It contains two disjoint task sets, *Fly-by-wire* and *Autopilot*. The pre-emption graphs for both task sets are shown in Figure 2.

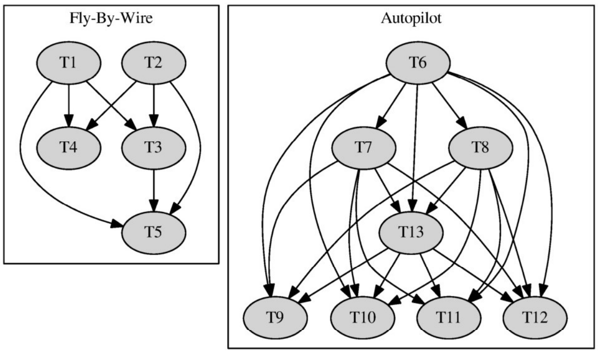


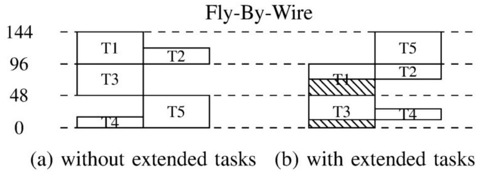
Fig. 2. Pre-emption constraints for PapaBench based on the precedence constraints, task frequencies and task priorities.

We have analyzed stack usages of PapaBench with Absint's static stack analyzer [14], which is used in industry to detect and prevent stack overflows. Table II provides the stack usages.

TABLE II
Maximal stack usage and stack address (Predictable Stack Sharing) for all 13 PapaBench Benchmarks for the ARMv7. The provided stack addresses are relative to a system-wide stack pointer and based on EMPRESS$^{\text{EXT}}$.

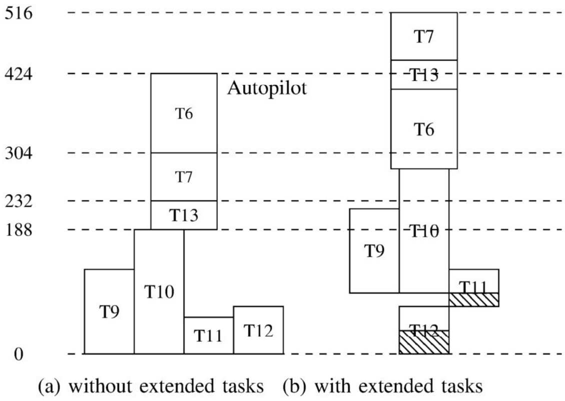
Task	Stack Usage (Byte)			Stack
	Total	Shared	Dedicated	address
	(SU)	(SU^{S})	(SU^{D})	(SA)
T1	48	24	24	48
T2	24	24	0	72
T3	48	36	12	0
T4	16	16	0	12
T5	48	48	0	96
T6	120	100	20	280
T7	72	72	0	444
T8	0	0	0	444
T9	128	128	0	92
T10	188	188	0	92
T11	56	36	20	72
T12	72	36	36	0
T13	44	44	0	400

The PapaBench [19] benchmarks are originally all scheduled as basic tasks. To demonstrate the difference between EMPRESS [2] and EMPRESS$^{\text{Ext}}$, we have randomly assumed that tasks T1, T3, T6, T11, and T12 are extended tasks. The results are shown in Figure 3 for Fly-by-wire, and Figure 4 for Autopilot. Despite the dedicated stack areas of task T1 and T3, EMPRESS$^{\text{Ext}}$ 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$^{\text{Ext}}$ is not able to achieve the stack usage of EMPRESS, but instead requires an additional 92 Byte (21%).



(a) without extended tasks (b) with extended tasks

Fig. 3. Stack layout and relative stack addresses for Predictable Stack Sharing: Fly-By-Wire Case Study.



(a) without extended tasks (b) with extended tasks

Fig. 4. Stack layout and relative stack addresses for Predictable Stack Sharing: Autopilot Case Study.

Further evaluation of EMPRESS$^{\text{Ext}}$ for various combinations of the number of extended tasks and the percentage of shared stack per extended task can be found in the technical report [5].

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 [9]. 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 (1), plus the sum of the dedicated stack usage of the extended tasks, i.e. $SU^{\text{ETAS}}(\mathcal{T}) = SU^{\text{EMPRESS}}(\mathcal{T}) + \sum_{\tau_i \in \mathcal{T}^{\text{E}}} SU_i^{\text{D}}$.

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 (4). 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 [9] 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 EMPRESS$^{\text{Ext}}$ are incomparable from a stack memory requirements perspective; see also Table III. Whereas the approach of ETAS needs less memory space than EMPRESS$^{\text{Ext}}$ for *Autopilot*, EMPRESS$^{\text{Ext}}$ 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 EMPRESS$^{\text{Ext}}$ may therefore be desirable. Further investigation, however, is future work.

TABLE III

Stack memory requirements of *Fly-by-wire* and *Autopilot* based on the configuration described in Table II.

SU	*Fly-by-wire*	*Autopilot*
SU^{ETAS}	144 + 36 = 180	424 + 76 = 500
$SU^{\text{EMPRESS}^{\text{Ext}}}$	144	516

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 EMPRESS$^{\text{Ext}}$, 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 EMPRESS$^{\text{Ext}}$ using a case study of an unmanned aerial vehicle, PapaBench. Finally, we briefly compared EMPRESS$^{\text{Ext}}$ with the patented approach from ETAS, and found that both approaches are incomparable.

References

[1] AUTOSAR – Specification of Operating System, Release 4.4.0. Technical report, 2019. [Online], Available: https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/SystemServices.zip.

[2] S. Altmeyer, R.J. Bril, and P. Gai. EMPRESS: an Efficient and effective Method for PREdictable Stack Sharing. In *Proc. IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug. 2018.

[3] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[4] M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin. Bounded shared-stack usage in systems with offsets and precedences. In *Proc. 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 276–285, July 2008.

[5] R.J. Bril, S. Altmeyer, and P. Gai. Stack memory requirements of autosar/osek-compliant scheduling policies. Technical report, June 2019. See: https://pure.uva.nl/ws/files/36529791/main.pdf.

[6] Thomas W. Carley. Private communication, June 2003.

[7] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt-driven programs. In *Proc. 10th International Symposium on Static Analysis (SAS)*, pages 109–126, June 2003.

[8] R.I. Davis, N. Merriam, and N. Tracey. How embedded applications using an RTOS can stay within on-chip memory limits. *Proc. WiP and Industrial Experience Sessions, 12th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 71–77, 2000.

[9] ETAS-RTA-OSEK. RTA-OSEK User Guide. RTA-OSEK v5.0.2.

[10] P. Gai. *Real Time Operating System design for Multiprocessor system-on-a-chip*. PhD thesis, Scuola Superiore S. Anna, Italy, 2004.

[11] P. Gai, E. Bini, G. Lipari, M. Di Natale, and L. Abeni. Architecture for a portable open source real time kernel environment. In *Proc. 2nd Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, Nov. 2000.

[12] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilizations of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 73–83, Dec. 2001.

[13] M.R. Garey and D.S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[14] AbsInt Angewandte Informatik GmbH. Static Stack Analyzer. https://www.absint.com/stackanalyzer/index.htm, 2018. [Online; accessed 22-February-2018].

[15] K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin. Determining maximum stack usage in preemptive shared stack systems. In *Proc. 27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 445–453, Dec. 2006.

[16] D. Kästner and C. Ferdinand. Proving the absence of stack overflows. In *Proc. 33rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 202–213, Sep. 2014.

[17] D. Kleidermacher and M. Griglock. Safety-critical operating systems. http://www.embedded.com/design/prototyping-and-development/4023830/Safety-Critical-Operating-Systems. Accessed: 2017-01-10.

[18] P. Koopman. A case study of Toyota unintended acceleration and software safety, Nov. 2014.

[19] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. PapaBench: a Free Real-Time Benchmark. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2006.

[20] OSEK group. OSEK/VDX operating system. Technical report, Feb. 2005. [Online], Available: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[21] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, pages 25–34, Dec. 2000.

[22] Arcticus Systems. http://www.arcticus-systems.com.

[23] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. IEEE 6th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 328–335, Dec. 1999.

[24] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), April 2009.