EMPRESS: an Efficient and effective Method for PREdictable Stack Sharing

Sebastian Altmeyer[1], Reinder J. Bril[2] and Paolo Gai[3]
[1]University of Amsterdam (UvA), Amsterdam, The Netherlands
[2]Technische Universiteit Eindhoven (TU/e), Eindhoven, 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. On the downside, stack sharing decreases the predictability of a system, e.g. may give rise to a substantial variation in the address space for the memory locations used for the stack of a task. As a result, the precision of execution-time bounds may be reduced, the pessimism in schedulability analysis increased, and optimizations to increase schedulability hampered.**

In this paper, we present EMPRESS, an Efficient and effective Method for PREdictable Stack Sharing. We assume priority-based scheduled systems, where the binary pre-emption relation on tasks is a strict partial order, and static bounds on each task's stack usage. Both assumptions are common in the embedded real-time domain. For such systems, EMPRESS provides a predictable stack sharing between tasks, i.e. the stack of every task is always located in the very same memory area, even for tasks sharing a stack. It therefore combines the predictability of dedicated stack spaces with the reduced memory need of a shared stack. We exemplify the benefits of EMPRESS using as a case study an implementation of an unmanned aerial vehicle, and explain how EMPRESS can be realized within the Erika Enterprise RTOS without additional overheads.

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, such as Erika Enterprise [13] and Rubus [31].

Compared to dedicated stacks for tasks, stack sharing between tasks increases the uncertainty in addresses of memory accesses. In a setting with a shared stack, the positioning of a stack frame, and with it the positioning of local data, depends on the current progress and the set of active tasks in case of pre-emptions. Static timing analyses for hard real-time systems require precise knowledge about the addresses of all memory accesses, however. Any variation within the address space may therefore prohibit the use of static timing verification and/or reduce the precision of the execution-time bounds [33]. This may result in increased pessimism in the schedulability analysis. Whenever programmable platforms of these systems contain a cache to bridge the gap between the processor speed and main memory speed, the schedulability analysis [3, 7] may become even more pessimistic, and optimizations of the layout of tasks in memory [26] may be hampered. The initial stack pointer of each task should therefore preferably be fixed and known statically to enable offline analysis and optimizations.

We consider priority-based scheduled systems, where the binary pre-emption relation between tasks is a strict partial order (SPO), i.e. is irreflexive and transitive. Further, we assume a system without virtual memory, or any other memory address translation, static bounds on each task's stack usage, and that all tasks share a common memory space

For such systems, we present EMPRESS, an Efficient and effective Method for PREdictable Stack Sharing, that enforces a fixed and statically known initial stack pointer for each task. It therefore combines the predictability of dedicated stack spaces with the reduced memory need of a shared stack, and it can be realized within the Erika Enterprise RTOS without additional overheads.

The paper is structured as follows: In Section II, we introduce our system model and the required technical background. Section III introduces EMPRESS, the Efficient and effective Method for PREdictable Stack Sharing. Section IV introduces the case study that we use to exemplify the benefits of EMPRESS: Section V discusses the reduction in the total stack size, and Section VI discusses the increased predictability. Section VII details how EMPRESS can be realized within ERIKA OS without additional overheads. The related work is described in Section VIII and Section IX concludes the paper.

II. Background

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

a) System Model: We assume a single-processor system, a set $\mathcal{T}$ of n tasks $\tau_1, \tau_2, \ldots \tau_n$, and priority-driven scheduling. 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. Tasks may share mutually exclusive resources using an *early blocking* resource access protocol, such as the stack resource policy (SRP) [4]. Tasks are not allowed to either suspend themselves or leave any data on the stack from one instance of the task to the next. 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 [23] 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 the initial stack pointer.

b) Maximal Stack Usage: As stack overflows are a common source of system failures, techniques exist to upper-bound the stack-usage [8, 22] 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 [24].

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 $SU_i + SU_j$.

c) Pre-emption Relation and Pre-emption Graph: We assume a binary pre-emption relation $\prec$ on tasks [6], which is derived from the priority levels and/or pre-emption levels of the tasks. The relation $\tau_j \prec \tau_i$ 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* ($\neg\tau \prec \tau$) and *transitive* ($\tau_k \prec \tau_j \wedge \tau_j \prec \tau_i \Rightarrow \tau_k \prec \tau_i$).

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

III. EMPRESS: an Efficient and effective Method for PREdictable Stack Sharing

In this section, we detail the predictable stack sharing method EMPRESS. First, we argue how the static stack address of each task can be derived, without increasing the worst-case stack usage, and then, we explain how the static stack pointers can be enforced during runtime. EMPRESS exploits a rather simple idea, but this simplicity is an advantage rather than a disadvantage as it allows the implementation of EMPRESS within a real-time operating system with minimal changes only.

A. Static Stack Usage Analysis

The method requires a static analysis of each task's stack usage and of the pre-emption relation. Once this data is available, we can derive the worst case, i.e., highest, stack pointer for each task by assuming that

- each task uses the stack up to its stack bound, and
- the worst-case pre-emption scenario occurs.

Both assumptions are conservative, but not necessarily pessimistic. Without any further information on either the stack usage or on the pre-emption relation, the worst-case situation must be considered feasible.

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 $\tau_i$.
1: for each $\tau_i$ (from highest to lowest index $i$) do
2:     $SA_i \leftarrow 0$;
3:     for each $\tau_j$ with $j > i$ do
4:         if $\tau_j \prec \tau_i$ then
5:             $SA_i \leftarrow \max(SA_i, SA_j + SU_j)$;
6:         end if
7:     end for
8: end for
```

Algorithm 1 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 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.

B. Stack Implementation

The aim of EMPRESS is to statically determine the memory address of each task's stack, and to enforce it during runtime. Whenever a new job of task τ_i is created, its stack will occupy the memory region $[SA_i\colon SA_i + SU_i - 1]$. For the sake of simplicity, we assume that all tasks share a common memory space, as for instance in TinyOS [11] and ERIKA Enterprise [13].

Some real-time kernels such as ERIKA Enterprise [13] and the RTOS part of the QP Framework [27] allow the possibility to provide a shared stack implementation. Thanks to a run to completion execution and to the Immediate Priority Ceiling protocol it is in fact possible to design a fixed priority scheduler with support for preemption allowing the existence of a single stack (more details in Section VII).

More complicated stack implementations, such as in FreeRTOS [5], can also be adapted, but we always assume a system without virtual memory, or any other memory address translation.

On pre-emption, the current value of the stack pointer must be stored (as usual), and replaced with the statically computed stack address SA_i of the new active task τ_i with the highest priority. On task completion, the old value of the stack pointer is restored and the previously pre-empted task continues execution.

A task's stack address therefore does not depend on the dynamics of the system, but is statically determined. Despite the static address of each task's stack space, the stacks are not statically allocated. Instead, the stacks of several tasks can share the same memory space, if the tasks are mutually non-preemptive; for instance due to shared priorities or limited-preemptive scheduling policies. The worst-case memory usage

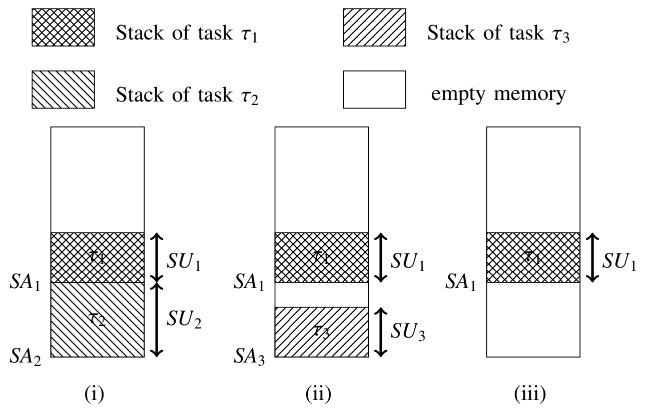

Fig. 1. The predictable shared stack implementation EMPRESS in action. In scenario (i), where τ_1 pre-empts τ_2, the stack of both tasks are placed sequentially, whereas in scenario (ii) and (iii) the memory is fragmented.

is not influenced by EMPRESS, but EMPRESS exhibits the same memory usage for the stack as a standard shared stack.

To the best of our knowledge, many real-time operating systems compute the memory address of a stack dynamically on task creation, such as FreeRTOS [5]. TinyOS [11] uses a limited number of statically allocated stacks, which already reduces the unpredictability, but the assignment of task to stack is still dynamic. Consequently, the worst-case memory usage may be increased, compared to our implementation, and the memory addresses of the stacks are still not completely predictable.

C. Example

In the following example, we assume a task set $\mathcal{T}$ consisting of three tasks τ_1, τ_2, and τ_3. Tasks τ_2 and τ_3 are mutually non-preemptive, but both can be pre-empted by task τ_1. We further assume that $SU_2 > SU_3$. Using Algorithm 1, we compute the stack addresses as follows:

$$SA_2 = SA_3 = 0$$

$$SA_1 = \max\{SU_2, SU_3\} = SU_2$$

Figure 1 shows how EMPRESS positions the tasks' stacks in three different scenarios: (i) τ_1 pre-empts τ_2, (ii) τ_1 pre-empts τ_3, and (iii) τ_1 pre-empts neither τ_2 nor τ_3. In the first scenario, the stack of both tasks τ_1 and τ_2 are positioned consecutively, whereas in the second scenario, a small gap of size $SU_2 - SU_3$ between the stacks of tasks τ_1 and τ_3 appears. In scenario three, only task τ_1 is active. Although the memory is fragmented in the second and third scenario, the worst-case memory usage, determined by scenario one, is not altered.

IV. Description of our Case Study

To exemplify the benefits of EMPRESS, we use PapaBench [28] a free real-time benchmark implementing the control software of an unmanned aerial vehicle (UAV). PapaBench is unique in that it provides a complete implementation including C-Code and a task-set definition with implicit deadlines, periods and precedence constraints.

A. Benchmark Suite: PapaBench

PapaBench provides 13 tasks statically assigned to two processors. Due to the static task partitioning to the two processors, we treat the 13 tasks as two disjoint benchmarks: T1 to T5 implement Fly-By-Wire functionality, and tasks T6 to T13 implement an Autopilot. Note that we have taken the notation from [28] and therefore write T1 instead τ_1. Moreover, the index of tasks in PapaBench does not correspond to the task's priority; the benchmark suite does not determine a scheduling policy but just provides the task-set description. Table I shows the description and frequency of all 13 tasks. The precedence constraints are provided in Figure 2.

Task	Program Description	Freq.	Period
T1	Receive Radio-Command	40Hz	25ms
T2	Send Data to MCU0	40Hz	25ms
T3	Receive MCU0 values	20Hz	50ms
T4	Transmit Servos	20Hz	50ms
T5	Check Failsafe	20Hz	50ms
T6	Managing Radio orders	40Hz	25ms
T7	Stabilization	20Hz	50ms
T8	Send Data to MCU1	20Hz	50ms
T9	Receive GPS Data	4Hz	250ms
T10	Navigation	4Hz	250ms
T11	Altitude Control	4Hz	250ms
T12	Climb Control	4Hz	250ms
T13	Reporting Task	10Hz	100ms

TABLE I
Description of the 13 PapaBench tasks. Upper tasks $T1$ to $T5$ implement Fly-By-Wire functionality, whereas tasks $T6$ to $T13$ implement an autopilot; see [28], Table 1 and 2.

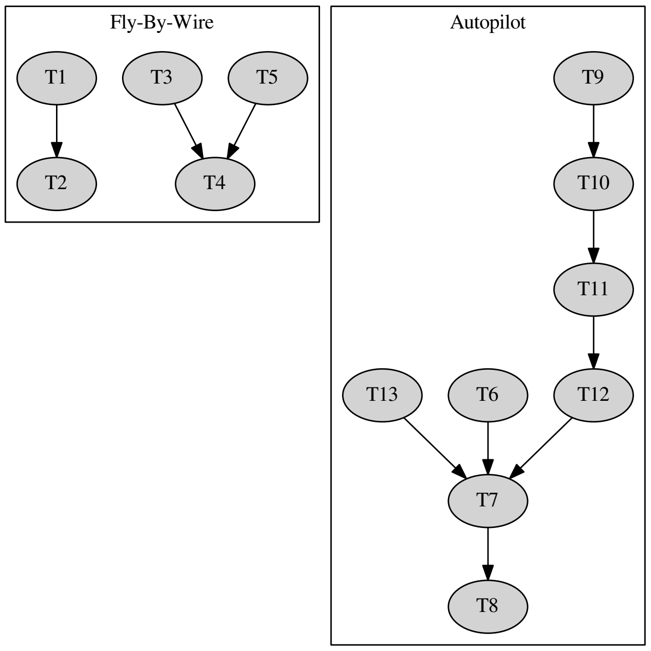

Fig. 2. Precedence constraints for PapaBench; see [28], Figure 2.

B. Target Architecture: ARMv7

As target architecture for the evaluation and case study, we have used an ARMv7[1]. The ARMv7 is a common embedded architecture, for which we have access to Absint's Stack Analyzer [17] and Absint's timing analysis tools [12] (a3[2] and Timing Explorer[3]), and to which PapaBench has been ported[4].

V. Reduction of the total stack usages

In this section, we discuss the reduction of the total stack usage of EMPRESS compared to a non-shared stack implementation with a dedicated stack region per task. Comparisons between a shared and a dedicated stack implementation have been published before various times. We will therefore restrict the evaluation to our case study only; a discussion about the general results is provided in this section for the sake of completeness.

A. Stack Usage Reduction for PapaBench

To evaluate the reduction in the stack size, we first need to analyze the stack need of our benchmarks. We have analyzed stack usages of PapaBench with Absint's static stack analyzer [17], which is used in Industry to detect and prevent stack overflows. Next, we need to deduct the pre-emption relation: From Table I and Figure 2, we see that the precedence and frequencies are not entirely coherent; despite a precedence from T12 to T7, T7 executes 5 times as often, and despite a precedence from T6 to T7, T6 has twice the frequency of T7. The precedence constraints alone therefore do not suffice to deduct the pre-emption relation. We need in addition the scheduling policy and configuration, which is not provided by the description of PapaBench. To this end, we assign static priorities[5] based on task frequencies with task indices used to break ties. The resulting priorities are shown in Table II, the pre-emption graphs for both task sets in Figure 3.

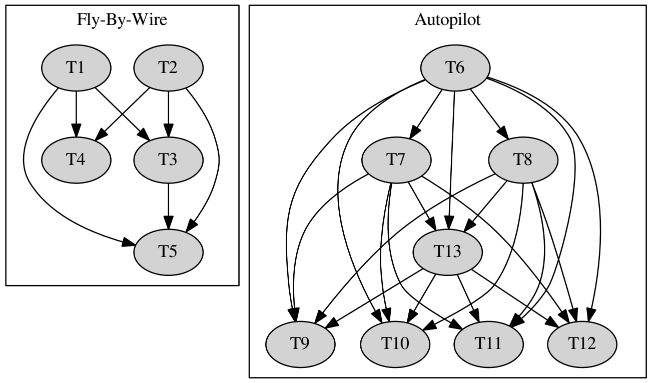

Fig. 3. Pre-emption constraints for PapaBench based on the precedence constraints (Figure 2), task frequencies (Table I) and task priorities (Table II). A directed edge from Tx to Ty reads as Tx can pre-empt Ty.

[1]https://developer.arm.com/products/processors/cortex-a/cortex-a7
[2]https://www.absint.com/ait/index.htm
[3]https://www.absint.com/timingprofiler/index.htm
[4]https://github.com/t-crest/patmos-benchmarks/tree/master/PapaBench-0.4
[5]A lower value represents a higher priority.

The initial stack addresses for EMPRESS, our predictable stack sharing method, as computed by Algorithm 1, are shown in Table II, 4th column. The stack configurations for both task sets are presented in Figure 4. The total stack usage for EMPRESS is 144 (Fly-by-wire) and 424 (Autopilot), compared to 184 and 680 using dedicated stacks for tasks, respectively. This means a reduction of 21% and 37% of the stack size and thus of the required memory.

Task	Priority	Stack Need (Byte)	Init. Stack pointer
T1	1	48	96
T2	2	24	96
T3	3	48	48
T4	4	16	0
T5	5	48	0
T6	1	120	304
T7	2	72	232
T8	3	0	232
T9	5	128	0
T10	6	188	0
T11	7	56	0
T12	8	72	0
T13	4	44	188

TABLE II
Assigned priorities, maximal stack usage, and stack pointer (Predictable Stack Sharing) for all 13 PapaBench Benchmarks for the ARMv7. The provided stack pointers are relative to a system-wide stack pointer.

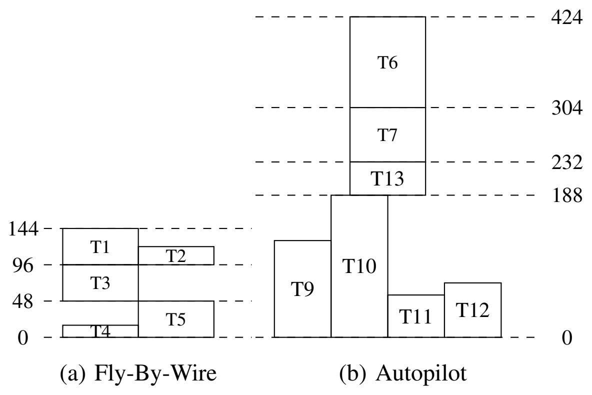

Fig. 4. Stack Pointer for the Predictable Stack Sharing.

B. General Discussion about the Stack Usage Reduction

The reduction in the total stack usage of a shared stack compared to dedicated stack regions strongly depends on the assumptions on the system. Estimates that have been published in related work [10, 6] are therefore only valid under the assumptions made in these papers.

For fully pre-emptive systems without any precedence constraints or resource sharing, the stack usage of both alternatives is the same: The worst-case assumption for a shared stack, and hence for EMPRESS, is a fully nested pre-emption of all tasks at each task's worst-case stack usage. Fully pre-emptive systems without precedence constraints or resource sharing, however, are unrealistic and mostly assumed in academic papers. We know for instance from [25] and from the Industrial Challenge[6] of ECRTS 2017 that task-sets in the automotive

[6]https://waters2017.inria.fr/challenge/#Challenge17

industry are far from fully pre-emptive, but instead exhibit a high number of precedence constraints and heavy resource-sharing. PapaBench, a strongly simplified example of the drone industry also exhibits more constraints on task pre-emption than tasks. We therefore consider the 21% and 37% stack usage reduction of our case study to be on the low-end side. Furthermore, even fully pre-emptive systems can be adapted, for instance using non-preemptive regions or fixed-priority threshold scheduling (FPTS) to reduce the stack usage while preserving schedulability: Davis et al. [10] report a reduction in the total stack usage by 75% on average of a shared stack with systems using between 16 and 32 tasks.

VI. Impact on the Predictability

In this section, we discuss the impact of EMPRESS on the predictability of an embedded real-time system compared to a standard shared stack: The difference between the two is the certainty about the stack pointer. EMPRESS provides and guarantees a unique stack pointer per task, just like with dedicated stack areas, whereas a standard shared stack can only provide a range of potential initial stack addresses.

We discuss the implication of, and support for, unknown stack pointers in timing analysis tools, examine the number of potential stack pointers that must be considered, and evaluate the increased precision of static timing analysis tools due to a known initial stack pointer.

Since both, dedicated stack areas and EMPRESS guarantee statically known initial stack pointers, this section provides an evaluation of dedicated stack region against a shared stack implementation, in terms of predictability, as well as an evaluation of EMPRESS against a shared stack implementation in terms of predictability.

A. Timing Analysis Tools and Stack Pointer

Precise knowledge about the stack pointer greatly improves the precision of timing analyses and is a pre-requisite for various optimizations. With an unknown initial stack pointer, the memory addresses of each access to a local variable remain unknown. This has two consequences:

1) Each memory access to a local variable must be considered a cache miss and thus increases the execution time bound during the timing analysis. As such accesses to local variables are common, the impact on the precision is significant.
2) Each unknown memory access pollutes the cache and therefore reduces the precision of the cache analysis for all accesses. Without knowing the exact cache set to which a memory access is mapped, the cache analysis has to assume conservatively that the access maps to any cache set, which greatly reduces the precision of the analysis.

The industrial standard for static timing analyses, Absint's Worst-Case Execution Time analyzer [12] a3 and Timing Explorer, require a stack pointer as input to the analysis. If the user does not provide an initial stack pointer, the analysis tool guesses a pointer. An incorrect stack pointer may result in an optimistic execution time bound, namely if the guessed stack pointer leads to a lower execution time than the correct stack pointer, or it may result in a highly pessimistic bound if the stack is falsely mapped to a slow memory region with high memory access times.

An extension to the static cache analysis, the relational cache analysis [18] has the potential to alleviate the pessimism due to imprecise memory addresses, but has neither been extended to the case of unknown stack pointer (it has been developed for array-accesses within loops), nor has it been adopted yet by any commercial timing analysis tool.

Optimizations of the task layout [15, 26], i.e., the mapping of memory blocks to cache sets, also require a known stack pointer — unless the optimizations are restricted to the instruction cache only. Similarly, analyses for the cache-related preemption delays (CRPD) [3, 7] of data caches require a known task layout, and therefore fixed stack pointers. Uncertainty in the stack pointer lead to unknown memory accesses and strongly impair the precision of the CRPD.

We note that analyses of and cache optimization for data caches, even those published by authors of this work [2, 3, 7, 15, 26], implicitly assume known stack pointers, without discussing this topic any further.

B. Reduction in the Number of potential Stack Pointers

In this section, we discuss for our case study the number of potential initial stack pointers that must be considered by the worst-case execution time analysis. For each potential stack pointer, a new execution time analysis has to be performed in order to derive a precise and safe bound on the task's execution time. In EMPRESS, a task's initial stack address is by construction statically determined and does not vary during runtime. In stark contrast, the variation of initial stack addresses in case of a standard shared stack may be significant.

To provide an indication of the range of possible stack pointers, we examine the number of stack pointers for our case study based on the pre-emption relation and the task's stack usages. The range of potential stack pointers for all tasks – for a standard shared stack – can be deduced directly from Table II. The difference between a standard shared stack and the predictable shared stack is that we fix the stack pointers at the worst-case position. As the stack pointers are word-aligned, we have to adjust the range [0 : *max*] by dividing by 4 to derive the number of admissible stack pointers.

Task T6 for instance, has the lowest priority in the second task set (Autopilot) and therefore can only execute if no other task is currently ready. Therefore, it has exactly one possible stack pointer, namely 0. Remember that the stack addresses are relative to a system-wide stack pointer set to 0. T9, however has 79 (= 236/4) potential initial stack pointers: it may pre-empt any of the task T6, T7, T8 and T13, potentially with nested pre-emption at any program point, and hence stack pointers from 0 to 236 (in steps of 4), or it may execute with relative initial stack pointer 0, if the processor has been idle prior to the execution of T9. The very same holds for tasks T10, T11, and T12.

For the rather simplistic benchmarks from PapaBench, performing up to 79 timing analyses may be feasible, but this is not a scalable solution. Firstly, the range of initial stack pointers increases with the stack usages and the number of tasks in the system. Secondly, performing a timing analysis may take several hours or even days depending on the complexity of both the analyzed code and the target architecture.

C. Increased Predictability due to a known Stack Pointer

Next, we examine the increased predictability due to a single, statically known stack pointer as provided by EMPRESS. To this end, we derive each task's worst-case execution bound under two assumptions:

1) Static stack pointer (using the values from Table II), and
2) Range of possible stack pointers (using the range of values derived above).

The most important difference between these two is that the cache analysis has to conservatively assume a cache miss for each access to the stack in the second case. Consequently, the difference between both cases, fixed stack pointer and a range of stack pointers, mostly depend on the number of memory accesses to the cache, and the memory access time, i.e., the additional delay to acquire data from main memory instead of from the cache.

Our target architecture ARMv7 can be instantiated with different cache configurations. We have assumed a standard configuration with an instruction scratchpad, and a data cache of size of 2kB, with a 4-way LRU replacement policy, a line size of 16 bytes and 32 sets.

As the memory access time is most relevant to the results, we have performed the timing analyses assuming an access time of 10 and 20 cycles. To derive the worst-case execution time bounds, we have used Absint's Timing Profiler for ARM. In our first experiment, we performed the analysis with the known static stack pointer, and then assuming a range of values for the stack accesses. The results are shown in Table III.

	access time 10			**access time 20**		
	range	**static**	%	**range**	**static**	%
T1	3940	3484	13.09	4782	3744	27.72
T2	1092	1010	8.12	1530	1339	14.27
T3	2173	2014	7.89	2905	2527	14.96
T4	1374	1367	0.51	2064	2047	0.83
T5	755	598	26.25	1055	708	49.01
T6	609	513	18.72	849	633	34.12
T7	2086	1732	20.44	2755	1972	39.71
T8	127	127	0	177	177	0
T9	24636	18831	30.83	32227	19591	64.59
T10	5530	5158	7.21	6957	6145	13.21
T11	12372	10540	17.38	15638	11609	34.71
T12	6616	6519	1.49	8046	7819	2.9
T13	2320	1840	26.09	3149	2090	50.66

TABLE III
WORST-CASE EXECUTION TIME BOUNDS (IN CYCLES) FOR PAPABENCH, ASSUMING A MEMORY ACCESS TIME OF 10 AND 20 CYCLES, RESULTS FOR A RANGE OF STACK POINTERS (STANDARD STACK SHARING) AND FOR A KNOWN INITIAL STACK POINTER (PREDICTABLE STACK SHARING).

We can observe an increased imprecision of the execution time bound due to the uncertainty in the initial stack pointer, ranging from 0% to 64%, with most values in between 10% and 20% (average reduction 13.69% and 26.67%). The extent of the imprecision loosely correlates with the task's stack size and the number of stack accesses. For task T9 with the second highest stack need of 128 bytes, for instance, we see an increase in the worst-case execution time bound by 30% for a memory access time of 10 and by 65% for a memory access time of 20 cycles. In stark contrast, task T4, with a stack need of only 16 bytes exhibits only a minimal increase by less than 1%, and task T8, which does not perform any memory accesses to the stack, shows no difference at all. The task with the highest stack usage, T10 (188 Bytes) however, shows a relatively moderate increase of 7% and 13%, only.

The evaluation clearly shows that a statically known initial stack pointer is fundamental for a precise timing analysis and hence, for timing verification of the embedded system. To enable a precise timing analysis, we therefore either have to resort to dedicated per task stack spaces, at the cost of increased memory usage, or employ the predictable shared stack method EMPRESS.

We acknowledge that the results of the evaluation strongly depend on the selected system configuration and task sets. Fewer stack accesses, shorter memory access times, or slower instruction memories for instance, may lead to less accentuated differences between the execution time bounds with known or unknown initial stack pointers, although memory access times of 10 and 20 cycles are already on the low side. Nevertheless, we consider the impact of known initial stack pointers on the predictability and precision of timing analyses considerable. Note that we omit evaluation of the increased predictability of, for instance, the analysis of cache-related pre-emption delays or the optimization of the task layout. Due to the lack of techniques to handle a range of stack pointers instead of a single statically-known value, we consider predictability a binary property in these cases. CRPD analysis [2, 3, 7] for data caches have so far always implicitly assumed dedicated stack regions per task, and are now also feasible using the predictable shared stack – with the benefit of a reduced memory need for the stack.

VII. PREDICTABLE SHARED STACK SUPPORT PROVIDED BY ERIKA

Every possible implementation of a predictable shared stack needs to take into account the internal implementation of the context change mechanism and stack allocation of the specific RTOS in use. In this section, we will analyse two possible architectures of shared stack implementation, which have been implemented in two versions (v2 and v3) of the ERIKA Enterprise Kernel [13].

The main concept that is present in an RTOS allowing a shared stack is the absence of blocking primitives (this is the case of the *Basic Tasks* in the OSEK OS BCC1/BCC2 conformance classes, where blocking primitives are not present and mutexes are handled using the Immediate Priority ceiling Protocol [29]). When blocking primitives are not allowed, tasks execute under a run-to-completion semantics (where, once activated, a task can only be preempted or terminate).

A run-to-completion semantics opens the possibility to implement a shared stack paradigm inside the RTOS (this has been in fact implemented by both ERIKA Enterprise in *mono stack* configuration and by the RTOS included in the QP Framework [27]). Basically, tasks are *activated* every time they are needed, and when terminating they *clean up* automatically their stack (this is done automatically by C compilers in the epilogue of the C function call implementation); preemption is implemented by means of interrupt stack frames and by a scheduler function called at the end of the interrupt handler. As a side effect, this way of implementing tasks and preemption has the advantage of lower memory usage, because tasks do not need to save registers on synchronous context changes (those initiated by a primitive called by the running task) because callee-saved registers will be saved automatically by the compiler prologue of the task body C function. On the other hand, caller-saved registers will be saved on asynchronous events (interrupts) only.

In addition to a *pure* stack sharing implementation as described above, the OSEK/VDX standard also mandates two primitives named `TerminateTask` and `ChainTask`, which are responsible for stack unwinding at task termination. Basically the preempting task, before starting, saves the callee-saved registers and the current stack pointer thus allowing stack unwinding at its termination, in a way similar to the POSIX `longjmp` primitive.

An implementation of the predictable shared stack under the assumptions above requires an additional stack saving and stack pointer modification every time a task is executed, and a restore of the previous stack pointer at task termination. This, in ERIKA Enterprise v2, can be done as part of the implementation of `TerminateTask`, taking advantage of the pre-existing stack unwinding, thus incurring a minimal additional overhead only at task startup.

In addition to that, we note that a complete implementation of the OSEK/VDX standard requires the support of private stacks in order to handle *extended tasks* calling the `WaitEvent` synchronization primitive. In order to support that feature, the implementation proposed in ERIKA Enterprise v2 allowed the coexistence of shared stacks with private stacks, in a so-called *multi-stack* configuration. In this kind of configuration, each task is statically assigned to one of the available stacks. Stack sharing therefore happens by default when all basic tasks are grouped in the same stack by the configuration tool RT-Druid. Also the predictable shared stack implementation must be therefore limited only to the basic tasks.

The further need of optimization, joined with the additional requirements of the AUTOSAR OS memory protection led the developers of the new ERIKA v3 RTOS to create a new context change mechanism which, in a single implementation, is merging the mono-stack, the multi-stack, and the `TerminateTask` implementation described above. The main idea is that, while still maintaining the possibility to share a stack among tasks, at each context change both the callee-saved registers and the stack pointer are saved and considered belonging to the preempted task (in contrast, as noted above, to ERIKA v2 where this information was saved by the preempting task).

Notably, this new context change mechanism saves and restores the stack value at each context change. Therefore, from the point of view of the predictable stack sharing, this means that it is possible to implement the predictable stack sharing EMPRESS *at no additional overhead* compared to the current implementation of ERIKA v3, making the technique appealing for a future integration into the product.

On the applicability of the proposed method to other RTOSes, we can note that the proposed solution is very similar to the special case of per-task stacks, where the stack spaces are overlaid in memory. This means in general that the user needs to perform two actions. The first action is the modification of the stack space allocation done by either the RTOS or by the task configuration, in a way that the various stacks are overlaid in the proper way; typically the change is small if the stacks are somehow pre-allocated with static addresses, or if the implementation allows the specification of the stack space when creating a task, like it happens with POSIX pthread attributes. The second action is related to the need to guarantee that the RTOS tasks will always execute under a run-to-completion semantics, in order to avoid that persistent data will stay on the stack while the task is blocked. This second aspect is guaranteed by construction in ERIKA and in the QP RTOS (this because of the scheduling algorithm used and because the kernel does not provide blocking primitives at all), but needs special care with generic RTOSes. In particular, the run-to-completion behavior can be obtained by creating/destroyng tasks at each start/end of a job, together with the guarantee that these tasks will never block. This last guarantee is the most difficult to obtain, as there is the need to guarantee that no blocking primitives will be called by the task or by the library functions called by the task (this includes for example, on a POSIX system, all the cancellation points), and there is the need to guarantee that in no case there will be a preemption due to an arbitrary change of priority in the task (for example, because the round-robin timeslice has been consumed by the task, or becasue the task reduced its own priority). Therefore, although not impossible to implement on a generic RTOS, we believe that the proposed approach has to be carefully planned beforehand by the RTOS implementer.

VIII. Related Work

A shared stack system to reduce memory requirements, in particular for resource-constrained embedded systems, has been addressed in many papers [4, 10, 30, 14, 19, 6, 34, 21].

Already in 1991, Baker [4] described that a shared stack was used in real-time executives for at least 15 years, and proved that the Stack Resource Policy (SRP) facilitates sharing of stack space between jobs in a context with mutually exclusive resource sharing. Limited pre-emptive scheduling through non-preemptive groups [10] or fixed-priority scheduling with preemption thresholds (FPTS) [32, 30] enable a significant reduction in memory requirements by using a shared stack. Non-preemptive groups as well as FPTS distinguish two kinds of priorities for tasks, a (base) priority π for which a job of a task competes for the processor upon activation and a preemption threshold (or dispatch priority) θ that is at least equal to the (base) priority, i.e. $\pi \leq \theta$. A running task τ_j can only be preempted by a task τ_i when the priority of τ_i is higher than the preemption threshold of τ_j, i.e. $\pi_i > \theta_j$. In [9], it has been demonstrated that for given (base) priorities, there exists a unique maximum preemption threshold configuration for which a set of tasks is schedulable. Moreover, in [16] it has been shown that the maximum preemption threshold configuration requires the least amount of shared stack. An algorithm to determine the maximum threshold configuration for given priorities is presented in [30]. Algorithms to determine the maximum stack usage given a preemption graph have been described in [19, 6]. An extension to a multi-processor configuration is described in [14]. Finally, a heuristic approach to compute priorities for stack optimization is presented in [34].

Although AUTOSAR/OSEK [1, 29] supports FPTS, it only supports a restricted version based on so-called internal resources [20]. Unlike FPTS, there no longer exists a maximum threshold configuration for given priorities for this restricted version. How to determine the minimal stack usage for AUTOSAR/OSEK is described in [21]. It is worth mentioning that this restriction has been lifted in some specific implementations of the AUTOSAR standard.

To the best of our knowledge, the combination of a shared stack and predictability has not been addressed in the literature before, however, and is the topic of this paper.

IX. Conclusions

In this paper, we have presented an Efficient and effective Method for PREdictable Stack Sharing called EMPRESS. EMPRESS statically analyses the worst-case, i.e., highest, stack pointer of each each task, and guarantees these stack pointers during runtime. It therefore combines the reduced stack usage of a shared stack with the predictability of dedicated stack regions, and hence enables precise timing analysis and system optimizations.

We have demonstrated these benefits based on a case study of an unmanned aerial vehicle, PapaBench. EMPRESS provides a reduction in the total stack usage of up to 37% compared to an implementation using dedicated stacks per task, and the worst-case execution time bound could be reduced by up to 26% on average compared to a standard, i.e., non-predictable shared stack. The rather simple concept behind a predictable shared stack allows us to realize EMPRESS in existing Real-Time Operating Systems, such as Erika Enterprise, without additional overhead. Given its simplicity, ease of support in an existing RTOS, and clear advantages from a predictability perspective, we believe EMPRESS is a significant contribution, in particular from an industrial perspective.

Acknowledgements

The research presented in this paper is partially funded by the NWO Veni Project "The time is now: Timing Verification for Safety-Critical Multi-Cores", and by the Eurostars project E10171 RETINA.

References

[1] AUTOSAR release 4.3. 2016. Available: http://www.autosar.org.

[2] S. Altmeyer, R. I. Davis, and C. Maiza. Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In *Proc. 32nd IEEE International Real-Time Systems Symposium (RTSS)*, pages 261–271, Dec. 2011.

[3] S. Altmeyer and C. Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, August 2011.

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

[5] R. Barry. *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

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

[7] R.J. Bril, S. Altmeyer, M.M.H.P. van den Heuvel, R.I. Davis, and M. Behnam. Integrating cache-related preemption delays into analysis of fixed priority scheduling with pre-emption thresholds. In *Proc. 35th IEEE Real-Time Systems Symposium (RTSS)*, pages 161–172, Dec. 2014.

[8] 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.

[9] J. Chen, A. Harji, , and P. Buhr. Solution space for fixed-priority with preemption threshold. In *Proc. 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 385–394, March 2005.

[10] R.I. Davis, N. Merriam, and N.J. Tracey. How embedded applications using an RTOS can stay within on-chip memory limits. In *Proc. Work in Progress and Industrial Experience Session, 12th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 43–50, June 2000.

[11] C. Duffy, U. Roedig, J. Herbert, and C.J. Sreenan. Adding preemption to TinyOS. In *Proc. 4th Workshop on Embedded Networked Sensors (EmNets)*, pages 88–92, Jan. 2007.

[12] C. Ferdinand and R. Heckmann. aiT: worst case execution time prediction by static program analysis. In *Proc. International Federation for Information Processing (IFIP)*, volume 156, pages 377–384, Aug. 2004.

[13] 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.

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

[15] G. Gebhard and S. Altmeyer. Optimal task placement to improve cache performance. In *Proc. 7th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, pages 259–268, Oct. 2007.

[16] R. Ghattas and A. G. Dean. Preemption threshold scheduling: Stack optimality, enhancements and analysis. In *Proc. 13th Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 147–157, April 2007.

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

[18] S. Hahn and D. Grund. Relational cache analysis for static timing analysis. In *Proc. 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 102–111, July 2012.

[19] 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.

[20] L. Hatvani and R.J. Bril. Schedulability using native non-preemptive groups on an AUTOSAR/OSEK platform. In *Proc. 20th IEEE International Symposium on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2015.

[21] L. Hatvani and R.J. Bril. Minimizing stack usage for AUTOSAR/OSEKs restricted fixed-priority preemption threshold support. In *Proc. 11th IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2016.

[22] 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.

[23] 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.

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

[25] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *Proc. 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2015.

[26] W. Lunniss, S. Altmeyer, and R. I. Davis. Optimising task layout to increase schedulability via reduced cache related pre-emption delays. In *Proc. 20th ACM International Conference on Real-Time and Network Systems (RTNS)*, pages 161–170, Oct. 2012.

[27] Quantum Leaps Miro Samek. http://www.state-machine.com.

[28] 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.

[29] OSEK group. OSEK/VDX operating system. Technical report, February 2005. OSEK OS 2.2.3 available as ISO Standard 17356-3 2005.

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

[31] Arcticus Systems. http://www.arcticus-systems.se.

[32] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with pre-emption threshold. In *RTCSA*, pages 328–38, Aug. 1999.

[33] 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*, 7(3):36:1–36:53, May 2008.

[34] H. Zeng, M. D. Natale, and Q. Zhu. Minimizing stack and communication memory usage in real-time embedded applications. *ACM Transactions on Embedded Computing Systems*, 13(5s):149:1–149:25, July 2014.