# Semi-Extended Tasks: Efficient Stack Sharing Among Blocking Threads

Christian Dietrich
Leibniz Universität Hannover, Germany
dietrich@sra.uni-hannover.de

Daniel Lohmann
Leibniz Universität Hannover, Germany
lohmann@sra.uni-hannover.de

***Abstract*—Memory is an expensive and, therefore, limited resource in deeply embedded real-time systems. Thread stacks substantially contribute to the RAM requirements. To reduce the system's *worst-case stack consumption (WCSC)*, it is state of the art to exploit thread-level preemption constraints to let multiple threads share the same stack.**

**However, deriving a tight, yet correct bound for the shared stack is a difficult undertaking and stack sharing is currently restricted to run-to-completion threads, which are preemptable, but cannot block (i.e., passively wait for an event) at run time.**

**With *semi-extended tasks (SETs)*, we propose a solution for efficient stack sharing among blocking *and* non-blocking threads on the system level.**

**For this, we refine the stack-sharing granularity from the thread to function level. We provide an efficient intra-thread stack-switch mechanism and an ILP-based WCSC analysis that considers fine-grained preemption constraints and possible function-level switching points from the private to the shared stack. A genetic algorithm then selects switching points that lead to the reduction of the overall WCSC. Compared to systems that run only non-blocking threads on the shared stack, semi-extended tasks decrease the WCSC in our benchmarks on average by 7 percent and up to 52 percent for some systems.**

***Index Terms*—Real-time operating systems, Static analysis, Worst-Case Stack Consumption**

## I. Introduction

Memory is a scarce resource in embedded control systems. The RAM footprint is a significant factor for per-unit costs in products of mass production, such as sensor network nodes [8], IoT devices, or automotive control units [4] running a *real-time operating system (RTOS)*. Especially for MCU-based systems, even a small increase in the RAM footprint can have significant effects on the per-unit costs: if the device's memory overflows, the designer often has to purchase the next larger MCU with a doubled amount of RAM.

As a consequence, operating systems for these domains try to avoid (or at least discourage) the employment of fully-fledged threads, because each thread instance requires the provisioning of a private stack in RAM, dimensioned for its worst-case call depth (plus potential interrupts). Operating systems for small sensor nodes, such as Contiki [8] or TinyOS [19] have long favored a thread-less event-driven programming model without preemption (restrictions that partly were relaxed later by optional thread packages [9, 16]), but the issue of (too) high stack costs by threads was also observed and mitigated in big systems like Mach 3 [7].

### A. Preemptable vs. Blocking Threads

In real-time systems, giving up preemptability at all is usually not an option. Here, the common technique to reduce the overall *worst-case stack consumption (WCSC)* is to co-locate different threads onto a single shared stack [1, 31, 33]. In the OSEK/AUTOSAR standard [23], for example, such threads are called *basic tasks (BTs)*.

On the shared stack, which we call the *basic-task stack (BTS)*, function-call frames from different threads are compactly stacked over each other. Given there are preemption constraints, so that some of these co-located threads cannot be active at the same time, the size of the BTS can be dimensioned smaller than the sum of the WCSCs of each individual thread. For example, if two threads running on the BTS share an implicit resource (thus, cannot preempt each other), then only frames from one of the two threads could be active at the same time.

However, albeit preemptable by higher-priority threads, BTs are not fully-fledged threads: They are not allowed to block (e.g., wait passively for an event). Instead, they always have to execute in a run-to-completion manner. This restriction prohibits a collision on the stack when a lower-priority thread gets scheduled and, thus, uses stack space beneath a blocking higher-priority thread that eventually wakes up and needs more stack space. Hence, not all threads can be executed on the BTS. If they need to block, they have to be executed on their own, private stack. OSEK/AUTOSAR [23], for instance, provides an optional abstraction for real threads, which are called *extended tasks (ETs)*.

To sum up: *basic tasks (BTs)* are preemptable, but must not block at run time. The RTOS can exploit preemption constraints to dispatch them more memory-efficient on a single shared stack, the *basic-task stack (BTS)*. In contrast, *extended tasks (ETs)* may block at run time, but have to be dispatched on costly private stacks.

### B. Deriving A Tight Worst-Case Stack Consumption (WCSC)

In the automotive industry, the distinction between non-blocking and blocking threads on the RTOS level is considered as a major success factor of OSEK OS: The BT concept has made it possible to use an RTOS even with very memory-constrained control units. The downside is increased complexity for the developers, who have to understand and define which threads belong to which class and ultimately have to specify the size of the shared BTS. Deriving a tight WCSC for the

BTS is not trivial: All preemption constraints have to be considered in the calculation, which do not only stem from the task configuration but also the implementation itself (e.g., due to mutual exclusion by locks). Furthermore, ETs – even if restricted to only those threads that need to block – still waste stack space in most cases: In principle, the private stack is only required for the blocking, whereas leaf functions may well be executed on the shared stack. Experienced developers manually split such threads into an ET and a dependent BT, which, however, is again not trivial: It further increases complexity and only decreases the overall WCSC, if the right split points are chosen.

### C. About This Paper

We lift the manual and coarse-grained distinction between non-blocking and blocking threads by introducing the notion of *semi-extended tasks (SETs)*. Conceptually, a SET starts as an ET on its own private stack (where it can call functions and issue blocking system calls) but switches to the BTS whenever this is possible *and* beneficial. We derive these switching points on the function level and achieve a tight WCSC by the first flow-sensitive and RTOS-aware preemption analysis on the task and function level. In our extensive evaluation, SETs reduce the WCSC (compared to systems that support only BTs and ET) by up to fifty percent. In particular, we claim the following original contributions:

1) The SET concept, enabled by an efficient intra-thread stack switching mechanism that transfers individual functions and their children onto the shared stack (Section III).
2) An *integer linear programming (ILP)*-based, worst-case stack-consumption analysis that considers fine-grained preemption information and supports semi-extended tasks (Section IV).
3) The extraction of fine-grained task–to-task and function–to-task preemption relations from the flow-sensitive global control-flow graph (Section V).
4) A heuristic to find optimal switch points at which threads transfer their execution to a shared stack, even if they wait at some other point (Section VI).
5) An extensive evaluation of the achievable stack savings using generated non-trivial task sets with dependencies, synchronization, IRQs, and sharing (Section VII).

In the remaining parts of the paper, we start with our system model in Section II before we describe the above main contributions in individual sections. We discuss the results in Section VIII, give an overview of the related work in Section IX, and finally conclude our article in Section X.

## II. System Model

We consider event-triggered real-time systems with fixed-priority, mixed-preemptive scheduling of a fixed set of tasks. We support single-core or partitioned multi-core real-time systems. Tasks, which are already mapped onto threads, are the objective of scheduling and their functionality is available in terms of their control-flow graph (e.g., source code). For the rest of the paper, we use the terms "task" and "threads" interchangeably, as we mean the technical and already materialized entities that are scheduled and dispatched by an RTOS implementation. For each thread, the scheduling priority, the preemption threshold [30], and the entry function is known. Furthermore, we know the whole call graph, beginning at these entry functions, and have an upper bound for the stack consumption of each individual function (stackusage(f)). Threads use system calls to interact with the RTOS and system-call sites are known in their source-code location and their arguments. Threads use system calls to activate each other, enter a waiting state, wake up another thread from the waiting state, access shared resources via the *stack resource policy (SRP)* [1], block interrupts, and terminate themselves. At every moment, at most one task instance (job) is actively running and if that instance terminates, no residual data is left on the stack. We assume that interrupts can also activate threads, but are handled on a separate stack. A prominent RTOS standard that is compatible with our system model is OSEK/AUTOSAR OS [23].

## III. Semi-Extended Tasks (SETs)

In the following, we introduce the concept of *semi-extended tasks (SETs)*, which are a generalization of the distinction between non-blocking BTs and blocking ETs. Basically, a SET can enter the blocking state, but its jobs can also execute (partially) on the BTS.

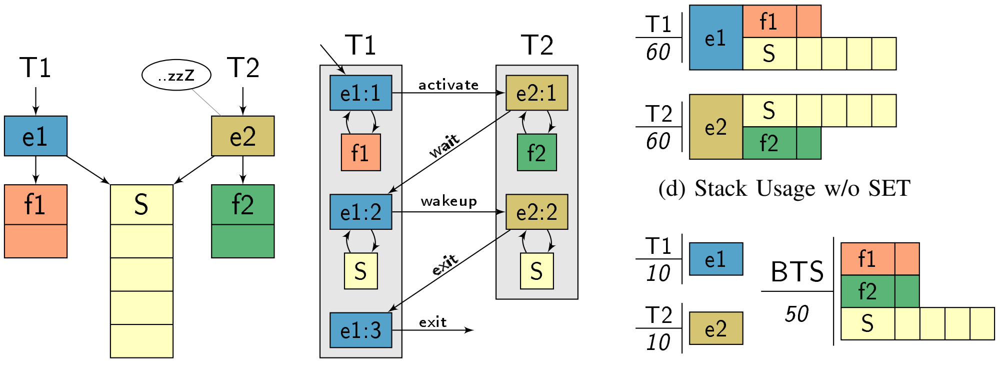
### A. SETs and Switch Functions

Conceptually, a SET is an ET with its own private stack on which it can call functions and issue blocking system calls. However, for the execution of some (sub-)function (and its children), the thread can switch to the BTS if no blocking system call is issued (directly or indirectly via its children) inside the function. We call functions that switch to the BTS *switch functions*. If the thread's entry function is a switch function, we can omit the private stack completely and the SET becomes a BT. If there is no switch function in the call graph of the thread, the SET runs only on its private stack and becomes an ET.

Figure 1 shows a situation where the usage of SETs outperforms the normal BT/ET concept, which places only whole threads on the BTS. Without SETs, both threads execute on their own private stack and have a combined WCSC of 120 bytes (see Figure 1d). Even when using a BTS, only T1 can become a BT since T2 issues a blocking system call, leaving the stack size at 120 bytes. However, the leaf functions `f1()`, `f2()`, and `S()` cannot wait and, therefore, can be executed on the shared BTS (see Figure 1e). Given a flow-sensitive preemption analysis (see Section V), we know that only one of these three functions can be active at any point in time and, therefore, the maximal stack consumption on the BTS becomes 50 bytes. With the now smaller private stacks for T1 and T2 (10 bytes each) the overall WCSC of this system is reduced to 70 bytes.

```
// T1.autostart = true
// T2.prio > T1.prio
Task(T1, e1) { // 10 bytes
    f1();      // 20 bytes
    activate(T2);
    S();       // 50 bytes
    wakeup(T2);
}
Task(T2, e2) { // 10 bytes
    f2();      // 20 bytes
    wait();
    S();       // 50 bytes
}
```

(a) Example System



(b) Call Graph (c) GCFG (d) Stack Usage w/o SET (e) Stack Usage w/ SET

Fig. 1: Example System with two tasks (T1, T2) and their corresponding entry functions (`e1()`, `e2()`) that interact via the RTOS and call leave functions (`f1()`, `f2()`, `S()`). The call graph indicates the caller-callee relationship, as well as the functions respective stack usage (1 box=10 bytes). The *global control-flow graph (GCFG)* results from a flow-sensitive RTOS–application interaction analysis (see [6]) and indicates all possible system-wide control flows including preemptions. The entry functions are split into several nodes (e.g., e2:1, e2:2). The worst-case stack usage is depicted for a system without support for SETs (120 bytes) and for one system with SET support (70 bytes) where the functions `f1()`, `f2()`, and `S()` switch to the shared BTS. When functions are drawn vertically above each other, they cannot be active at the same time (e.g., f1,f2,S on the BTS in (e)).

## *B. Implementation of Switch Functions*

In the following, we explain how we modify single functions to switch with minimal effort to the shared stack, as well as the required compiler and RTOS modifications. In order to understand the stack-switching mechanism, we shortly have to explain a few terms that are related to call frames as they are handled by compiler-generated code. Without loss of generality, we use IA-32 as an example, since calling conventions on other architectures share similar concepts.

During thread execution, each function call has a *call frame* on the stack, which contains the passed parameters, local variables, and return address. This call frame is addressed by two pointers that live, for the currently active frame, in registers: The stack pointer `esp` points to the end of the frame; child functions use it as starting point for their own frame. The (frame) base pointer `ebp` points to the return address and argument block and is used to access the arguments. Hence, local variables and callee-saved values can be accessed by the compiler relative to either the stack pointer or the base pointer.[1]

In order to implement switch functions, we subtly constrain the access patterns employed by the compiler (in our case LLVM), so that local variables are only accessed via the stack pointer, while arguments and caller-saved values are only accessed via the base pointer. By separating these access paths, we can split the call frame between arguments and local variables and place it at two different memory locations (i.e., the private stack and the BTS). To perform the actual switch, we furthermore add a single instruction to the function prologue to set the stack pointer to the top of the BTS. The current value for the top of the BTS is stored in the variable `TOS_BTS`, which is maintained by the RTOS.

[1]A third register, `esi`, is used if the function employs `alloca()` or C99 variable-sized arrays.

Figure 2 shows the disassembly of a switch function for IA-32: The standard function prologue saves the base pointer of the caller as a callee-saved register (line 3) and sets up its own base pointer (line 4). We introduce a single additional instruction that loads the value of `TOS_BTS` into the stack pointer and, thereby, switches to the BTS (line 5 and bold edge in Figure 2b). Afterwards, the frame for this function is split between private (arguments) and shared stack (local variables). However, all functions that are called from switch functions directly execute on the shared stack without any further modification (e.g., `bar()`). When the function returns, the standard function epilogue implicitly also switches back to the private stack by restoring the original stack pointer (line 15) from the base pointer. It then restores, as usual, the callee-saved frame pointer (line 16), and returns to the caller (line 17).

Hence, with only one additional instruction (line 5) and minimal changes to the compiler's code generation, we can provide for very efficient switch functions that do not need to activate the operating system. However, as a switch function unconditionally jumps to the shared stack, we have to ensure in our analysis (Section VI) that no child function of a switch function is a switch function itself. Otherwise, line 5 would set the frame pointer back to the value of the first stack switch.

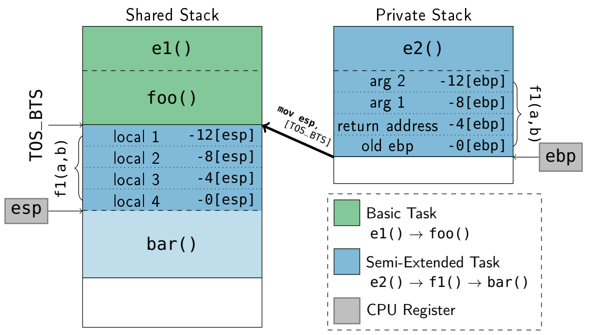
Besides the switch functions, we also have to modify the RTOS to provide the `TOS_BTS` variable. When the RTOS is left and an application thread is resumed, this variable must always point to the end of the *used* BTS area. Whenever a thread is preempted on the BTS it is increased; whenever a thread terminates on the BTS it is decreased. With these modifications, we support a mixed-usage of the BTS where BTs and SETs coexist and stacked on top of each other. Note that for the stack switch in user mode, `TOS_BTS` only needs to be read. It is only modified inside the kernel, so the kernels protection integrity can be kept. We further discuss the topic of memory protection with our model in Section VIII.

```
<f1>:
    ;; Function - Prologue
    push   ebp             ; Save old framepointer
    mov    ebp, esp        ; Load new framepointer
    mov    esp, [TOS_BTS]  ; Switch to shared stack
    sub    esp, 16         ; Allocate local variables

    ;; Function Body
    ;; - Access local variables via esp
    ;; - Access parameters via ebp
    ;; - Access local variable-sized arrays (alloca)
    ;;   via a third stack pointer esi, if needed

    ;; Function Epilogue
    mov    esp, ebp        ; Restore old stackpointer
    pop    ebp             ; Restore old framepointer
    ret
```

(a) IA-32 Disassembly of `f1()`



(b) Stack Diagram

Fig. 2: Implementation of switch functions. `f1()` switches from the private stack to the shared stack, by loading `TOS_BTS` into `esp`. The stack diagram shows a situation where the SET preempted the basic task and switched with `f1()` to the BTS.

## IV. Stack Consumption on a Shared Stack

In order to dimension the shared stack, we have to assess the total *worst-case stack consumption (WCSC)* for all threads and their functions running on the BTS (BT+SET). This assessment has to be done in a single combined step since we can only tighten the static BTS size if we can show that two function-call frames cannot be simultaneously active at any point in time. Therefore, this assessment has to consider the call graph and information about (forbidden and impossible) preemptions.

Several works [15, 3, 5, 29, 28] proposed methods to give a safe upper bound for the WCSC of several threads executing on the same stack. However, as we want to support recursive call-graph structures and the usage of fine-grained constraints on possible preemptions, we present a new WCSC analysis that utilizes an ILP solver. In contrast to the ILP construction presented by Wang et al. [29], we use the *implicit path-enumeration technique (IPET)*.

Similar to the usage of IPET for the worst-case execution time analysis [20, 24], we calculate the WCSC by finding the number of thread activations and function activations that occur during the *costliest call chain* and the *costliest preemption chain*.

We formulate a maximization problem and start out by introducing one binary variable $T_i$ for every possible thread on the BTS. If a thread is active in the costliest preemption chain, its frequency $T_i$ is 1. Hereby, we can already formulate mutual-exclusive constraints. For example, if the threads $\{T_l, \ldots, T_m\}$ share an implicit SRP resource, only one thread can be active in the WCSC case. Hence, we only one thread frequency can be 1:

$$\sum_{i=l}^{m} T_i = 1$$

To find the costliest call chain, we formulate a subproblem that captures the reachable call graph for each thread. First, we select all functions in the entry set $F_E$ that are starting points for this thread's BTS consumption. For basic tasks, this includes only the entry function; for semi-extended tasks, this includes all switch functions. From this initial set, we recursively include all called functions into the set of reachable functions $F$ and record all leaf functions in $F_L$.

Scoped and prefixed with $T_i$, we introduce integer-valued ILP variables: for every function $f$ ($T_i[f]$), for every call edge between $f_k$ and $f_l$ ($T_i[f_k, f_l]$), and one artificial edge variable for every entry function $f$ ($T_i[E, f]$). Due to the scoping, several function variables can exist for one function $f$ in different context ($T_1[f]$ vs. $T_2[f]$). These variables capture the activation frequencies for the WCSC case and we use them to formulate a maximum-flow subproblem for each thread. If a thread is part of costliest preemption chain, one of its entry functions is called:

$$T_i = \sum_{f \in F_E} T_i[E, f]$$

We constraint the function frequencies with the sum of their incoming call edges. For entry functions, these are the artificial entry edges and the normal call edges; for the other functions, only its callers are relevant:

$$\begin{aligned} \forall f \in F_E: \quad T_i[f] &= \sum_{f_s \in \text{caller}(f)} T_i[f_s, f] \quad + T_i[E, f] \\ \forall f \in F \setminus F_E: \quad T_i[f] &= \sum_{f_s \in \text{caller}(f)} T_i[f_s, f] \end{aligned}$$

As the call chain is one path through the call graph, every single function activation can lead to at most one function call. Hence, the sum of outgoing call edges is less or equal to the function frequency of the caller:

$$\forall f \in F \setminus F_L : \quad \sum_{f_d \in \text{callee}(f)} T_i[f, f_d] \quad \leq T_i[f]$$

Furthermore, we add recursion limits, which are supplied by the developer, to the ILP like loop bounds are added in WCET problems. At last, we add an optimization objective that connects the function frequencies to the stack consumption:

$$\max \left( \sum_{T_i} \sum_{f \in F_{T_i}} T_i[f] \cdot \text{stackusage}(f) \right)$$

With this ILP formulation, we can add more fine-grained constraints about impossible preemptions. For example, we can forbid that a thread $T_p$ can preempt a thread $T_i$ if the latter currently executes the function $f$ (see Figure 1c, T2 cannot preempt T1 in `f1()`). With the big-M-method [13], where M is sufficiently large, and an intermediate binary variable $x$, we formulate the constraint: If function $f$ is active in the context of $T_i$, $x$ must be 1. In that case $(1 - x)$ is zero and $T_p$ must become zero.

$$x \in [0, 1] \qquad T_i[f] \leq x \cdot M \qquad T_p \leq (1 - x)$$

In order to determine the WCSC for the whole system, we solve the ILP, calculate the WCSC for all private stacks, and sum up private and shared stack usage. However, for the private stacks, we filter all functions from the call graph that surely run only on the BTS, as they can no longer extend the costliest call chain. Summarized, we calculate the WCSC for a system with a set of basic tasks and a set of switch functions while taking fine-grained preemption knowledge into account.

## V. Flow-Sensitive Preemption Analysis

For a tighter WCSC analysis (Section IV), which will reveal the benefit of the SET approach, we have to formulate constraints about impossible preemptions and mutual-exclusive paths in the system. For this kind of WCSC analysis, it is state-of-the-art to use knowledge about priorities, non-preemptability, and preemption thresholds [30, 11, 29, 31]. In contrast, our analysis works on the granularity of function frames; hence, we can incorporate (and benefit from) more fine-grained knowledge.

In previous work [6], we successfully analyzed the interaction between the application and the RTOS for OSEK-like systems in a *flow-sensitive* manner. This analysis brings together application structure, RTOS configuration, and RTOS semantics in order to calculate a *global control-flow graph (GCFG)* for the whole system. The GCFG captures all possible control-flow transfers between code blocks within a thread's execution and all possible context switches between threads. The captured transitions also include all context switches that are triggered within asynchronous interrupts. For the work at hand, we use the polynomial fixpoint algorithm described in Dietrich, Hoffmann, and Lohmann [6] to calculate the GCFG. For detailed information about the analysis, we refer you to that article.

In order to illustrate the results of the interaction analysis, Figure 1c shows the GCFG for our example system (Figure 1a). In this (simplified) graph, the entry functions `e1()` and `e2()` are split into parts (at system-call boundaries) to make the analysis flow sensitive. The functions `f1()`, `f2()`, and `S()` are shown as one unit of execution. As T1 automatically starts at boot time and starts with the execution of e1:1. In this block, we either execute `f1()` or get preempted by T2, because we synchronously activated this higher priority thread. The execution flow continues to alternate between T1 and T2 until both threads terminate. As the GCFG includes *all* preemptions, we know for sure, that T2 can preempt T1 in general. However, as there is no direct edge between `f1()` (or `S()`) and any block from the T2 context, we know that T2 cannot preempt T1 in `f1()`.

More formally, the GCFG is a graph with 3-tuples $(T, f, i)$ as nodes and edges, which are optionally labeled with a system call, between them. Every node represents a code block executed and is identified by the currently running thread $T$, the currently executing function $f$, and which function part $f_i$ is currently active. If there is an edge between two nodes, the respective code blocks can be executed directly after each other, induced either by intra-thread control flow or by a context switch. In order to determine if a preemption (directly or indirectly) between the threads $T_i$ and $T_j$ is forbidden, we check, with depth-first search, that there is no path matching the pattern:

$$(T_i, *, *) \xrightarrow{*} (T_x, *, *) \rightarrow (T_j, *, *) \qquad T_x \neq T_i$$

If no path matches this pattern, we forbid the task-to-task preemption in our ILP (Section IV). Similar, we forbid a function-to-task preemption if a thread $T_i$ can never be preempted in function $f$ such that $T_j$ is reachable:

$$(T_i, f, *) \xrightarrow{*} (T_x, *, *) \rightarrow (T_j, *, *) \qquad T_x \neq T_i$$

As the GCFG also covers the semantics of SRP resources, non-preemptability, and interrupt blockades, we automatically consider these mechanisms in our WCSC analysis. For example, if a function is only executed with a higher dynamic priority (i.e., due to an SRP resource), we automatically include the respective preemption constraint.

To the best of our knowledge, our derivation of preemption constraints is the first flow-sensitive, RTOS-aware preemption analysis that considers system-call order, application-logic (e.g., conditionals), and RTOS semantics.

## VI. Selection of Stack-Switch Functions

As already explained in Section III, the proposed stack-switch mechanism brings the function unconditionally onto the BTS. While this makes the stack switching very efficient, an execution flow that is already on the BTS must not switch stacks again. Therefore, it is essential that no child of a switch function switches itself. However, the optimal selection of switch functions under this constraint is not trivial: For example,

| Dimension | Description | Range |
|---|---|---|
| #threads | RTOS Threads | [20, 50] |
| #IRQs | External, asynchronous thread activations | [1, 10] |
| #waiting | Number/Ratio of blocking threads | [0, 15] |
| #functions | Number of functions in the call graph | [100, 1000] |
| #resources | SRP resource groups with $> 2$ threads | [1, 10] |

TABLE I: Dimensions of the Synthetic Benchmarks

in Figure 1b it is not optimal to greedily select the top-most functions that could act as switch functions (`e1()`, `f2()`). The T2 stack would still use 60 bytes (`e2() +S()`) and the BTS 60 bytes (`e1() +S()`).

It would be desirable to formulate the WCSC-ILP and the switch-function selection in a single ILP. However, as the IPET formulates a maximization problem and the switch-function selection is a minimization problem over the same variables, the combination is a bilevel optimization problem, which cannot be formulated easily as ILP. Instead, we use a genetic algorithm as a heuristic to determine the set of switch functions for a given system.

We start out by defining the genome of our problem as a bit string with one bit for every function that is able to switch to the BTS (i.e., does not wait). If the bit is 1, the function becomes a switch-function. By this representation, we also can decide whether it is more beneficial to mark a whole thread as a basic task: When a thread's entry function is a switch function, we make the thread to a basic task and remove the entry function from the switch-function set. As fitness function for a switch-function set, we formulate and evaluate the ILP (Section III) and calculate the WCSC for the whole system.

As heuristic parameters, we used a population size of 20 individuals. In every generation, we breed 6 new individuals: With 5 percent probability, we select only one parent and mutate a single bit; with 95 percent probability, we select two parents and generate the child by the crossover operation. As this can result in invalid genomes (i.e., child and parent function switch), we repeat the breeding until 6 valid individuals are found. After evaluating the fitness function, we select the 20 individuals with the lowest stack consumption for the next generation. To avoid calculating the WCSC twice, we use a cache to store genome–WCSC pairs. We stop the genetic algorithm, when progress is done in 1000 generations or 60 seconds.

## VII. Evaluation

In order to assess the improvements in stack-space savings induced by the usage of SETs, we generate synthetic benchmarks with varying characteristics. We compare the benefits of having basic-task support and semi-extended–task support against a system that supports only private stacks.

### A. Benchmark Generation

Each of our synthetically generated systems is characterized by five different parameters (Table I): #threads, #IRQS, #waiting, #functions, #resources. We use a pseudo-number generator with varying seeds to get different, but reproducible results. We start out by generating a directed, acyclic thread-dependency graph with #threads nodes. #waiting of these nodes get more than one predecessor node and will not only be activated by a predecessor but also wait for other predecessors to wake them up at some point in their execution. Note that a system will always have exactly #threads threads. #IRQs of the nodes are additionally activated by an external interrupt. We shuffle the threads priorities such that every thread has its own priority and mark ten percent of the threads as non-preemptible. Furthermore, we form #resources SRP resource groups with at least two threads.

For the call graph, we first form randomly an n-rooted forest with #threads roots and #function nodes. In order to form a more complex call-graph, we add 20 cross-tree call edges which also results in the sharing of functions between threads. For every function, we uniformly choose a stack consumption between 90 and 120 bytes. For every thread, we distribute its system calls for thread activations, wait-for and wakeup operations, IRQ-blockade sections ($0.1\cdot$ #threads), and SRP-protected critical sections into the first four functions beginning at the entry function.

The restriction on the placement of system calls stems from the current implementation of the GCFG, which system-call–invoking shared functions. Nevertheless, to ensure a minimal call-graph complexity, we choose the worst function/thread ratio (200/50) as the number of system-call–invoking functions.

After generation, we dump the system configuration as an OSEK configuration file and materialize the call graph into an OSEK-compatible C source-code file. These two files are the input to our OSEK system generator that implements the proposed analysis method.

### B. Evaluation Scenario

In order to quantify the impact of different system parameters on the SET savings, we choose the configuration (#threads=20, #IRQS=1, #waiting=10, #functions=200, #resources=1) as our base parameter class. From this parameter class, we explore each of the five dimensions independently (see Table I). For example, Figure 4a explores the influence of #IRQs and shows the results for the parameter range (20,**1**,10,200,1) up to (20,**10**,10,200,1). For every parameter class, we generated 300 synthetic systems. In total, we analyzed 14 700 systems.

For every generated system, we examine three variants: As baseline variant, we use an RTOS configuration that only supports ETs on private stacks. The BTS-system variant greedily marks all non-blocking threads as basic tasks and co-locates them on the shared stack. The SET-system variant optimizes the decision whether a thread should become a basic task or where switch functions should be used according to the method described in Section VI. In all three variants, the stack consumption is calculated according to Section IV using the same fine-grained preemption constraints (Section V).

### C. Evaluation Method

We integrated our analysis and optimization approach into the dOSEK RTOS generator [14], which is written in Python.

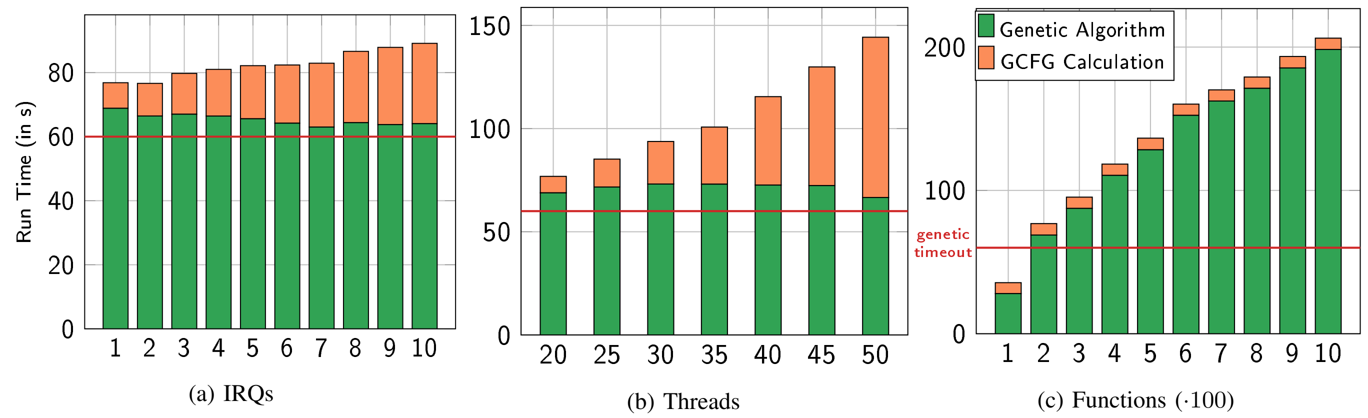


Fig. 3: Run-time for preemption analysis (GCFG calculation) and for finding a good assignment of stack switching functions (Genetic Algorithm). Every data point is the average of 300 synthetic systems.

dOSEK reads in the system configuration and the application logic and performs the system-state flow analysis [6] to calculate the GCFG. In order to get the actual stack usage, we use the LLVM code-generation backend [18] for Intel IA-32 to extract the size of the stack frame for every function. Furthermore, we use Gurobi 8.0 in the default configuration as our ILP solver. The whole evaluation is run on an Intel i5-6400 quad-core system with 32 GiB of main memory. However, main memory was at no point a limiting factor and the optimization was CPU bound.

In order to implement SETs, we introduced the `TOS_BTS` variable in dOSEK and adapted the LLVM backend to support switch functions. For the LLVM modification, we had to change 35 lines of code in the IA-32 code-generation backend.

### D. Results

*1) Run Time and Scalability:* First, we take a look at the run time of the system analysis and the optimization step, especially since it repeatedly calls an ILP solver to evaluate the fitness of individual system configurations. For all of our synthetic systems, the genetic algorithm ran, on average, for 81 seconds and invoked the ILP solver 2672 times. However, as we set the timeout for the genetic algorithm to 60 seconds of no improvement, we see that most of the time is spent for further exploration of the solution space after the final result is already found.

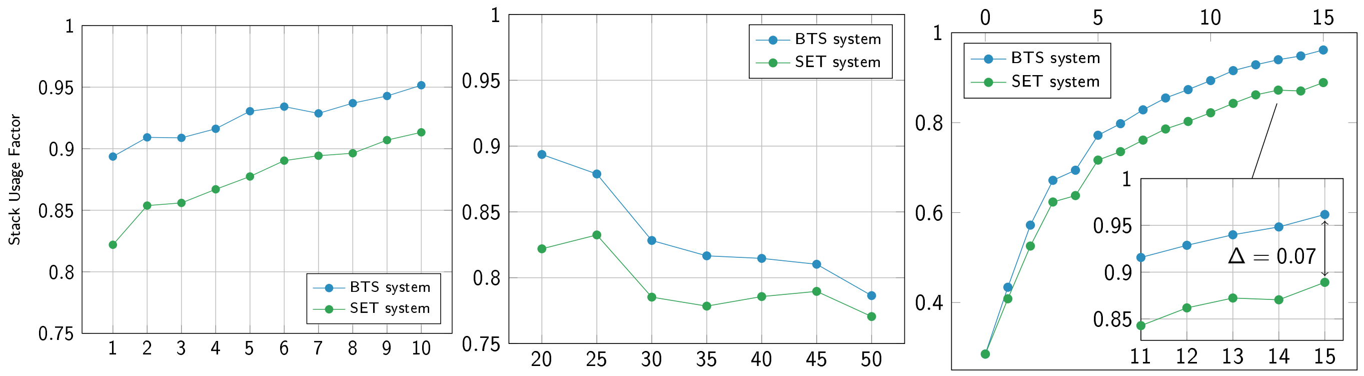
In Figure 3, we show the results for three most interesting dimensions for the run-time and scalability consideration. The results for the other dimensions can be found in the extension Figure 6. We separated the time for the GCFG calculation, which is required to extract fine-grained preemption information, from the run time of the genetic algorithm. Furthermore, the 60 seconds timeout is indicated by a red horizontal line. We can observe, that the GCFG analysis takes more time, when we increase the number of IRQs (Figure 3a), as the GCFG contains more inter-thread edges and becomes denser. Furthermore, we see that the run time of the genetic algorithm slightly drops, as fewer preemption constraints are included into the ILP as more preemptions, induced by interrupts, are possible. When we increase the number of threads (Figure 3b), we also see an increased time spent for the GCFG analysis but no significant change in the run time of the genetic algorithm.

The most impacting system parameter is the number of functions (Figure 3c). As it is directly reflected in the number of ILP variables, we see a significant run-time increase in the ILP solving. The time used for a single ILP invocation increases from 0.01 seconds for 100 functions to 1.02 seconds for 1000 functions, with a doubling of the run time for every 145 additional functions. This prolonged ILP-solve time resulted in a less intense exploration of the result space and the number of ILP invocations dropped (2695 $\rightarrow$ 584).

*2) Stack-Space Saving:* As the optimization step is done offline, before the run time, the most important quantity to evaluate the benefits of SETs is the stack-space saving factor. We take the ET-only system as the baseline and give factors relative to this baseline (lower is better). For every parameter class, we use the geometric mean for calculating averages. For SET systems, the average stack consumption over all benchmarks goes down to 0.78, while BTS systems achieve only a factor of 0.83.

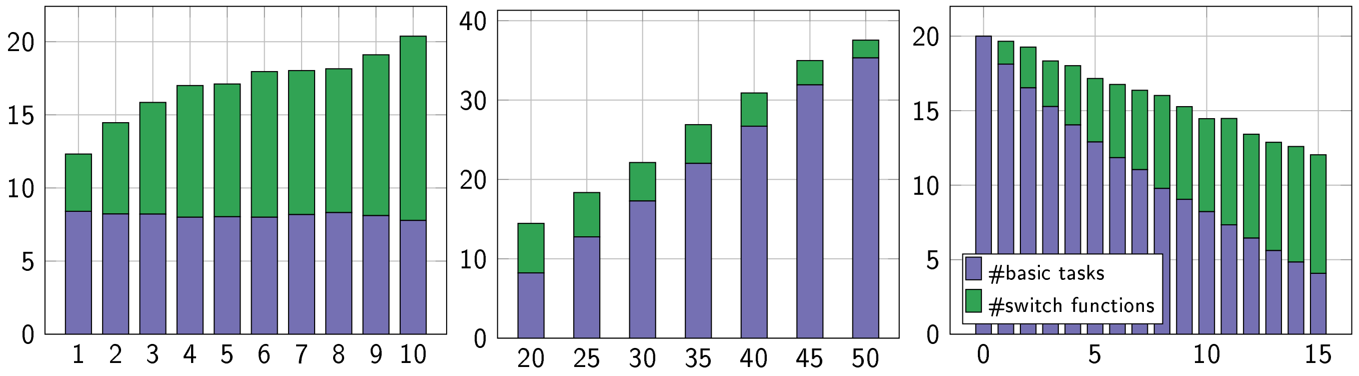
In Figure 4, we show the results for three dimensions, while the other, less interesting, dimensions are shown in Figure 6. As a general trend, we see that more IRQs (Figure 4a) impairs stack saving, as more IRQ-induced preemptions between threads are possible. However, the distance between both lines keeps relatively stable.

When we increase the number of threads (Figure 4b), both variants take advantage of this and drop under a factor of 0.8. However, the gap between both methods keeps converging with a rising number of threads. We can explain this trends when we look at the number of basic tasks and the number of switch functions for each benchmark for the SET systems. We see that the importance of switch functions decreases with a rising number of threads (Figure 5b). This stems from the fact that the complexity of the thread's call graph decreases, as we keep the number of functions constant. Thereby, it is



(a) IRQs (b) Threads (c) Waiting Threads

Fig. 4: Whole-system stack-space saving factor. The baseline (factor=1) is a system that supports only private stacks. We compare a system that supports only whole threads to be executed on the shared stack to a SET enabled system. Every data point is the geometric mean of 300 synthetic systems. *Lower is better*.



(a) Functions (·100) (b) Threads (c) Waiting Threads

Fig. 5: Number of basic tasks and switch functions for SET systems. Every data point is the average of 300 synthetic systems.

more often beneficial to mark a whole thread as a basic task.

The most interesting trend can be found when we increase the ratio of waiting threads (Figure 4c). When no thread waits, both methods equally mark all threads as basic tasks and we get a factor of 0.29. This situation is well known from the literature about stack sharing on the thread granularity. However, when we start to mark threads as waiting, these threads are no longer able to utilize the shared stack and benefits melt down quickly. Although SETs cannot stop this decline completely, the benefits remain larger with a growing gap between both methods ($\Delta_{max} = 0.07$). Again, we understand this trend, when we look at the number of basic tasks and switch functions for the SET system in Figure 5c. Although the number of basic tasks drops with more waiting threads, the drop is partially absorbed by an increasing number of switch functions.

Furthermore, we see in Figure 5b that the addition of functions makes it necessary to mark more functions as switch functions to achieve an equal stack-space factor. This stems from the fact that the reachable call graph of each thread is potentially wider as it includes more functions on average. Overall systems, we marked on average 11 threads as basic tasks and 6 functions as switch functions. For the BTS systems, 13 threads were marked as basic tasks.

To give a better impression of the potential of the SET approach, we also looked at individual results of single systems. Our best benefit over a BTS system was for an element of the class (20,1,10,600,1): The SET system achieved a factor of 0.45, while BTS system only achieved a factor of 0.95. In 80 percent of all synthetic systems, the SET system had a lower stack consumption than the BTS system.

## VIII. Discussion

*a) Threats to validity and generalizability:* One threat to validity of our results is that we used our own WCSC measurement to evaluate the SET mechanism. This was necessary, as no other WCSC method supports intra-thread stack switching. However, as already discussed in Section VII-D2, the parameter class without blocking threads in Figure 4c resembles the standard evaluation scenario for basic tasks. We found that for this class the results are in the same range (~ 30 %) as the numbers reported by others [11]. Furthermore, we have conducted a conservative evaluation of the advantages of the SET concept: We not only used the same WCSC analysis for all evaluations, but also the same set of preemption constraints for BTS and SET systems. The fine-grained function-level preemption constraints yields tighter bounds for the BTS than the regular WCSC method that only consider task-level preemption constraints. Hence, our approach will probably also be advantageous for BTS systems that do not support SETs. This, however, is a topic for further research.

The generalizability of our findings might be limited due to the construction of the synthetic benchmark systems. It is for sure that some thread-dependency graphs and call graphs benefit more from the SET concept than others. We tried to minimize this potential threat by restricting the form of these graphs as little as possible. Furthermore, our evaluation along five different dimensions gives us confidence that the observed trends are stable over a wide range of system parameters.

*b) Compiler limitations:* The presented approach to implement switch functions depends on compiler and RTOS modifications, which may be problematic for the broader applicability in industry, especially if closed-source compilers are used. However, as we already hinted in the introduction, switch functions can also be implemented with one additional BT, per SET. Instead of calling the switch function directly, the thread stores the function pointer and the arguments in a global memory area and activates its corresponding artificial BT, which executes the actual function call. This method requires more interaction with the RTOS and higher overheads, but one could still benefit from the optimized switch-function set and the WCSC analysis.

Another, more invasive, but faster, approach to implement switch functions is to reconstruct our modifications with inline assembler and binary post-processing: For example for GCC 8.2, we can insert the stack-switch instruction at the function entry with inline assembler. Additionally, we clobber the stack pointer[2] to force the usage of base *and* stack pointer. Furthermore, we have to post process the binary to reorder the instruction to the correct position within the function prologue. We can achieve the `TOS_BTS` bookkeeping without RTOS modifications if user-definable hooks are provided for thread preemption and resumption. For example, OSEK provides, with PreTaskHook and PostTaskHook, such extension points.

Our compiler modifications also have an impact on potential optimizations, as we need the compiler to use a frame/base pointer *and* a stack pointer. This disables certain optimizations, such as `-fomit-frame-pointer`, which avoids the usage of a frame pointer whenever possible to reduce the register pressure. However, in practice, the effects of `-fomit-frame-pointer` are negligible in general; in our case it furthermore affects only switch functions – the compiler is still allowed to omit the frame pointer in all other functions. As the number of switch functions is small (6 functions are average), the impact will be very low.

*c) Callee vs. caller-site stack switch:* One decision for the implementation of SETs is whether the stack switch is done at the call-site or within the called function. We decided to modify the switch function itself since it requires fewer code modifications and can be done efficiently with only one additional instruction. On the other hand, call-site stack switching would give us even finer control over the location of the call frame and the same function could invoked either on the BTS or on a private stack, which might lead to an even lower stack consumption. However, the invocation of a parameterized function on another stack requires larger compiler modifications and the simultaneous handling of two stack pointers when pushing arguments from the private stack onto the BTS. For calling conventions that use registers for parameter passing this might be simpler. This is a topic for further research.

*d) Library functions in the call graph:* As a switch function transfers not only its own stack frame, but also the frames of its children, SETs allow stack sharing between leaf functions and leaf subgraphs. Therefore, the potential benefit is directly related to the size distribution of stack frames. With increasing distance from the thread entry, it is more likely that we can transfer a function onto the BTS. For actual systems, these functions are also more likely to require a larger amount of stack, as the application logic is often located close to the entry function and most of the stack consumption originates from the activation of library functions deep down in the call graph. As such library functions are often shared by several threads, their transfer to the BTS shrinks the WCSC of several private stacks.

*e) Memory protection:* An important issue with any kind of stack sharing is memory protection, which in embedded control systems is typically provided by means of a *memory protection unit (MPU)*. In distinction to an *memory management unit (MMU)*, which implements protection by page-wise address translation, an MPU controls the access to ranges of the physical memory. For the execution of a basic task, an RTOS with memory protection configures the MPU to grant access only to the unused part of the shared stack to ensure integrity of the preempted basic task's state. For a SET, the RTOS must do the same, but additionally grant access via the MPU to the SET's private stack, which contains the actual parameters and, potentially, other variables that were passed by reference. Hence, SET support demands an additional MPU range. Furthermore, the RTOS must ensure read-only access to the `TOS_BTS` variable. However, in this domain, kernels typically already provide read-access to a part or even the complete kernel state for efficiency reasons. A typical platform that is well suited for SETs would be the Infineon TriCore [27], which is widely used in safety-critical automotive systems. Its MPU provides four ranges for data memory; the CPU core additionally offers global address registers, which can be marked as read-only for the application, to efficiently store the `TOS_BTS` variable. To sum up: SET support works well together with memory protection and just requires one additional MPU range.

*f) Worst-Case Execution Time (WCET) Impact:* Since we do not modify the callgraph(s) and the control-flow graphs of individual functions, the program structure stays intact, as well as all preemption relations between threads. Therefore,

[2]`asm volatile("mov TOS_BTS, %%esp;" ::: "%esp");|}`

regular WCET analyses can be with small modifications on the machine-code level: (1) One additional non-branching, memory-read instruction (`mov esp, [TOS_BTS]`) is added for every switch function. (2) For the cache analysis, it has to be considered that the switch function and its children work on a stack from a different memory region. However, every thread-execution path, also the longest path that constitutes the WCET, includes at most one stack-switch operation. Therefore, we only have to account for a single stack discontinuity. The impact of SET can be roughly estimated: (1) 1 instruction, 1 cache miss for `TOS_BTS`, and 1 cache eviction. (2) 1 cache miss and 1 cache eviction for the first stack access after the discontinuity.

## IX. Related Work

The discrepancy between the pessimistic allocation of static private stacks and the actual combined dynamic stack usage was addressed from different directions:

*a) Dynamic allocation of function-call frames:* The first direction is to allocate stack space not statically for each thread, but allocate the required memory on demand. For MESA [17], call frames were allocated from a specialized heap, linked to their dynamic predecessors and freed on return. This allowed the compact storage of activation frames since frames from different threads are mixed on the heap. Yi et al. [32] fruitfully applied per-function dynamic frame allocation to wireless sensor networks as their nodes suffer from strict memory constraints. As a per-call scheme for frame allocation imposes a high run-time overhead, Grunwald and Neves [12] analyze the function-call graph to insert allocations of stack segments only at neuralgic program points where the current stack segment has potential to overflow. Similarly, Behren et al. [2] segmented the call graph into regions that have an upper limit on stack consumption. Checkpoints are inserted into the region-entering function to ensure that the current stack segment can hold all frames up to the next checkpoint. In comparison to [12, 2], MTSS [22] starts each thread on a statically-allocated stack and react to imminent stack overflows. Run-time checks in each function detect overflows and page-sized stack segments are allocated from the system allocator to utilize all available system memory. Mauroner and Baunach [21] brings the reactive MTSS scheme to the hardware level, and use a specialized OS-aware MMU to transparently grow and shrink the available stack space. In contrast to software-based stack segmentation schemes, the MMU exhibits a linear logical stack space to the threads. Compared to our approach that switches unconditionally at the switch functions to the pre-allocated shared stack, the dynamic frame-allocation schemes pay continuously for run-time checks and for maintaining the allocator. Furthermore, these schemes work in a best-effort manner and give no upper limit on the stack consumption.

*b) Preemption-threshold scheduling (PTS):* The real-time community proposed to modify the scheduling parameters such that the system remains schedulable, but the usage of a shared stack becomes beneficial. Wang and Saksena [30] proposed preemption-threshold scheduling: Each task is assigned a preemption priority that it uses to preempt other tasks and a preemption threshold that it uses to prevent preemption by other tasks. Thereby, all tasks with the same preemption threshold cannot be active at the same time and, therefore, can share their space on the shared stack. Ghattas and Dean [11] showed that *preemption-threshold scheduling (PTS)* effectively decreases stack consumption and worst-case response time and increases the schedulability. For partitioned and global fixed priority scheduling, Wang et al. [29] and Wang, Gu, and Zeng [28] use ILP to find optimal priority and preemption-threshold assignments that are still schedulable but minimize the stack consumption. Compared to our approach, the PTS approaches change priorities and thresholds to optimize the stack consumption and, thereby, change the real-time schedule. Since we take the priority assignment for granted and optimize stack consumption at the system level, our approach can be combined with PTS. Furthermore, our switch functions exploit potential savings on the call-graph level, while PTS works on the granularity of whole tasks. Also, none of these methods handles the execution of blocking threads on the shared stack.

*c) Stack resource policy (SRP):* Coming from a different direction, Baker [1] proposed the SRP as an extension to the *priority-ceiling protocol (PCP)* [26]. When systems use the SRP to access shared resources, tasks raise their dynamic priority immediately to the ceiling protocol of the requested resource, preventing all other tasks that potentially request the resource from being scheduled. Thereby, resource acquisitions cannot block, deadlocks are prohibited, and the resource-requesting tasks can run on a shared stack. Gai, Lipari, and Di Natale [10] showed that preemption thresholds are a special case of the SRP where tasks take an implicit resource, which is shared by all tasks of the same preemption-threshold level. Yao and Buttazzo [31] first considered the task's fine structure, in terms of AUTOSAR runnables. They start with tasks that have already a priority and execute a fixed sequence of runnable linearly. Their algorithm assigns an intra-task preemption threshold to each runnable, which is enforced by pseudo SRP request enclosing the runnable code. Zeng, Di Natale, and Zhu [33] extended this model and also modified the runnable-to-task mapping to minimize the stack consumption. Again, the knob to save stack space is the real-time configuration and, thereby, the schedule is changed. Again, no method considered blocking and when a thread's fine-structure was used, the model was much simpler than the full call graph considered by us.

*d) Tighter Worst-Case Stack Consumption Bounds:* Hänninen et al. [15] proposed a WCSC analysis for shared-stack systems with time-triggered and event-triggered tasks. They utilized the detailed timing information about offsets to reduce the over approximation of used stack space. Later [3], they extended their approach to systems with offsets and precedences. Our approach to determining the WCSC does not rely on this detailed timing information but on flow-sensitive analysis of the GCFG. For interrupt-driven programs without RTOS, Brylow [5] used model checking and Regehr, Reid, and Webb [25] abstract interpretation to determine an upper stack bound.

## X. Conclusion

Stack sharing among several threads is an effective way to reduce the overall memory consumption of an embedded real-time system. With semi-extended tasks, we bridge the gap between non-blocking threads, which can be executed on a shared stack, and threads that passively wait within their control flow for an event to happen. We described an efficient way for a thread to switch from its private stack to the shared stack without activating the operating system. Our IPET-based worst-case stack consumption analysis supports systems with a mix of basic, semi-extended, and extended tasks. Furthermore, our analysis supports the incorporation of fine-grained preemption constraints, which we extract from the global control-flow graph of the system. By using a genetic algorithm, we applied the semi-extended task approach selectively and achieved significant stack saving over a wide range of system parameters. We improved the state-of-the-art, for 80 percent of our synthetic benchmark systems and could further reduce the stack size on average by 7 percent.

## Acknowledgements

The authors thank the anonymous reviewers for their feedback. Furthermore, we thank Stefan Bader and Roman Sommer for the fruitful discussions about the initial implementation. This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/4-1 and SFB/TR 89 ("Invasic"), Project C1.

The source code, changes to LLVM and dOSEK, the raw results, and a virtual-machine image with the experimental setup are available at:
`https://www.sra.uni-hannover.de/Research/AHA/SET/`.

## List of Acronyms

| | |
|---|---|
| **BT** | basic task |
| **BTS** | basic-task stack |
| **ET** | extended task |
| **CFG** | control-flow graph |
| **ILP** | integer linear programming |
| **IPET** | implicit path-enumeration technique |
| **GCFG** | global control-flow graph |
| **MMU** | memory management unit |
| **MPU** | memory protection unit |
| **PCP** | priority-ceiling protocol |
| **PTS** | preemption-threshold scheduling |
| **RTOS** | real-time operating system |
| **SET** | semi-extended task |
| **SRP** | stack resource policy |
| **WCSC** | worst-case stack consumption |

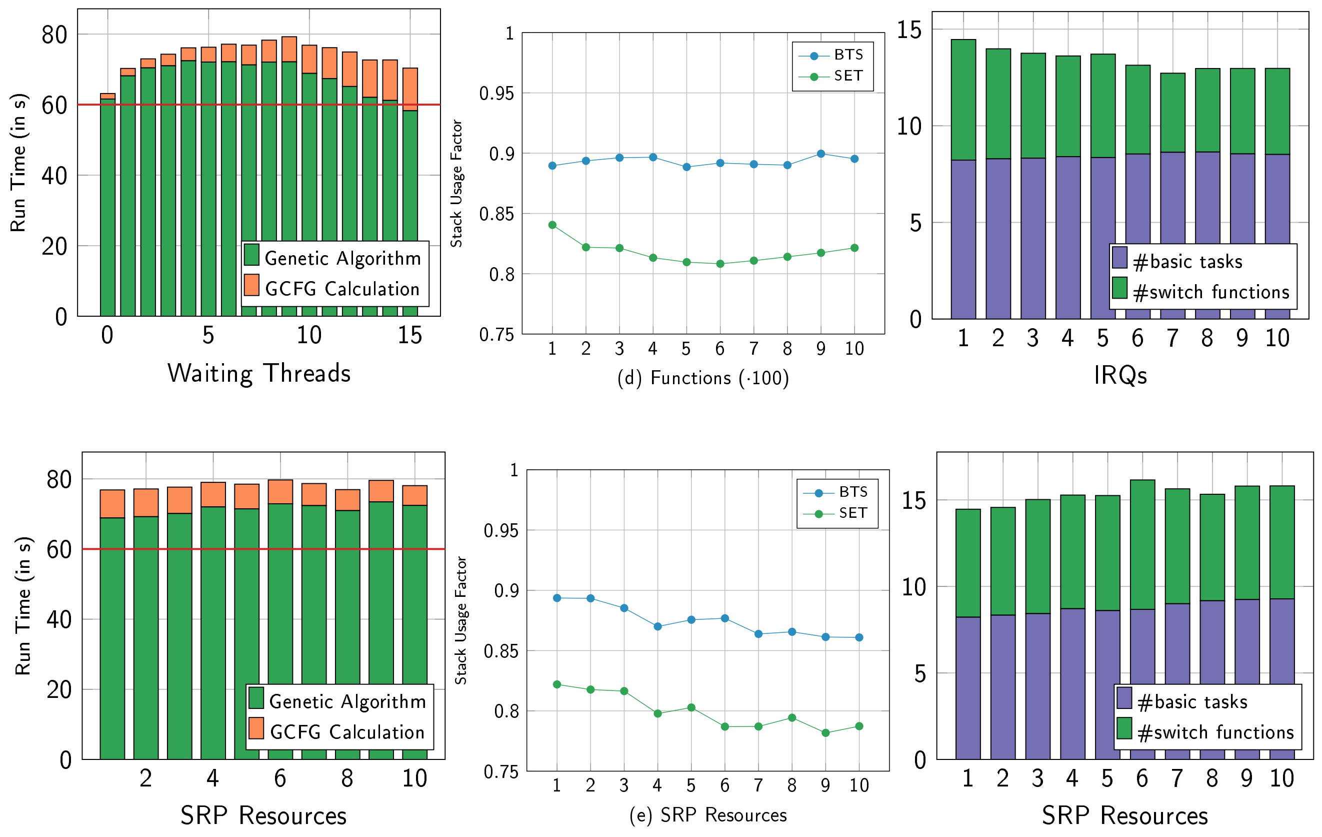


Fig. 6: Extension table for Figure 3, Figure 4, and Figure 5.

## References

[1] Theodore P. Baker. "Stack-based Scheduling for Realtime Processes". In: *Real-Time Systems Journal* 3.1 (Apr. 1991), pp. 67–99. ISSN: 0922-6443. DOI: 10.1007/BF00365393.

[2] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 268–281. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945471.

[3] M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin. "Bounding Shared-Stack Usage in Systems with Offsets and Precedences". In: *2008 Euromicro Conference on Real-Time Systems*. 2008, pp. 276–285. DOI: 10.1109/ECRTS.2008.29.

[4] Manfred Broy. "Challenges in Automotive Software Engineering". In: *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)* (Shanghai, China). New York, NY, USA: ACM Press, 2006, pp. 33–42. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134292.

[5] Dennis Brylow. "Static Checking of Interrupt Driven Software". PhD thesis. Purdue University, Aug. 2003. URL: http://www.mscs.mu.edu/~brylow/papers/Brylow-Dissertation2003.pdf.

[6] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. "Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis". In: *ACM Transactions on Embedded Computing Systems* 16.2 (2017), 35:1–35:25. DOI: 10.1145/2950053.

[7] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems". In: *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)* (Pacific Grove, CA, USA). New York, NY, USA: ACM Press, Sept. 1991, pp. 122–136. ISBN: 0-89791-447-3. DOI: 10.1145/121132.121155.

[8] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki — a Lightweight and Flexible Operating System for Tiny Networked Sensors". In: *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*. Tampa, FL, USA, Nov. 2004.

[9] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. Boulder, Colorado, USA, Nov. 2006. URL: http://dunkels.com/adam/dunkels06protothreads.pdf.

[10] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. "Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip". In: *Proceedings of the 22Nd IEEE Real-Time Systems Symposium*. RTSS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 73–. ISBN: 0-7695-1420-0.

[11] Rony Ghattas and Alexander G Dean. "Preemption threshold scheduling: Stack optimality, enhancements and analysis". In: *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*. IEEE. 2007, pp. 147–157.

[12] Dirk Grunwald and Richard Neves. "Whole-Program Optimization for Time and Space Efficient Threads". In: *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (Cambridge, MA, USA). New York, NY, USA: ACM Press, 1996, pp. 50–59. ISBN: 0-89791-767-7. DOI: 10.1145/237090.237149.

[13] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. Seventh. New York, NY, USA: McGraw-Hill, 2001.

[14] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. "dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel". In: *Proceedings of the 21st IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '15)*. Washington, DC, USA: IEEE Computer Society Press, 2015, pp. 259 –270. DOI: 10.1109/RTAS.2015.7108449.

[15] K. Hänninen, J. Maki-Turja, M. Bohlin, J. Carlson, and M. Nolin. "Determining Maximum Stack Usage in Preemptive Shared Stack Systems". In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 2006, pp. 445–453. DOI: 10.1109/RTSS.2006.18.

[16] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. "TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS". In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*. Berkeley, California: ACM, 2009, pp. 127–140. ISBN: 978-1-60558-519-2. DOI: 10.1145/1644038.1644052.

[17] Butler W. Lampson. "Fast Procedure Calls". In: *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*. ASPLOS I. Palo Alto, California, USA: ACM, 1982, pp. 66–76. ISBN: 0-89791-066-4. DOI: 10.1145/800050.801827.

[18] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). Washington, DC, USA: IEEE Computer Society Press, Mar. 2004.

[19] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. "TinyOS: An Operating System for Wireless Sensor Networks". In: Ambient Intelligence. Heidelberg, Germany: Springer-Verlag, 2005.

[20] Yau-Tsun Steven Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration". In: *ACM SIGPLAN Notices*. Vol. 30. ACM. 1995, pp. 88–98.

[21] F. Mauroner and M. Baunach. "StackMMU: Dynamic stack sharing for embedded systems". In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2017, pp. 1–9. DOI: 10.1109/ETFA.2017.8247614.

[22] Bhuvan Middha, Matthew Simpson, and Rajeev Barua. "MTSS: Multitask Stack Sharing for Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 7.4 (Aug. 2008), 46:1–46:37. ISSN: 1539-9087. DOI: 10.1145/1376804.1376814.

[23] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.

[24] Peter Puschner and Anton Schedl. "Computing Maximum Task Execution Times: A Graph-Based Approach". In: *Real-Time Systems* 13 (1997), pp. 67–91.

[25] John Regehr, Alastair Reid, and Kirk Webb. "Eliminating Stack Overflow by Abstract Interpretation". In: *ACM Transactions on Embedded Computing Systems* 4.4 (2005), pp. 751–778. ISSN: 1539-9087. DOI: 10.1145/1113830.1113833.

[26] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". In: *IEEE Transactions on Computers* 39.9 (Sept. 1990), pp. 1175–1185. ISSN: 0018-9340. DOI: 10.1109/12.57058.

[27] *TriCore 1 User's Manual (V1.3.8), Volume 1: Core Architecture*. Infineon Technologies AG. 81726 Munich, Germany, Jan. 2008.

[28] Chao Wang, Zonghua Gu, and Haibo Zeng. "Global Fixed Priority Scheduling with Preemption Threshold: Schedulability Analysis and Stack Size Minimization". In: *IEEE Transactions on Parallel and Distributed Systems* 27.11 (Nov. 2016), pp. 3242–3255. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2528978.

[29] Chao Wang, Chuansheng Dong, Haibo Zeng, and Zonghua Gu. "Minimizing Stack Memory for Hard Real-Time Applications on Multicore Platforms with Partitioned Fixed-Priority or EDF Scheduling". In: *ACM Transactions on Design Automation of Electronic Systems* 21.3 (May 2016), 46:1–46:25. ISSN: 1084-4309. DOI: 10.1145/2846096.

[30] Yun Wang and Manas Saksena. "Scheduling Fixed-Priority Tasks with Preemption Threshold". In: *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*. RTCSA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 328–. ISBN: 0-7695-0306-3.

[31] Gang Yao and Giorgio Buttazzo. "Reducing Stack with Intra-task Threshold Priorities in Real-time Systems". In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT '10. Scottsdale, Arizona, USA: ACM, 2010, pp. 109–118. ISBN: 978-1-60558-904-6. DOI: 10.1145/1879021.1879036.

[32] Sangho Yi, Seungwoo Lee, Yookun Cho, and Jiman Hong. "OTL: On-Demand Thread Stack Allocation Scheme for Real-Time Sensor Operating Systems". In: *Computational Science – (ICCS'07)*. Ed. by Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 905–912. ISBN: 978-3-540-72590-9.

[33] Haibo Zeng, Marco Di Natale, and Qi Zhu. "Minimizing Stack and Communication Memory Usage in Real-Time Embedded Applications". In: *ACM Trans. Embed. Comput. Syst.* 13.5s (July 2014), 149:1–149:25. ISSN: 1539-9087. DOI: 10.1145/2632160.