*Sys*WCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems

Christian Dietrich*, Peter Wägemann‡, Peter Ulbrich‡, Daniel Lohmann*
{dietrich, lohmann}@sra.uni-hannover.de {waegemann, ulbrich}@cs.fau.de
*Leibniz Universität Hannover (LUH)
‡Friedrich-Alexander Universität Erlangen-Nürnberg (FAU)

***Abstract*—The worst-case response time (WCRT) – the time span from release to completion of a real-time task – is a crucial property of real-time systems. However, WCRT analysis is complex in practice, as it depends not only on the realistic examination of worst-case execution times (WCET), but also on system-level overheads and blocking/preemption times. While the implicit path enumeration technique (IPET) has greatly improved automated WCET analysis, the resulting values still need to be aggregated manually with the system-level overheads – an error-prone and tedious process that yields overly pessimistic results.**

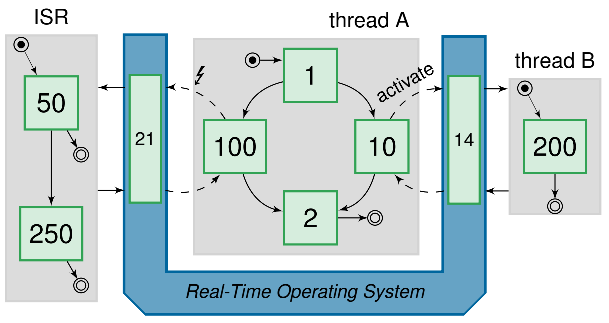
With *Sys*WCET, we provide an integrated approach for the automated WCRT analysis across multiple threads of execution, locks, interrupt service routines, and the real-time operating system (RTOS) in particular. Our approach spans a single IPET formulation over the whole system and exploits RTOS and scheduler semantics to derive cross-kernel flow facts in order to significantly reduce pessimism in the WCRT analysis.

We evaluate our approach with a fully functional implementation of *Sys*WCET for the automotive OSEK-OS standard (ECC1), including threads, alarms, interrupt-service routines, events, and PCP-based resource management.

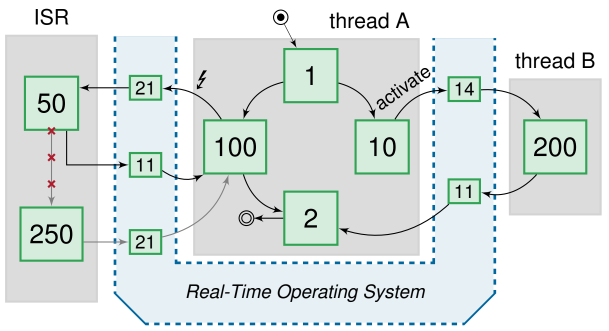
I. Introduction

The *worst-case response time* (WCRT) is a vital temporal property of real-time tasks with hard deadlines. Such tasks are usually embedded in a larger system context by mapping them to preemptive threads provided by a *real-time operating system* (RTOS). Hence, the WCRT of an individual task cannot be determined in isolation from its *worst-case execution time* (WCET), but needs to reflect worst-case overheads and interferences induced by other tasks and the RTOS.

The common approach to the inclusion of such task-extrinsic factors is *compositional* WCRT analysis. Components are analyzed and aggregated bottom-up, which essentially leads to a summation of their WCRTs and latencies for preemptive priority-based scheduling. For example, Figure 1a illustrates this approach by the WCRT analysis of *thread A*, which is implemented as an RTOS thread and accompanied by *thread B* and an *interrupt-service routine* (ISR). In a first step, the individual WCET evaluates to 103 cycles for thread A, 200 cycles for thread B, and 300 cycles for the ISR, respectively. Subsequently, precedences and interference are handled by summation of these local worst-case estimates. Thread-preemption delays can be easily accumulated: The activation of thread B (with higher priority) in the right branch bloats the WCRT of A to 303 cycles ($103 + 200$, neglecting the RTOS influence for



(a) Compositional WCRT analysis: The WCET of all system components (ISR $= 300$, thread A $= 103$, thread B $= 200$, RTOS $= 21$) is computed in isolation. Since the RTOS acts as a natural boundary for the analysis knowledge we can achieve only pessimistic results.



(b) Integrated WCRT analysis: A permeable RTOS boundary allows the integration of global cross-kernel flow facts, which result in tighter WCRT bounds due to inter-thread knowledge. For example, the ISR takes a shortcut exit, if it interrupted thread A in the left branch.

Fig. 1: Motivating example: WCRT analysis of thread A.

now). Asynchronous events are more difficult to handle, as their occurrence is unpredictable from a task-local perspective. Assuming only a single activation and servicing of the ISR in our example raises the WCRT to 603 cycles ($303 + 50 + 250$). Finally, there is the RTOS itself. In practice, its timing impact is either neglected or estimated overly pessimistically [24], but rarely realistically: Syscalls, for example, are generally assumed to be synchronous and their costs are approximated by the WCET of the longest path through the kernel [10] (i.e., 14 and 21 cycles for each OS-interaction in thread B and the ISR, respectively). Finally, all these local estimates need to be aggregated to determine the WCRT of thread A, which here leads to a total of 659 cycles ($603 + 2 \cdot 21 + 2 \cdot 14$).

We will soon show that this result can be brought down to 238 cycles ($1 + 10 + 14 + 200 + 11 + 2$, rightmost path) by employing additional system knowledge.

A. Problem Statement

Overall, the commonly applied bottom-up aggregation of individual, local WCETs is easy to implement, yet fraught with inevitable over-estimations that rise tremendously on higher levels. In this context, we identified the segregation between the RTOS and the application as a major obstacle for improvements. The application–RTOS boundary is virtually impenetrable for bottom-up analysis, denying access to valuable knowledge.

We therefore suggest a top-down approach with a global WCRT analysis that, in particular, integrates RTOS semantics as well as thread interactions into the analysis. For intra-thread analysis, the *implicit path enumeration technique* (IPET) [21, 28] has already drastically reduced pessimism in the estimation of WCETs. It composes well with techniques for the analysis of hardware effects, such as the influence of caches and pipelines [22]. Further refinements of the IPET already remove infeasible and mutually exclusive paths [8, 9, 19] by exploiting knowledge about the control flow and data dependencies. In this paper, we leverage the IPET for an automated whole-system WCRT analysis in order to solve the aforementioned problems with automated inter-thread analysis. In a nutshell, our approach makes it possible to also remove paths that become infeasible across the RTOS: Informally, such an *infeasible system execution path* is a path that we can only prove infeasible due to the current RTOS state, which in turn is determined by control-flow decisions taken by other threads.

Figure 1b demonstrates the potential for tightening the WCRT estimates: The central issue is the absence of flow facts for non-hierarchical control flows. If thread A has taken the left path in our example, thread B becomes infeasible and will never be dispatched. Additionally, the path through the ISR depends on the actual interruption context: If triggered in thread A's block 100, the handler always takes the shortcut exit (50 cycles vs. $50+250$ cycles). Likewise, not all paths through the kernel itself are independent from their calling site and the RTOS state. Self-termination of thread B, for example, may be faster than other context switches (11 cycles vs. 14 cycles) since the kernel certainly resumes to thread A. This kind of RTOS-context sensitivity is hard to analyze and requires global knowledge about the system semantics (e.g., scheduling and synchronization). Finally, WCRT analysis is a fix-point search problem [4], as the physical environment impacts the behavior of the system and thereby also possible paths. For instance, the minimal inter-arrival time of interrupts (whose handling takes computational time) leads to a cyclic dependency between WCRT and number of interrupts. Assuming that such global system knowledge is available, we could derive the actual WCRT for thread A of 238 cycles (rightmost path).

B. Our Contributions

Multiple challenges arise form pursuing the envisioned top-down approach: (1) analysis of RTOS–application interaction for a given task set and derivation of global control flows, (2) incorporation of semantics for fixed-priority scheduling and synchronization as well as interrupt handling, and (3) construction of an integrated IPET problem formulation.

In this paper we present *Sys*WCET, an integrated analysis approach to tackle the aforementioned limitations of compositional WCRT estimation. Based on IPETs, our approach facilitates formulations of whole-system analyses that span over multiple threads of execution and ultimately break up the segregation of application and RTOS. Additionally, *Sys*WCET integrates the semantics of fixed-priority scheduling and synchronization along with the impact of asynchronous interrupts. The key contributions of this paper are:

- Whole-system control-flow and RTOS-state analysis.
- Formulation of response-time problems for global, non-hierarchical control-flows as an integer linear program, which enables cross-kernel analysis in particular.
- Full integration of fix-priority RTOS semantics, preemptive scheduling, and resource management.
- Handling of asynchronous events and interrupts by inclusion of minimal inter-event and inter-arrival times.
- A fully operational prototype for OSEK ECC1: The *Sys*WCET toolchain that automatically analyzes given OSEK applications and constructs the IPET problem as well as the concrete RTOS instance.

II. System Model and Restrictions

Our automated cross-kernel WCRT analysis demands three basic properties from the real-time system: (1) An RTOS with a deterministic scheduling policy, such as fixed-priority preemptive scheduling. (2) All scheduling-relevant system objects (threads, ISRs, resources, etc.) and their configuration are known ahead of time; either provided by some configuration file or statically extractable from the source code. (3) Syscall locations and system–object-referencing arguments are known at compile time. Although we rely on a mostly static application structure, the system has not to be deterministic in its dynamic behavior – asynchronous timers and ISRs that influence the scheduling are well supported.

The proposed *Sys*WCET approach operates on the system-wide control-flow level and aims to reduce the RTOS-related pessimism in WCRT analysis. Therefore, we focused on a system-wide IPET formulation and chose a simple processor model [29] without inter-instruction effects or caches. Hence, WCET values basically depend only on the number of executed instructions. We are aware that this hardware-timing model is realistic only for few relevant architectures (e.g., ARM Cortex-M0+ [3]); however, it perfectly fits the focus of this work. In fact, the *Sys*WCET approach itself is not dependent on the hardware timing model and should be combinable with more advanced models (and even increase their feasibility), such as techniques to refine cache-related preemption delays [11, 37]. We shall discuss this topic further in Section VIII.

A. Overview of OSEK-OS

The OSEK standard defines a widely used class of fixed-priority RTOSs and has been the dominant industry standard for automotive applications for the last two decades. Without loss of generality, we based our approach on the system model mandated by the OSEK-OS standard [26]. In the following, we briefly introduce the abstractions provided by its API.

Basically, OSEK offers two main control-flow abstractions: *ISRs* and *tasks* (i.e., threads and not to be confused with abstract real-time tasks or jobs). ISRs are activated asynchronously by the hardware and have limited access to system services, while threads possess a statically assigned priority and are activated synchronously by software. Threads are allowed to use all system services and are executed according to a fixed-priority preemptive scheduling policy. On each new activation, threads start from the very beginning until their (self-) termination. Whether ISRs are nestable is implementation defined; for this work, we assume non-interruptible ISRs.

Critical sections can be synchronized either by a coarse-grained global interrupt lock, or more fine-grained *resource* objects. Based on a *stack-based priority-ceiling protocol* [1], OSEK resources ensure mutual exclusion while preventing deadlocks and unbounded priority inversion. Furthermore, precedence constraints can be stated by events: A thread can wait for an event to be set and remains in the waiting state until another control flow signals the arrival of the event.

Recurring periodic as well as aperiodic activations and events are triggered with the help of statically declared *alarms*. These are configured with a phase and a period, triggered by a hardware-timer interrupt and can either be started automatically at system startup or dynamically at run time.

For a specific application, the developer declares all system objects and their parameters in a domain-specific configuration file. At compile time, a system generator derives the concrete RTOS instance statically, allocating fixed-sized arrays of preconfigured system objects, and links application and OS library into a single system image.

III. Background

In a nutshell, we calculate the WCRT by modeling *system-wide* control flows with the implicit path enumeration technique [21, 28]. We extract information about the possible control flows for a given task from the *operating-system state-transition graph* (STG) [14]. In the following, we briefly introduce necessary background knowledge and basic concepts.

A. Atomic Basic Blocks

For static analyses, application code is usually partitioned into linearly executed basic blocks that are connected in the *control-flow graph* (CFG). Nevertheless, for a system-wide examination, the regular CFG structure is too fine-grained as it exposes too many nodes. Hence, we choose *atomic basic blocks* (ABBs) [15, 30] to partition the code and to abstract from the application's microstructure.

An ABB is a control-flow super structure that subsumes one or more basic blocks and conceptually spans between syscalls, forming a single-entry single-exit region: it has *exactly one* entry and one exit block, which typically is the delimiting syscall. Furthermore, ABBs may be split arbitrarily for optimization reasons as long as the result complies with the single-entry single-exit rule.

These construction rules imply that an ABB executes, if no interrupt occurs, *atomically* from a scheduling perspective. While syscall blocks are determined in their size, the extent of *computation* ABBs (no syscall) is variable, but restricted by the surrounding syscall blocks. For our implementation, we use ABBs of maximal size. In Section VIII, we will discuss the influence of the ABB size on the analysis results.

For the rest of the paper, we will stick to the example system shown in Figure 2a, which consists of one ISR and three threads with high, medium, and low priority. The code is partitioned into ABBs; only ABB_2 and ABB_5 contain an explicit syscall. To make the example more concise, only ABB_1 and ABB_4 are interruptible by the hardware. Furthermore, we assigned costs to reflect required computation times.

B. Operating-System State Transition Graph

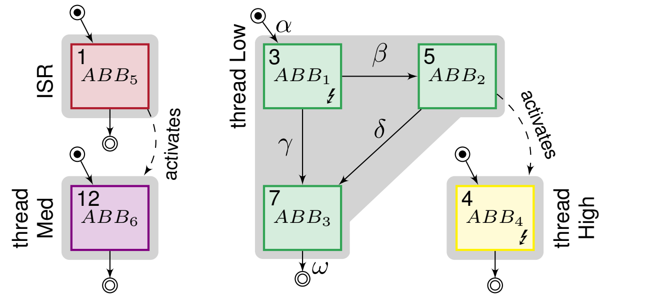
At this point, the ABB only captures a static view on the application structure. In order to also incorporate the dynamic behavior of the whole system, we employ the *operating-system* state-transition graph [13, 14]. It *explicitly* enumerates (a) all possible (abstract) system states and (b) all transitions between them. The system states include information like the list of runnable threads, acquired resources, and the currently running control flow. The transitions express all possible execution paths through the system, including thread switches. Transitions are caused by application control flow, syscalls, or external interrupt activations. In sum, the STG captures the complete system behavior including environmental influences like interrupts.

Conceptionally, the full STG contains *all* possible RTOS states S and the transitions T between them. It has exactly one entry state (S_b) that is set up by the boot code. In the system-state enumeration [14], the application logic (i.e., ABB graph), RTOS semantics (i.e., fixed-priority scheduling and resource protocols), and the environment model (i.e., minimal inter-arrival times of interrupts) are combined to calculate the STG. From the entry state, all OS states are explicitly enumerated by simulating the application logic on an abstractly instantiated and configured model of the RTOS. The RTOS model also complies OSEK's PCP-locking [1] mechanism, which achieves mutual exclusion by raising the priority of the lock-holding thread.

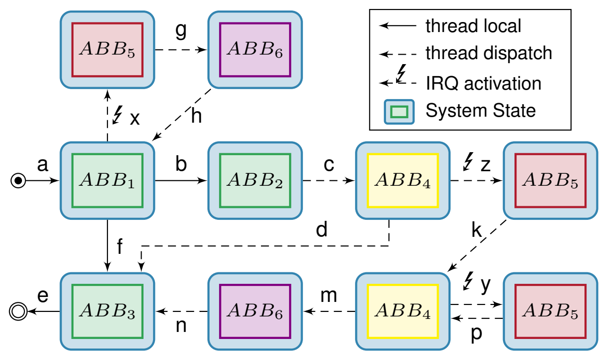
$$S = \{S_i \mid S_i \text{ is reachable from } S_b\} \qquad T \subseteq S \times S$$

Besides the currently executed ABB ($abb(S)$), each state has exactly one currently running thread ($thread(S)$). According to source and target state, we partition transitions into three semantically distinct groups: *Local Transitions* (T_{local}) proceed within the same thread. *Dispatch Transitions* ($T_{dispatch}$) switch to another thread. *Interrupt Transitions* (T_{irq}) dispatch to an ISR entry state and are activated asynchronously by hardware.

$$\begin{aligned} T_{local} &= \{t \mid t \in T,\ thread(t.from) = thread(t.to)\} \\ T_{dispatch} &= \{t \mid t \in T \setminus T_{local},\ t.to \neq isr_{entry}\} \\ T_{irq} &= \{t \mid t \in T \setminus T_{local},\ t.to = isr_{entry}\} \end{aligned}$$



(a) The application is structured into three threads with high, medium, and low priority and one interrupt service routine. The application code is partitioned into *atomic basic blocks* (ABBs). In ABB_2 (cost: 5), the low-priority thread activates the high-priority thread with a syscall. The ISR can be triggered in ABB_1 and ABB_4. The ISR issues a syscall that activates the medium-priority thread.



(b) This *state-transition graph* (STG) expresses all possible state transitions from the start to the termination of the low-priority thread. Every state carries and executes exactly one ABB.

Fig. 2: Example system

Figure 2b illustrates the resulting STG for our previous example (Figure 2a): Again, the response time of the low-priority thread is sought. Consequently, the graph spans from its initial start state (ABB_1, edge a) to the thread's termination state (ABB_3, edge e). Subsequent states and branches result from either local, dispatch, or IRQ transitions. ABB_4, for example, is only reachable by a consecutive execution of the low-priority thread (ABB_2, edge b) and the synchronous activation of the high-priority thread. Such syscalls (dashed edges), which are issued from the application code, manipulate the OS state and may lead to rescheduling and preemption. After termination of ABB_4, the low-priority thread is certainly resumed, as indicate by edge d. In a similar manner, edges (flash-tagged) are added for all possible asynchronous transitions to the ISR. Here, the system eventually returns to the interrupted state after ISR and all higher-priority threads are finished. In the end, each node in the STG represents a dedicated system state associated with a single ABB that is executed in this context. Consequently, ABBs may appear multiple times within the graph. For further details on the STG, we refer to our previous work [14, 13].

C. Implicit Path Enumeration Technique

The *implicit path enumeration technique* (IPET) [21, 28] is an established approach to derive a safe upper bound for the execution time of a program in a control-flow sensitive manner. Basically, the control-flow graph is translated to an *integer linear program* (ILP) with execution costs as weights from which a specialized solver derives an upper WCET bound. The control-flow graph is modeled with integer-valued frequency variables – one variable for each edge and each block. *Structural constraints* over these variables limit the possible value combinations and, therefore, encode the control-flow. Additional constraints about upper loop bounds or infeasible paths are added to the problem to assist the solver and cause a tighter bounding. To illustrate the technique, we give a small IPET model for the control-flow graph of the low-priority thread from Figure 2a (ABB_1, ABB_2, and ABB_3):

$$WCET = \max\left(x_{ABB_1} \cdot 3 + x_{ABB_2} \cdot 5 + x_{ABB_3} \cdot 7\right)$$
$$1 = x_\alpha$$
$$x_\alpha = x_{ABB_1} = x_\beta + x_\gamma$$
$$x_\beta = x_{ABB_2} = x_\delta$$
$$\underbrace{x_\gamma + x_\delta}_{\text{in-flow}} = x_{ABB_3} = \underbrace{x_\omega}_{\text{out-flow}}$$

First, we introduce a positive, integer-valued variable for each ABB and each control-flow edge (e.g., x_{ABB_1}, x_α). For each block, the sum over the incoming edges, and sum over the outgoing edges is constrained to be equal to the frequency of the block. For example, $x_{ABB_1} = x_\beta + x_\gamma$ corresponds to the control-flow decision in ABB_1 to take either edge β or edge γ but not both. Furthermore, a constraint ensures that the entry edge (α) is executed exactly once to make the problem bounded in the first place.

The ILP optimization objective is constructed by assigning an execution cost (c_{entity}) to every variable. The *maximized* sum over the products of execution cost and frequency is an upper bound for the actual WCET. All frequency variables constitute the execution count on the longest execution path.

In our example, we assigned arbitrarily chosen execution costs for all three ABBs ($c_{ABB_1} = 3$, $c_{ABB_2} = 5$, $c_{ABB_3} = 7$). In a real analysis, these execution costs stem from a hardware-specific instruction analysis. With this cost vector, the ILP solver derives a WCET estimate of 15 for the low-priority thread's body and outputs the maximizing assignments for the variables: all ABBs are executed once and only x_γ is zero; γ is not visited in the longest execution.

However, the value of 15 for the low-priority thread's WCET only considers the *thread-local* CFG, so it holds only if the low-priority thread runs in isolation. To obtain a valid WCRT, we have to further add the WCETs of all possibly preempting control flows (threads Med, High, and the ISR) plus any RTOS-induced overhead for the respective context switches and syscalls. This summing up of WCETs obtained by local analyses is state of the art but often leads to overly pessimistic results. In the following, we overcome this by spanning the IPET over the whole system instead.

IV. Whole-System Response-Time Analysis

*Sys*WCET is an approach to whole-system response-time analysis based on ILP. The overarching goal is to overcome application–RTOS boundaries by a global, top-down analysis that integrates environment, RTOS semantics, and thread interactions. Leveraging this system-state knowledge, *Sys*WCET reduces analysis pessimism and tightens WCRT bounds.

A. Integration of Fixed-priority Scheduling Semantics

A key element of our approach is to identify global execution paths that are infeasible or mutually exclude each other in the given context and system state. An activation of a thread with higher priority will, for example, always preempt the running thread, whereas an activation of a lower-priority thread will never lead to preemption. In any case, the results are well-defined control-flow decisions within those threads. Consequently, we call paths that are only infeasible due to the given RTOS system context *infeasible system execution paths*.

In contrast to data-flow–driven intra-thread analysis of infeasible paths, our inter-thread analysis is based on the system state in order to capture the semantics of the RTOS and, in particular, its fixed-priority scheduling and resource handling. Instead of formulating the global system state and, thus, the scheduling semantics as part of the ILP, *Sys*WCET leverages the STG for that purpose: During STG construction, a model of OSEK's fixed-priority scheduling and the stack-based *priority-ceiling protocol* (PCP) for locks and resources [1] is used to add feasible transitions accordingly. Hence, the STG is by construction already in full compliance with all possible execution paths through the concrete RTOS instance (including locks, scheduling, and interrupts) when transformed into an ILP. This way, the system state is handled independently and does not bloat the ILP unnecessarily.

B. Initial ILP Transformation

Technically, *Sys*WCET relies on the IPET, which is applied on a subgraph of the STG to calculate the maximal execution cost for the processing of a thread under consideration of the environment and RTOS semantics.

First, we have to identify a subset of the STG that is relevant for our analysis. For example, we extract a *partial* STG for a given thread that spans from its release until its completion. Start and end point are indicated by two ABB sets (ABB_{start} and ABB_{end}). From these, we derive a start-state set S_{start} and an end-state set S_{end}, and extract the subgraph between them with a depth-first search. The resulting partial STG (S_{sub}, T_{sub}) has possibly multiple entry states, multiple exits, and may consist of several disconnected components:

$$\begin{aligned} S_{start} &= \{s \mid s \in S, abb(s) \in ABB_{start}\} \\ S_{end} &= \{s \mid s \in S, abb(s) \in ABB_{end}\} \\ S_{sub} &= \{s \mid s \in S,\ s_s \xrightarrow{T*} s,\ s \xrightarrow{T*} s_e, \\ &\qquad s_s \in S_{start}, s_e \in S_{end}\} \end{aligned}$$

Since the partial STG is *the* defining input for our IPET construction, we will reference it only as STG from now on.

C. SysWCET IPET Construction

Our IPET problem consists of two layers: (1) the *state layer* models the STG and exposes *state-frequency variables* that indicate how often the system visits the corresponding system state. (2) The *ABB layer* consists of one IPET fragment for every occurring ABB. Deriving an activation frequency for each ABB from the state frequencies connects both layers. Thus, the actual flow facts are already included in the STG.

1) State Layer: Within the IPET problem, we formulate a subproblem from the STG and include the following system-level constraints. For each state and every transition, we introduce a positive, integer-valued frequency variable (x_s and x_t, respectively): structural constraints ensure that incoming and outgoing transitions have the same frequency as the state-frequency variable:

$$\forall_{s \in S} \left(\sum_{t \in \{* \to s\}} x_t = x_s = \sum_{t \in \{s \to *\}} x_t \right)$$

As there are multiple entries and exits, we introduce artificial state transitions to entry and from exit states. Their combined frequencies have to be equal to one:

$$\sum_{s \in S_{start}} x_{\to s} = 1 \qquad\qquad \sum_{s \in S_{end}} x_{s \to} = 1$$

Besides regular control-flow loops, additional loops may occur on the STG level. These are caused by interrupts, whose control-flow returns to the interrupted state (see Figure 2b, $x \to g \to h$) once serviced. Giving an explicit loop bound for interrupt-induced loops is impossible in general. Nevertheless, we have to ensure that an interrupt loop is only accounted for, if one of its entries is taken at least once. To express this structural constraint, we use an artificial yet sufficiently large loop bound M. If no loop entry is executed, the sum of all back edges in the loop must be ≤ 0:

$$\sum_{t \in \text{backlink edges}} x_t \leq \sum_{t \in \text{loop entries}} M \cdot x_t$$

However, restricting the mere occurrence of interrupt-induced loops is insufficient and we therefore model their overall count in addition. Consequently, for each interrupt source I we relate its minimal *inter-arrival time* (IAT), the WCRT (T_{WCRT}), and the frequency-variables (x_t) of all related interrupt transitions ($T_{irq,I}$). The resulting ILP optimization objective is replicated to a newly introduced variable T^*_{WCRT}. With this self-referential technique, loop constraints (i.e., interrupt frequency) are fed back to the optimization process of T_{WCRT}. This accounts for the factor that the longer the execution time, the more potential interrupts and vice versa. Consequently, a maximal interrupt-activation frequency can be derived from T_{WCRT} and the minimal IAT, limiting the combined interrupt-transition count. Furthermore, we take a possible release jitter J_I [4] of the source into account and allow the first interrupt to occur J_I ticks before the scenario starts, reflecting a delayed delivery of the interrupt:

$$\text{IAT}_{min,I} \cdot \sum_{t \in T_{irq,I}} x_t \leq (\text{IAT}_{min,I} + J_I + T_{WCRT})$$

With this constraint, the solver is free to distribute the maximal interrupt frequency over all interrupt transitions, which corresponds to an arbitrary arrangement of interrupts. In practice, multiple periodic alarms may be implemented and driven by a single timer ISR, which in turn has to fire multiple times before an alarm expires and issues a syscall. If the period of an alarm is constant, we constraint the activation path by deriving a maximal alarm-expiration frequency from the alarm's period and the basic timer frequency. This way, we eliminate over-approximations (e.g., thread activation) due to ineffective timer activations.

2) ABB Frequencies: The state layer provides a frequency x_s for every state s in the STG. To drive a machine-level IPET, we have to derive an activation frequency x_a for every ABB.

Since we allow interruption of threads by ISRs, the ABB frequency may be less than the combined state frequencies over all states executing the ABB. To illustrate this, assume that the start state in Figure 2b ($a = 1$, $b = 1$) is interrupted by the ISR ($x = 1$). Then ABB_5 executes and the RTOS resumes to ABB_1 ($h = 1$), resulting in a state frequency ($a + h$) of 2, although ABB_1 is completed only once.

We achieve a tighter WCRT bound by subtracting the number of *completed* interrupt-resume cycles from the state-frequencies. To reiterate, this is only valid for completed interrupts: if ABB_5 would be a terminal state ($a = 1$), the interruption would not resume ($h = 0$) and subtracting $x = 1$ from the frequency would lead to an under-approximation ($a + h - x = 0$).

For each ABB_a, we analyse the number of activations: first, states executing ABB_a (S_a) are identified For this state set, all interrupt ($T_{a,i}$) and resume transitions ($T_{a,r}$) derived. In the resume set, we only include transitions that are reachable via an interrupt transition, as these edges can form actual interrupt-resume cycles:

$$\begin{aligned}
S_a &= \{s \mid s \in S_{sub},\ abb(s) = a\} \\
T_{a,i} &= \{t \mid t \in T_{irq},\ t.from \in S_a\} \\
T_{a,r} &= \{t_r \mid t_r \in T_{dispatch},\ t_r.to \in S_a, \\
&\qquad t_i \in T_{a,i},\ t_i \xrightarrow{T*} t_r \ \}
\end{aligned}$$

From the interrupt and resume sets, we derive the number of interruptions that lack an associated resume ($x_{a,i}$). We use a *special ordered set* (SOS) of type 1 with two variables $x_{a,i}$ and $x_{a,r}$; only one variable can be larger than zero, while the other must be zero. We subtract the interruptions from the resumptions. If there are more interruptions, the sum is negative and $-x_{a,i}$ absorbs it; if there are additional resumptions, the sum is positive, $x_{a,r}$ absorbs it, and $x_{a,i} = 0$. Finally, we calculate the ABB frequency x_a: add all state frequencies, subtract all interrupt frequencies, and, again, add all "incomplete" interruptions $x_{a,i}$:

$$\begin{aligned}
\text{SOS1} : x_{a,i} > 0 &\veebar x_{a,r} > 0 \\
-x_{a,i} + x_{a,r} &= \sum_{r \in T_{a,r}} x_r - \sum_{i \in T_{a,i}} x_i \\
x_a &= \sum_{s \in S_a} x_s - \sum_{i \in T_{a,i}} x_i + x_{a,i}
\end{aligned}$$

We calculate the frequency x_4 for ABB_4 in our example from Figure 2b: The first constraint captures the number of additional interruptions with a SOS ($x_{4,i}$, $x_{4,r}$). The second uses the transition variables and $x_{4,i}$ to derive x_4:

$$\begin{aligned}
-x_{4,i} + x_{4,r} &= \overbrace{(x_p + x_k)}^{\text{resumes}} - \overbrace{(x_z + x_y)}^{\text{IRQs}} \\
x_4 &= \underbrace{(x_c + x_k + x_p)}_{\text{in-flow}} - \underbrace{(x_z + x_y)}_{\text{IRQs}} + \underbrace{x_{4,i}}_{\text{unresumed IRQs}}
\end{aligned}$$

3) ABB Layer: Finally, we embed an IPET problem for each ABB and activate the ABB scope with the ABB frequency x_a. Since ABBs are single-entry single-exit regions, we can construct a regular IPET including all loop bounds and path refinements. Note that we encode every ABB only once, even if it is referenced from many states.

Besides the application code, the ABBs also include all RTOS code and, therefore, we take the actual RTOS overheads into account. For example, ABB_2 from Figure 2a contains exactly the kernel path for the activation of thread High, which is not necessarily the longest path through the kernel.

At last, we assign a timing cost to all entities in the combined IPET problem: From the machine-code analysis, we get execution costs for basic blocks and intra-ABB edges. On the state layer, costs from the source's exit block to the target's entry block are assigned to state–state transitions. Furthermore, interrupt transitions T_{irq} carry the worst-case cost that arises in the source ABB if it is interrupted. By this over-approximation, we avoid the need to consider interruptions after every single instruction. For a more complex timing model that includes interference between different basic blocks, edge and block frequency variables can be used to drive specialized ILP fragments that add to the overall WCRT. However, this is a topic of further research and briefly discussed in Section VIII-A

D. Interrupt-Detection Latency

Up until now, we calculated the WCRT from a start ABB to an end ABB; we assumed that the processing starts in a well-defined start state ABB_{start}. But real-world systems do not immediately transition into this state when the activating event (i.e., *interrupt request* (IRQ)) occurs. Therefore, we have to determine a detection delay, which the system needs to transition into one of the start states.

This detection delay can stem from several sources: On the hardware level, the interrupt controller might have an arbitration delay to determine the highest-priority IRQ source. On the software side, the activation of ISRs is delayed, if applications or the kernel disable the interrupts (e.g., for synchronization).

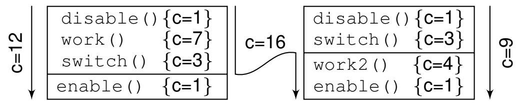


Fig. 3: Maximal interrupt detection latency

To complete our WCRT analysis, we calculate the interrupt detection pessimistically and add it to the WCRT.

The challenge in calculating the worst-case interrupt-blockade time is located within the RTOS: If the kernel disables interrupts during the context switch, the `enable()` instruction is executed in the context of the resumed thread. Since a context switch is a non-local control flow, it is not sufficient to find the lexically next `enable()`, but we need to find the most distant `enable()` in all resumed threads.

We calculate three WCETs adhering only local control flows with an IPET: from disabling interrupts to a lexically following enable ($T_{d \to e}$), from a disable to the context switch ($T_{d \to s}$), and from any resume point to a following IRQ enable ($T_{s \to e}$). The maximal interrupt-detection latency is then the maximum of $T_{d \to e}$ and ($T_{d \to s} + T_{s \to e}$). In Figure 3, the detection latency analysis reveals: $T_{s \to d} = 12$, $T_{d \to s} = 11$, $T_{s \to e} = 5$, and therefore the maximal interrupt detection delay is 16.

V. Implementation

We integrated the *Sys*WCET approach into the *d*OSEK [17] generator framework and the `platin` [27] WCET analysis tool. The *Sys*WCET toolchain automatically analyzes an OSEK application, generates the RTOS instance, calculates the STG, compiles a system image, and constructs the IPET problem. The complete toolchain, including our evaluation data, is publicly available [35].

*d*OSEK is, both, a generator framework and a RTOS that conforms to the OSEK [26] (see Section II-A) conformance class ECC1. This conformance class includes waiting states and OSEK resources, but only allows one thread per priority and no multiple thread activations (activations on already activated threads have no effect). *d*OSEK does, furthermore, not allow nested interrupts or interrupts during the kernel execution; both are implementation decisions that the OSEK standard allows. Originally designed to provide a fault-tolerant, application-specific RTOS implementation, *d*OSEK is also able to emit unhardened OSEK instances and already includes the STG calculation.

Our WCRT analysis starts with an application written in C++ and a system configuration provided by the developer expressed in an OSEK-specific configuration language (OIL). Furthermore, the developer annotated starting and end points (ABB_{start}, ABB_{end}) within the application with marker function calls. *d*OSEK compiles the application with CLang and extracts the application CFG from the LLVM [20] *intermediate representation* (IR).

We extended *d*OSEK to emit its analysis results in `platin`'s PML format [27]: the ABB to basic-block mapping, deduced flow facts for the kernel code, and the partial STG. The partial STG, which is located between the marker function calls, is extracted from the full STG with a depth-first search starting at all S_{start} states. The search stops descending if we encounter a local-transition edge (T_{local}) originating from a stop state (S_{end}).

As a target platform we choose PATMOS [33], a timing predictable hardware architecture that eases WCET analysis and thus allowed us focus on the core aspect of this work: the system analysis. PATMOS comes with its own analysis framework `platin`, which allowed for an easy integration of *Sys*WCET. `platin` imports the program information, the STG, and kernel-level flow facts to construct the IPET. It should, however, be noted that the *Sys*WCET concept is oblivious to special hardware features and thus the platform choice has no major implication for our approach (cf. Section II).

One problem of flow-fact handling is the mapping between CFG and machine code, since the compiler is allowed to reorganize code. Because ABBs are formed from LLVM basic blocks, we encountered the same problem. Luckily, `platin` already includes handling of control-flow relation graphs (CFRG) [18] to solve the mapping problem. A CFRG consists of ordered pairs that may reference a LLVM basic block in the first element and a machine-code basic block in the second one. A pair with two valid elements is called a *progress node*, since its execution synchronizes the progress on the LLVM and the machine level. Along these progress nodes, we propagate ABB- and LLVM-level flow facts down to the machine level.

In operating-system code, not everything is analyzable by our LLVM toolchain: Inline assembler (i.e., context switch and interrupt entry/exit) is handled by LLVM as a mere string that is copied verbatim into the generated assembly. Therefore, we fill up these gaps by parsing the disassembled machine code and integrate the missing instructions back into the PML file.

VI. Experimental Results

For an experimental validation, we apply our approach on application fragments and complete systems of differing sizes. We calculate the interrupt-detection latency, compare the accumulated WCRT and the *Sys*WCET WCRT, and give actual observed execution times.

A. Evaluation Scenarios

We use *d*OSEK [17] on the PATMOS [33] architecture as the base for our evaluation and choose three classes of benchmarks: system microbenchmarks, application microbenchmarks, and one larger control application.

With the first class of benchmarks, we show that our OS-state–based approach is well suited to measure WCRTs of typical RTOS usage patterns. This includes the activation of another thread, the minimal round-trip time for a *wait-and-wakeup* operation, or the WCRT of an ISR that activates a regular thread. The second class of scenarios consists of three applications that expose thread inter-dependencies and interaction with operating system and environment. With the third class of benchmarks, we demonstrate that our approach scales to larger systems with several interrupt sources.

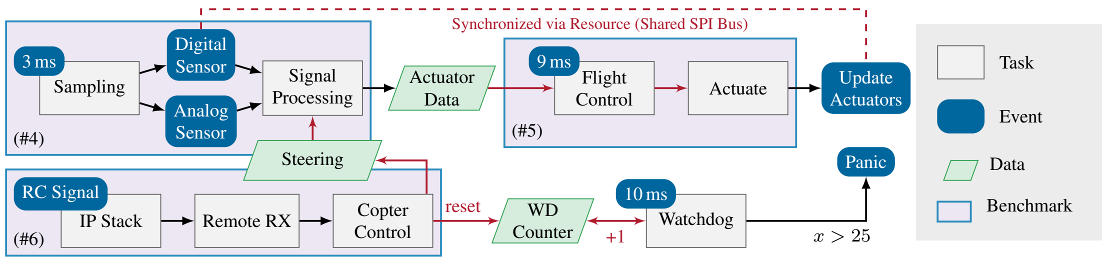


Fig. 4: The task setup of the *I4Copter*.

```
void th_HI_computation() { ... }

void th_LO_control() {
  for (i=0; i < 3; i++) {
    activate(th_HI_computation);
  }
}
```

Listing 1: Triple Modular Redundant Application

(#1, Listing 1) Triple Modular Redundancy: A low-priority thread activates a high-priority thread three times from a bounded loop. After the `activate`, *d*OSEK immediately switches to `th_HI_computation`. After completion, control is returned to the low-priority thread; both executions are interwoven. This scenario resembles a thread setup for a triple-modular redundant computation located in the high-priority thread. The actual result voting would take place after the loop.

(#2, Listing 2) Alarm-interfered Computation: The low-priority thread computes for over 750 000 instructions. Meanwhile, a timer interrupt with a minimal IAT of 100 000 instructions drives an OSEK alarm that triggers on every third timer interruption. The alarm activates a high-priority thread that is dispatched *after* the interrupted is completed.

(#3,Listing 3) Abortible Computation: A high-priority thread delegates a long running computation to a low-priority thread and waits for one of two events: either the computation thread itself signals completion or an interrupt with an IAT of 10 000 arrives and also wakes the high-priority thread to abort the computation.

(#4-#6) I4Copter: We derived the third class of benchmarks from the *I4Copter* [39] demonstrator, a safety-critical embedded control system (quadrotor helicopter) developed in cooperation with Siemens Corporate Technology. We extract the thread setup (i.e., no real computation code), including the RTOS interactions and the environment model, and analyze three different components of the system (Figure 4): (#4) signal gathering consists of 5 threads with a high priority. The component is activated periodically by an OSEK alarm and synchronized through an OSEK resource with the flight-control component. (#5) flight control is implemented in 3 threads with a medium priority. It is also activated periodically, but only on every third signal-gathering cycle. (#6) the remote-control component is activated asynchronously by an interrupt. The associated ISR activates a low-priority thread that blocks interrupts for synchronization. Periodic activations are driven by a timer interrupt, similar to Listing 2. The system contains another watchdog thread with the lowest priority, which does not interfere with the other components.

Naturally, all of our results will be highly specific for the application under test and a carefully crafted benchmark could overestimate the WCRT savings *Sys*WCET can achieve. Imagine a constructed benchmark with an excessive computation on a path that we later can prove infeasible by our cross-kernel approach and, thus, would bring *Sys*WCET an arbitrarily large WCRT reduction. Instead, we decided to remove *all* application-level computations and reduce our benchmarks to only the necessary code to set up the desired interaction pattern: Only (#2) and (#3) contain a tight computation loop; all other benchmarks include no actual computation.

Removal of application code has two consequences: (1) the lack of complex application microstructure reduces the analysis time. Nevertheless, since we only removed the contents of ABBs, the application-OS interaction pattern remains untouched. (2) Our WCRT savings look worse than they actually will be in reality: For instance, if *Sys*WCET detects an unfeasible preemption in a conditional branch, the WCRT now only reflects the reduced kernel overhead, not the WCETs of the (assumingly) preempting higher-priority computation.

B. Interrupt Detection Latency

First, we analyze the system's latency for accepting hardware events that is introduced by code paths with disabled interrupts. The *d*OSEK RTOS disables interrupts during the whole kernel

```
void th_LO_computation(){/* 760 021 instrs.*/}
void th_HI_urgent() { ... }

void isr_timer() {
  counter++;
  if (counter % 3 == 0) {
    activate(th_HI_urgent);
  }
}
```

Listing 2: Computation interrupted by periodic Alarm

```
void th_HI_control() {
  activate(thread_LO_computation);
  wait_for(sig_done || sig_abort);
}
void th_LO_computation() {
   do_computation();  // 55 613 instrs.
   wakeup(sig_done);
}
void isr_abort() { wakeup(sig_abort); }
```

Listing 3: Abortible Computation

TABLE I: Interrupt Detection Latency and WCRT Estimates.

[instructions]	Detection Latency	*Worst-Case Response Time*		
		*Sys*WCET	Accumulated	Observed
Activate w/o Disp.	224	318	318	123
Activate w/ Disp.	224	596	596	395
Terminate w/ Disp.	224	593	601	398
Wait & Wakeup.	233	607	610	436
Interrupt w/ Dispatch.	396	464	466	339
#1: TMR	206	3319	3319	2893
#2: Alarm	485	765 716	766 733	764 785
#3: Aborted Comp.	799	56 340	60 425	55 738
#4: Signal Gathering	771	5626	6286	1168
#5: Flight Control	771	9279	10 057	2261
#6: Remote Control	771	9768	10 541	790

TABLE II: IPET Complexity and Run Time

	STG		IPET Problem		
	ABBs	States	Vars	Constr.	Run Time
Activate w/o Disp.	3	3	146	264	96 ms
Activate w/ Disp.	6	6	271	487	105 ms
Terminate w/ Disp.	6	6	266	478	131 ms
Wait & Wakeup.	6	6	232	416	124 ms
Interrupt w/ Sched.	3	12	155	290	99 ms
#1: TMR	9	9	248	458	94 ms
#2: Alarm	5	13	319	594	232 ms
#3: Aborted Comp.	9	35	601	1088	295 ms
#4: Signal Gathering	33	9506	16 269	30 432	14.72 s
#5: Flight Control	55	7690	16 528	30 666	161.56 s
#6: Remote Control	63	4608	12 987	26 849	92.57 s

execution and during the processing of (generated or user-defined) ISRs. Since *d*OSEK generates a customized kernel, we have to calculate the latency for every benchmark according to Section IV-D.

In Table I, we show the maximal latency interrupt blocks delay an ISR. Since all (#4-#6) benchmarks are included in one system image, their detection latency is equal; this also applies to the first three micro benchmarks.

The large latency for (#3) can be explained by the execution pattern described in Section IV-D: At most, it takes $T_{d\to s} = 527$ instructions from disabling interrupts to a context switch, and $T_{s\to e} = 272$ instructions afterwards. Combined, the latency is larger than the longest continuous code path without a context switch ($T_{d\to e} = 625$).

C. Worst-Case Response Time

In columns 2–4 of Table I, we compare the manually accumulated WCRTs to the results of *Sys*WCET for each benchmark. Furthermore, we execute the systems in a simulator and give the longest processing time we could observe as an under-approximation.

For the accumulated WCRT analysis, we used the following information about the benchmarks: thread priority, minimal IATs, and the *activates* relation between threads. We use `platin` [27] to calculate function WCETs and combine them manually into a WCRT according to the given information [4].

Executing all benchmarks and counting the instructions in the PATMOS simulator `pasim` measured the longest observed processing time. Since the construction of a worst-case event sequence for system-level benchmarks is hard, the observed processing time is only an under-approximation.

For the micro benchmarks, we see nearly no difference for the accumulated and the *Sys*WCET approach, since these scenarios include no complex thread–RTOS interactions that are exploitable by *Sys*WCET. Nevertheless, the full automation and the whole-system view of our approach reduces the risk of missing out code and under-approximating the WCRT.

The TMR (#1) benchmark also reveals the exact same WCRT in both analysis methods. In both cases, the calculated WCRT is 14.73 percent larger than the observed time.

For the alarm benchmark (#2), both analyses assume 8 timer interruptions. Our 1017 instructions tighter WCRT arises from the alarm handling. The *Sys*WCET toolchain automatically takes the alarm and timer periodicities from the RTOS configuration into account and thereby visits the alarm-expire path within the kernel only on every third timer interrupt. Furthermore, if we subtract the computation (99.26 %), which is needed to set up the benchmark structure, the *Sys*WCET bound is 15.15 percent tighter.

In the aborted-computation benchmark (#3), our approach reveals the correct control flow: After the computation finishes and before completion is signaled (between line 6 and 7 in Listing 3), the interrupt aborts the computation thread. Therefore, not only the `wakeup()` has to be taken into account, but also the interrupt-activation overhead. Again, without the computation WCET (line 6), we get a 84.89 percent tighter WCRT for application-structure code.

The tighter WCRT bounds for the *I4Copter* benchmarks (#4-#6) arise from the interaction of application and the operating system. Depending on the application's control-flow, different syscalls are issued that influence the scheduling and make ABBs, which include syscalls themselves, reachable or unreachable. Compared to the accumulated WCRT, we find a better bound in all three benchmarks ($-10.5\,\%$,$-7.74\,\%$,$-7.33\,\%$). Since the actual worst-case event sequence is hard to trigger, the discrepancy between calculated and observed response time is larger for this benchmark class.

D. Analysis Complexity

In Table II, we give an overview on the complexity of our benchmark scenarios and the *Sys*WCET analysis. The number of ABBs that are referenced from the STG denote their *static* structural size, while the number of STG states accounts for the *dynamic* structural size that arises from the interaction of application, RTOS, and environment. In all cases, there were at most 25 percent more STG transitions than states.

Both, static and dynamic size affects the number of variables and constraints required to express the problem as an IPET. The large number of states for the (#4-#6) benchmarks arises from the indeterminism introduced by interrupt sources, which can trigger in each computation block.

For the run-time measurements, we ran the *Sys*WCET analysis on a 16-core Intel E5-2690 machine with a processor speed of 2.90 GHz. As ILP solver, we utilized the industrialstrength `gurobi` solver and measured its run time. After we had initially very diverse run times ranging from a few minutes to several hours, we changed the `MIPFocus` configuration, as suggested by `gurobi` manual to 1. This switch influences the weight `gurobi` puts on its different solving heuristics.

VII. Related Work

Traditional analysis approaches to compute worst-case response times [4, 5, 6] usually treat the impact of the operating system as constant overhead, which is pessimistically added in a deferred step to each thread's WCET.

In contrast to approaches such as the real-time calculus [38] or the SymTA/S approach [16], *Sys*WCET allows handling events in a context-sensitive way. That is, as shown in the evaluation of context-sensitive alarms (see Section VI), the flow-sensitive knowledge allows for a reduction in pessimism of the worst-case response time.

The usage of DAG-based task models [34] and their connection to OpenMP [40] have recently gained attention for real-time scheduling. DAG-based task models explicitly express dependencies between tasks in analogy to the *Sys*WCET approach. However, in contrast to our approach, these approaches do not consider the influence of external interrupts within the same problem formulation.

To our knowledge, the *Sys*WCET approach first solves the problem of whole-system response-time analysis by providing a common ILP formulation of multiple threads, ISRs, and their scheduling across the RTOS. However, there has been much effort on analyzing the WCET of operating systems [24] in order to provide end-to-end response times:

Blackham et al. proposed a context-aware WCET analysis of the seL4 microkernel [10]. From the WCET between interrupt-enabled preemption points, they could give an upper bound for the interrupt-detection latency of seL4.

Lv et al. presented a WCET analysis of the μC/OS-II real-time kernel [23]. In their work, they analyzed the WCET of each syscall, which is then added to the respective thread's execution time. A similar approach to determine worst-case response time of tasks by analyzing the RTEMS operating systems was presented by Colin and Puaut [12].

However, decoupling the analyses of application and RTOS code can lead to significant over-estimations. In contrast to all this existing work, *Sys*WCET respects application logic, as well as the scheduling semantics (i.e., priority of threads, lock protocols) in order to compute WCRTs of whole applications.

Schneider also stated that RTOSs cannot be analyzed without considering the application and vice versa [32]. He suggested a semi-automatic approach to combine schedulability and WCET analysis of applications running on top of OSEK [31]. His framework consists of a multi-stage process, where interfering threads are iteratively added to the response time of the threads.

In contrast, our approach avoids the separate consideration of preemption costs in order to provide a higher degree of integration by a single, context-sensitive representation of all program flows in the whole real-time system.

In the model-checking community, Waszniowski and Hanzalek [41] described OSEK systems and the application logic with timed automata and proved properties like deadlock freedom and upper bounds for the WCRT. Although their WCRT analysis considers the application's microstructure and the RTOS semantics, again, their approach is two-tiered and needs WCET information about each basic block in the system. Furthermore, the inner structure of the RTOS is lost in abstraction. However, their approach could be combined with *Sys*WCET to derive further system-wide flow facts regarding interrupt occurrences.

VIII. Discussion & Future Work

*Sys*WCET provides an integrated view on RTOS, scheduling, and application code, and, thereby, exposes unique possibilities and benefits for end-to-end analyses in *real-time system* (RTS). However, on the other side, there are also several limitations. Both shall be discussed in the following.

A. Integration of More Complex Hardware Models

As pointed out in Section II, our current implementation assumes a (sufficiently) time-predictable architecture, such as an ARM Cortex M0+. We do not yet consider inter-instruction effects (e.g., pipelining) or caching behavior across threads, as the focus of this work is on RTOS-induced pessimism in WCRT analysis. However, in general the *Sys*WCET approach provides a coherent view on possible execution paths of a system. We assume that this path-sensitive knowledge can be combined with analyses of more complex hardware platforms and improve their results regarding both inter-thread dependencies and cache-related preemption delays [11, 37].

1) Inter-thread Dependencies: Flow-sensitive analyses follow the rationale that the longer the execution history the more information is available to determine and refine analysis results [22]. Since the STG captures all possible execution paths throughout the system in a context-sensitive way, including preemptions and thread dependencies, it is ideal as fundamental representation for RTOS- and scheduling-aware flow analyses. For example, since all instructions are covered by the STG, pipeline analyses across threads become more precise.

2) Cache-related Preemption Delays: The STG maintains all potential transitions from threads to preempting interrupts. This information is essential to give bounds on the delay related to the preemption (i.e., cache-related preemption delay, CRPD). Additionally, the information about memory blocks accessed inside the preempting thread allows refining the CRPD [11]. Knowledge about preempted and preempting tasks as well as their used instruction and data memory blocks is inherent to the STG. We believe that the usage of the STG is promising for further research on whole-system timing analyses.

B. Possible Limitations

The *Sys*WCET toolchain provides developers with an automated means to analyze the WCRT between any two points in the control flows of an OSEK-based RTS – even across threads, ISRs, and resources using the PCP. Consequently, many infeasible system execution paths (c.f. Section IV-A) are detected to tighten the WCRT. Nevertheless, the resulting

STG may still contain paths that become infeasible due to thread-local control-flow decisions. However, existing data-flow sensitive path-refinement techniques [8, 9, 19] are directly applicable to the STG to enhance the WCRT analysis. This is a topic of further research.

Our approach reaches its limits if the analyzed path transitions through a sleep state (i.e., wait for an interrupt signal). Since such sleep states are equivalent to an idle loop with an unknown number of executions within the kernel, the IPET becomes unbounded. To restrict such unboundedness, analysis results of precedence constraints [4] (e.g., expiration of timer alarm) can be attributed to our IPET problem formulation as additional constraints. To overcome this, we would need to model the *maximal* IAT of each interrupt source. However, we cannot derive a minimal interrupt frequency from T_{WCRT} as easily as the maximal interrupt frequency: Only the time *before* the IRQ accounts for the maximal IAT, but IPETs describe execution frequencies and not execution paths. A pessimistic over-approximation is possible, if we restrict the idle time with the maximal IAT. However, we share this limitation with the compositional approach and, moreover, believe that such idle scenarios typically do not occur in safety-critical paths. *Sys*WCET's demands on RTOS semantics and application structure (see Section II) may restrict a broader applicability of our approach: In essence, the automatic STG extraction currently works only for fixed-priority systems where all application–OS interaction (i.e., syscalls) is explicit, that is, does not occur via (nontrivial) function pointers. For real-time systems with an earliest deadline first scheduler, or any other scheduler that performs dynamic priority assignments, we are probably not able to construct a (reasonably sized) STG, as the explicit enumeration of all system states and transitions (cf. Section III-B) will become intractable.

However, fixed-priority scheduling is prescribed by all relevant industry standards, including OSEK/AUTOSAR [26, 7], ARINC 653 [2], μITRON [36], and POSIX.4, so in many areas this restriction is not a fundamental limit. The same holds for the static application structure, as syscalls are explicit in real-world applications and the use of function pointers is discouraged anyway in MISRA-C [25].

C. Scalability

With eleven threads, three alarms, and one ISR the *I4Copter* benchmark is relatively small, but systems of that size are not uncommon in some domains, such as automotive. Nevertheless, especially for systems with many interrupt sources, the size of the resulting STG may become intractable, as in theory each interrupt request forks the STG in every computation block. Our *explicit* enumeration may lead to an exponential number of STG states. We see three directions to improve on this issue: (1) more expressive STG semantics, (2) more system information, and (3) merging of similar STG states.

Currently, we use an explicitly enumerated STG where each edge expresses that one transition is possible. If we enrich the expressive power of such transitions to include testing and setting of state variables, we should be able to reduce the number of states significantly without sacrificing precision.

Furthermore, in real-time systems, interrupts are rarely *totally* unpredictable: In practice, we already constrain their occurrence by minimal inter-arrival times, control-flow dependent preconditions (e.g., the send-buffer-empty interrupt cannot occur before `send()`) and other flow facts. If we provide more of these facts to the analysis, more interrupt paths can be eradicated already during the STG construction.

Another approach to tackle the size problem would be to deal in analysis precision for run time: Many states in the STG execute the same ABB. If we merge such states into a single one, we also get a whole-system view on the possible control flows, which is known as the *global control-flow graph* (GCFG) [14]. However, the GCFG restricts the possible flows less and, therefore, allows more infeasible paths. Nevertheless, we could construct a hybrid between STG and GCFG by using a more restricted merging predicate (e.g., merging all states that have the same ready list and execute the same ABB) to achieve a more suitable trade-off.

D. Influence of ABB Size

In Section III-A, we chose our computation ABBs to be regions of maximal size. This choice reduces the number of STG states and, therefore, fosters the analysis' scalability. With our basic processor model, this choice does not influence the WCRT bound: in the worst case, a interrupt always occurs after the last instruction of a computation region, regardless of the number of ABBs that are used to partition it.

However, with more complex hardware models the ABB-size choice might worsen the WCRT bound: As described in Section IV-C3, we assume the worst preemption delay at outgoing interrupt transitions, even if that situation occurs in the middle of the ABB. In such cases, we account for the complete ABB execution *and* its worst-case preemption delay. One potential solution to this issue would be to split ABB regions with high preemption delays.

E. Benefits of the Approach

Besides the WCRT analysis, the integrated IPET formulation with cross-kernel flow facts lifts many other analyses on the whole-system level. Locks become easy to analyze: They are folded away in the process of STG construction according to the semantics of the underlying lock protocol (cf. Section IV-A).

Another option is to instruct the solver to calculate the minimum instead of the maximum objective, thereby we can get lower bounds for best-case response times. For benchmark (#3), the abort interrupt would trigger right before the control thread goes to sleep and after the computation thread was activated. In this case, the system executes at least 447 instructions.

On the mere control-flow level, we can express infeasible paths and other flow facts that span over threads, ISRs, and the kernel code. For an ABB that executes a syscall, the STG references the exact kernel entry point. Hence, we do not have to consider the longest path through the kernel, but can handle every syscall site individually. Currently, we use only information about the entry point and the structure of the STG. In the future, we plan to derive further flow facts for the kernel code from the detailed state information.

IX. Conclusion

With *Sys*WCET, we presented the first integrated, non-compositional WCRT analysis for fixed-priority, statically configured real-time applications including the RTOS. We use an operating-system state-transition graph to capture all interaction between threads, ISRs, and the RTOS. This control-flow sensitive information is encoded, together with information about periodicity and minimal inter-arrival time of interrupts, as an IPET problem and connected to IPET fragments of application and kernel code. Our implementation based on the *d*OSEK RTOS generator and the `platin` WCET analyzer covers the complete feature set of OSEK ECC1, including PCP-based locks, event-based thread synchronization, alarms, and interrupts. It provides automated means to obtain tight WCRTs between any two points in the real-time system. In our evaluation with a real-world quadrocopter flight-control application, we achieve up to 10.5 percent better WCRT bounds than with the current state of the art.

Acknowledgments

The authors thank the anonymous reviewers and our shepherd for their feedback and tremendous help to improve content and style of this paper. This work has been supported by the German Research Foundation (DFG) under grants no. LO 1719/1-3, no. SCHR 603/9-2, no. SCHR 603/13-1, and the SFB/Transregio 89 "Invasive Computing" (Project C1).

The source code of SysWCET is available at:
`https://gitlab.cs.fau.de/syswcet`

References

[1] H. Almatary, N. Audsley, and A. Burns. "Reducing the Implementation Overheads of IPCP and DFP". In: *RTSS '15*. 2015.

[2] AEEC. *Avionics Application Software Standard Interface (ARINC Specification 653-1)*. ARINC Inc, 2003.

[3] ARM Limited. *Cortex-M0+ Technical Reference Manual*. 2012.

[4] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. "Applying new scheduling theory to static priority pre-emptive scheduling". In: *Software Engineering Journal* 8.5 (1993).

[5] N. Audsley, A. Burns, and A. J. Wellings. "Deadline monotonic scheduling theory and application". In: *Control Engineering Practice* 1.1 (1993).

[6] N. Audsley, K. Tindell, and A. Burns. "The end of the line for static cyclic scheduling". In: *In Proceedings of the 5th Euromicro Workshop on Real-Time Systems*. 1993.

[7] AUTOSAR. *Specification of Operating System (Version 5.1.0)*. Tech. rep. Automotive Open System Architecture GbR, 2013.

[8] B. Blackham and G. Heiser. "Sequoll: A framework for model checking binaries". In: *RTAS '13*. 2013.

[9] B. Blackham, M. Liffiton, and G. Heiser. "Trickle: Automated Infeasible Path Detection Using All Minimal Unsatisfiable Subsets". In: *RTAS '14*. 2014.

[10] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. "Timing analysis of a protected operating system kernel". In: *RTSS '11*. 2011.

[11] C. Burguière, J. Reineke, and S. Altmeyer. "Cache-related preemption delay computation for set-associative caches–pitfalls and solutions". In: *OASIcs-OpenAccess Series in Informatics*. Vol. 10. 2009.

[12] A. Colin and I. Puaut. "Worst-case execution time analysis of the RTEMS real-time operating system". In: *ECRTS '01*. 2001.

[13] C. Dietrich, M. Hoffmann, and Lohmann D. "Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis". In: *TECS* 16 (2 2017).

[14] C. Dietrich, M. Hoffmann, and D. Lohmann. "Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems". In: *LCTES '15*. 2015.

[15] Florian Franzmann, Tobias Klaus, Peter Ulbrich, Patrick Deinhardt, Benjamin Steffes, Fabian Scheler, and Wolfgang Schröder-Preikschat. "From Intent to Effect: Tool-Based Generation of Time-Triggered Real-Time Systems on Multi-Core Processors". In: *ISORC '16*. 2016.

[16] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. "System level performance analysis – the SymTA/S approach". In: *IEE Proceedings–Computers and Digital Techniques* 152.2 (2005).

[17] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. "dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel". In: *RTAS '15*. 2015.

[18] B. Huber, D. Prokesch, and P. Puschner. "Combined WCET Analysis of Bitcode and Machine Code Using Control-flow Relation Graphs". In: *LCTES '13*. 2013.

[19] J. Knoop, L. Kovács, and J. Zwirchmayr. "WCET Squeezing: On-demand Feasibility Refinement for Proven Precise WCET-bounds". In: *RTNS '13*. 2013.

[20] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *CGO'04*. 2004.

[21] Y.-T. S. Li and S. Malik. "Performance analysis of embedded software using implicit path enumeration". In: *ACM SIGPLAN Notices*. Vol. 30. 11. ACM. 1995.

[22] Y.-T. S. Li, S. Malik, and A. Wolfe. "Cache Modeling for Real-time Software: Beyond Direct Mapped Instruction Caches". In: *RTSS '96*. 1996.

[23] M. Lv, N. Guan, Y. Zhang, R. Chen, Q. Deng, G. Yu, and W. Yi. "WCET Analysis of the mC/OS-II Real-Time Kernel". In: *CSE '09*. 2009.

[24] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. "A survey of WCET analysis of real-time operating systems". In: *ICESS '09*. 2009.

[25] *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*. 2004.

[26] 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, 2005.

[27] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. "The T-CREST Approach of Compiler and WCET-Analysis Integration". In: *SEUS '13*. 2013.

[28] P. Puschner and A. Schedl. "Computing Maximum Task Execution Times: A Graph-Based Approach". In: *Real-Time Systems* 13 (1997).

[29] C. Rochange. *WCET Tool Challenge 2014*. Talk held at WCET '14. 2014.

[30] Fabian Scheler and Wolfgang Schröder-Preikschat. "The Real-Time Systems Compiler: migrating event-triggered systems to time-triggered systems". In: *SPE* 41.12 (2011).

[31] J. Schneider. *Combined schedulability and WCET analysis for real-time operating systems*. Shaker, 2003.

[32] J. Schneider. "Why you can't analyze RTOSs without considering applications and vice versa". In: *WCET '02*. 2002.

[33] M. Schoeberl et al. "T-CREST: Time-predictable multi-core architecture for embedded systems". In: *JSA* 61.9 (2015).

[34] M. A. Serrano, A. Melani, M. Bertogna, and E. Quinones. "Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions". In: *DATE '16*. 2016.

[35] *SysWCET Project Repository*. https://gitlab.cs.fau.de/syswcet. 2016.

[36] H. Takada and K. Sakamura. "μITRON for Small-Scale Embedded Systems". In: *IEEE Micro* 15.6 (1995).

[37] C. Tessler and N. Fisher. "BUNDLE: Real-Time Multi-Threaded Scheduling to Reduce Cache Contention". In: *RTSS '16*. 2011.

[38] L. Thiele, S. Chakraborty, and M. Naedele. "Real-time calculus for scheduling hard real-time systems". In: *ISCAS '00*. Vol. 4. IEEE. 2000.

[39] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. "I4Copter: An Adaptable and Modular Quadrotor Platform". In: *SAC '11*. 2011.

[40] R. Vargas, E. Quinones, and A. Marongiu. "OpenMP and timing predictability: A possible union?" In: *DATE '15*. 2015.

[41] L. Waszniowski and Z. Hanzalek. "Formal Verification of Multitasking Applications Based on Timed Automata Model". In: *Real-Time Systems* 38.1 (2008).