Appears in: Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium (RTAS'19)
Montreal, Canada, 16–18 April 2019

Proving Real-Time Capability of Generic Operating Systems by System-Aware Timing Analysis

Simon Schuster, Peter Wägemann, Peter Ulbrich, Wolfgang Schröder-Preikschat
Department of Computer Science, Distributed Systems and Operating Systems
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—The static timing analysis of universal real-time operating systems (RTOS) with generically implemented services requires application and system-context–specific knowledge (e.g., number of currently active tasks) to bound overheads. However, due to the missing notion of OS semantics, contemporary timing analysis tools are unable to exploit such information, resulting in failing or overly pessimistic analysis. To tackle this issue, we present our System-wide WCET Analyses framework (SWAN). SWAN's heart is PLATINA, a parametric source-level annotation language that facilitates the expression and propagation of context information from the application over the OS down to the machine-code level. Through the expression of semantic inter-dependencies in a unified and reusable way, analysis pessimism is significantly reduced, as we demonstrate by case studies on FreeRTOS, Linux, and a real-world flight-control system.**

Just as important as our system-aware timing analysis is the tool support for its practical usability. Therefore, we augmented SWAN by a powerful interactive visualization and annotation environment. This enables developers to quickly identify context-dependent spots that require annotation and thus to cope with large implementations associated with universal RTOSs. Eventually, SWAN allows determining if a generically implemented system is real-time capable and thus timeliness is guaranteed.

I. INTRODUCTION

The worst-case response time (WCRT) is a vital temporal property of tasks with hard deadlines. Its evaluation demands a sound worst-case execution time (WCET) analysis of all implementation artifacts. Accordingly, static timing analysis of real-time applications is a well-established field with commercially available tools [1] that yield precise bounds on the WCET in practice. They, nonetheless, rely on a mostly static program structure, which is inherent to safety-critical applications, to accurately infer data and control flows, for instance to bound loops and resolve function pointers.

However, tasks are typically executed in a broader system's context using a real-time operating system (RTOS). Consequently, the RTOS implementation has to be subject to the same temporal analysis. The overhead induced by the RTOS is commonly treated as constant and pessimistically added in a deferred analysis step to each task's WCRT [2], [3]. The reason is that system calls and preemptive scheduling intermit with data and control flows and thus necessitate a dedicated analysis of kernel paths. The use of such a compositional approach is consequently only feasible when given a static setting, which means that operating-system (OS) implementations are tailored to a specific set of statically decidable application parameters. A good example of this is the well-tried concept of tailored library OSs, such as OSEK-OS variants [4].

A relatively new development in safety-critical real-time systems is the employment of dynamic runtime environments, such as Real-Time Linux [5], [6], which corresponds more to a general-purpose OS and promotes code reuse by generically implemented algorithms and interfaces. A reliable indicator of this development is the future *Adaptive AUTOSAR* standard [7], which addresses the increasing complexity of driver-assistance and autopilot functions in vehicles. Such RTOSs feature a dynamic system-call layer, runtime reconfigurability, and facilities that are designed to serve numerous application scenarios. This versatility requires dynamic data structures and execution paths that cause the loss of a tailored and static OS implementation as well as the decoupling of control flows within the application and the OS kernel. This, in turn, implies a likewise decoupled examination, which burdens OS overheads even more with inevitable overestimations that rise tremendously with system complexity, jeopardizing static timing and schedulability analyses of such dynamic systems [8].

The critical challenge to obtain more realistic bounds is in the contextual and system-wide knowledge, which, however, is generally inaccessible to binary-level analysis. Instead, we opt for a tailorable WCET analysis that incorporates high-level knowledge and proves real-time capabilities specifically in a given system context without losing the implementation's generic nature. Moreover, conventional iterative analysis (i.e., compilation, analysis, flow-fact annotation) quickly loses feasibility with increasing system complexity and is thus unable to handle the multitude of possible system settings. Consequently, the analysis process should scale with both application and OS kernel sizes and support the developer in verifying new and changing system settings.

In this paper, we present SWAN, a system-aware WCET analysis approach specifically designed to address said challenges in the verification of real-time capabilities of dynamic RTOSs. Its key idea is to integrate whole-system knowledge and system-dependent flow facts from different levels of the system in a generic and traceable way. Thereby, high-level semantics and flow facts previously inaccessible to a decoupled analysis become available and can be leveraged to link operating–system-level control flows and application states again. Combined with a compiler-optimization–aware timing analysis, SWAN can ultimately do both: prove real-time capabilities in a given setting and eliminate overly pessimistic constant bounds on RTOS overheads. Beyond that,

This paper has passed an Artifact Evaluation process.

SWAN's parametric analysis approach facilitates an interactive annotation and analysis process that supports developers in identifying and straightening out paths that are unanalyzable or unbounded. At the same time, our infrastructure significantly reduces the round-trip costs of iterative system analysis and thus fosters the overall scalability of proving timeliness.

In summary, this paper makes the following contributions: (1) It introduces our approach to system-aware WCET analysis of generic RTOSs and provides details on SWAN's core concepts, the notion of context-sensitive *system facts*, and its annotation language PLATINA. (2) It gives insights into the open-source prototype and its optimization-aware compiler- and timing-analysis framework. (3) It presents SWAN's interactive analysis and annotation infrastructure, which we designed to tackle large code-bases of dynamically-implemented real-time systems. (4) It evaluates the approach on two generic RTOSs and the case-study of a real-world flight-control system.

II. PROBLEM STATEMENT

We now detail two core challenges of proving real-time capabilities of generic RTOSs: (A) How to introduce system-awareness into control-flow analysis of a generic implementation and (B) how to cope with system size and complexity. Finally, we provide details on our system model.

A. Challenge #1: System-Aware Control-Flow Constraints

OS overheads and task response times heavily depend on the execution context. In Figure 1, for example, when highest-priority Task 1 issues `ResumeTask()`, the costly `reschedule()` operation is conditionally omitted as Task 1 continues execution. However, relying on universal, system-agnostic flow facts, WCET analysis of this operating-system call is forced to make overestimations and pessimistically include the expensive reschedule operation in any case. This simple example could be addressed by extending the analysis to the application code, assuming that the current and next priority can be inferred as constant values. However, such call-context–aware approaches quickly fail for complex context information and scheduling semantics. In Figure 2, Task 1 now suspends itself by `SuspendTask()`, whereby rescheduling is inevitable. In this case, the overhead depends on the number of priority queues to be probed and thus the highest priority level that is guaranteed to contain a runnable task. Such relationships are up to the *system context* and are regularly not representable by traditional (e.g., binary-level) annotations and flow facts. We see significant potential in the analysis of said system-state–sensitive control flows by incorporating semantical knowledge about the execution environment.

Analyses already exist that exploit scheduling semantics to identify infeasible paths [9], [10]. However, these high-level system analyses entail excessive interdependencies between applications and kernel and thus require global control flows to determine constraints. While these approaches work for static, tailored OS implementations, they fail for generic RTOSs as necessary global information is inaccessible to the analysis.

The resulting challenge is, therefore, to provide a unified and reusable way of formulating, passing, and evaluating system-state information between the high-level system and low-level timing analysis.

Our Approach: Our two basic concepts to improve timing analysis of real-time systems using a generic operating system kernel are (1) gathering information about the system's behavior (i.e., interaction with the operating system, scheduling semantic) and (2) propagating this knowledge across all layers of the toolchain in order to make use of it in the final WCRT estimation. For the former, we introduce the notion of *system facts* to expose information on the system state and context. Such facts may range from simple global configuration parameters over application- and kernel-specific context to high-level system semantics, for example, acquired by our whole-system analysis [9] for fixed-priority preemptive scheduling. We address the latter by PLATINA, an expressive annotation language that allows us to formulate universal constraints at the source-code level that are context-sensitively instantiated by system facts. We use an optimization-aware compiler and timing-analysis infrastructure to thereof infer tighter yet sound bounds for generically implemented systems.

B. Challenge #2: System Size & Complexity

For the WCET analysis of a small microkernel with limited complexity of around 8700 lines of C code, the authors of [2] reported substantial effort, especially, to correlate program points in the binary's control-flow graph with the operating system's source code. Considering the huge code base of generically implemented operating systems, we found the correlation complexity and thus the number of necessary annotations to grow dramatically. Moreover, annotating code on assembly level, in comparison to source-code–level annotations, is known to be more error-prone, rendering this approach inapplicable to our problem domain.

Finally, we identified another fundamental issue that only came to light when working with larger code bases: *the round-trip time for analysis, annotation, and (re)compilation*. For Real-Time Linux, for example, this can take up to several hours (cf. Section VI-D). Consequently, in such cases the common stepwise annotation approach of existing WCET analysis techniques [11] is infeasible.

Our Approach: By using the *compiler-aware code correlation* of PLATINA annotations, system facts, and flow facts from source- to assembly-code level based on sound relation graphs [12] in our code-optimizing infrastructure, we enable both the correlation and visualization of control flows and the resulting integer linear program (ILP) formulation.

Thus, the developer can quickly identify issues, insert annotations, and then judge the impact of these changes. To cope with larger code bases, we developed an *interactive analysis mode*, which enables annotation in an iterative workflow without the need for time-consuming recompilation and reanalysis. We argue that with SWAN, we reduce the necessary effort for WCET analysis of large systems to a minimum.

III. SYSTEM MODEL AND BACKGROUND

Precise knowledge about a system's execution flow and semantics is essential when analyzing universal operating

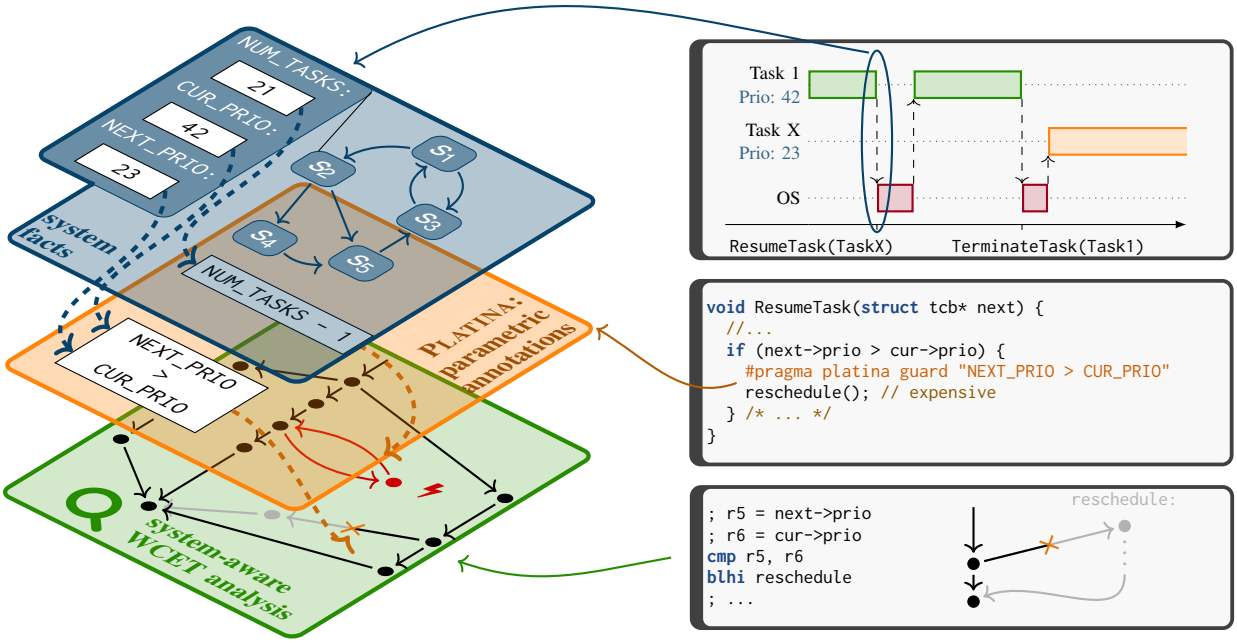

Figure 1. SWAN's information propagation: semantic context information is gathered from a semantic model (e.g. the process priorities and the associated scheduling semantics). Parametric annotations use these parametric information (i.e., system facts) to express the semantics of system-context-sensitive execution flow at the source-code level (e.g. by conditionally masking the path using a `guard` annotation). After lowering to the assembly-code level in a sound way (i.e., sematic-preserving and optimization-aware), the static WCET analysis leverages the specific system facts to exclude infeasible execution flows.

systems. RTOS standards typically provide a well-defined interface for system calls, scheduling semantic (e.g., fixed-priority), and resource handling (e.g., priority ceiling) [13].

A. System Model

Our system-aware WCET analysis of universal OSs demands the following fundamental properties: SWAN operates on the control-flow level and aims to reduce the RTOS-related pessimism in WCRT analysis. Regarding the system's structure, we assume an RTOS with fixed-priority scheduling semantics and an application setting that allows for a constant task set (i.e., the number of system objects is fixed) running on a single-core processor. Usually, timing analysis is split into the two steps of high-level (i.e., hardware-independent) program-flow analysis and low-level (i.e., target-aware) machine-code analysis. Our technique integrates into the first step of flow analysis and is thus agnostic to particular hardware properties. Yet, to void the influence of these hardware-dependent overestimations and obtain a pristine evaluation of our technique, we chose a simple processor model (see Section VI), which is commonly used when benchmarking flow analyses [14].

Although the paper's focus is on improving the aspect of path analysis in real-time systems executing on a generic operating system, we can seamlessly use existing hardware-analysis techniques [15]. Although SWAN is primarily designed for path analysis, its analysis results can also help to reduce the pessimism in hardware analysis through the system-aware knowledge of code paths: For example, the analysis pessimism of OS-induced delays, such as cache-related preemption delays (CRPD) [16], can be reduced, since scheduling knowledge excluding specific system paths (e.g., in a priority-ceiling protocol) is now available to the WCET analysis. We discuss this aspect of future work in Section VII.

B. Background on Whole-System Analysis

The definedness of RTOS standards enables automatic determination of system execution flows and leads to high-level system analyses within recent years [10], [17]. For SWAN, we build upon previous work [9] on whole-system analysis of static operating-system implementations that we extended to dynamic settings. In the following, we briefly introduce the basic concepts of said previous work on the analysis of system-wide control flows as well as the construction of an *operating-system state-transition graph* (STG) [17] and detail their application in the given context.

Figure 2 illustrates the inference of abstract system states. Its first step is to derive an implementation-agnostic representation of the interaction between tasks and the operating system and to determine regions in which the system state is invariable. We, therefore, employ the concept of atomic basic blocks (ABB) [18], which is a control-flow superstructure that subsumes one or more basic blocks and conceptually spans between state-changing system calls. This construction implies that an ABB executes atomically from a scheduling perspective. At this point, the resulting ABB graph captures a static view on the application structure.

To also incorporate the scheduling semantics and system behavior, we thereof infer the operating-system state-transition graph. 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 tasks,

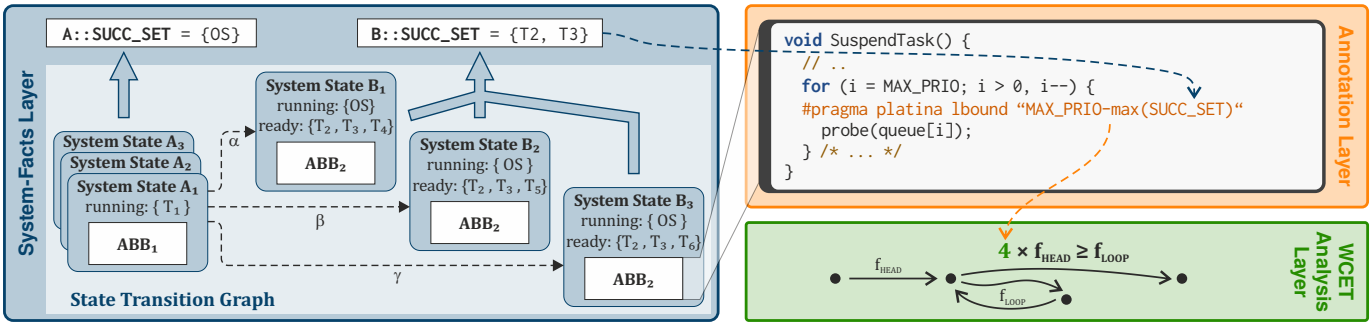

Figure 2. Multistage process of system-fact interference: (1) Static analysis partitions the system into atomic basic blocks (ABBs). In this example, Task 1 calls `SuspendTask()`, resulting in ABB_1 (app. code) and ABB_2 (syscall code). (2) A *state-transition graph* is derived that captures all possible *system states* and transitions between ABBs. (3) *System facts* are generated from the graph to make its semantic knowledge accessible. For example, a must analysis reveals the set of obligatory successors (`SUCC_SET`). This information is leveraged in the PLATINA `lbound` annotation and lowered to flow facts of the WCET analysis.

acquired resources, and the currently running control flow. The transitions express all possible execution paths through the system, including task switches. For the running example in Figure 2, the call of `TaskSuspend()` will cause the control-flow to switch from Task 1 (ABB_1) to the kernel (ABB_2). From the state enumeration, we further know that there are three potential transitions (i.e., α, β, γ) and system states (i.e., $B_{1\text{-}3}$) for the execution of ABB_2. This system-wide state enumeration conceptually combines thread-local abstract interpretation with scheduling semantics. In our previous work [9], we used this knowledge to generate tailored and thus entirely static system-call implementations for each transition that we then inlined in the application code for optimal analyzability. However, the genericness of universal RTOSs impedes this approach altogether. Instead, with SWAN we leverage our system awareness to tailor the analysis.

IV. The SWAN Approach

In this section, we detail our SWAN [19] approach on the WCET analysis of universal real-time operating systems. We therefore first provide details on our notion of system facts, how to determine these facts, and how to soundly propagate them through the analysis layers down to the machine-code level. Finally, we outline how to identify annotation points for system facts by our novel interactive annotation scheme.

A. The Notion of System Facts

Following the aforementioned limitations, we introduce the notion of *system facts*. They differ from flow facts in that they do not associate with a dedicated instruction per se, but rather apply to particular sections of the execution. System facts expose information on the system context, which is the sum of all active application states and the operating-system's state, and represent a generalized form of our previous work [9] on the abstract modeling and extrapolation of system-wide control-flows and states. In accordance with the assumed system model (cf. Section III-A) our current incarnation of system facts captures, but is not limited to, the semantics of fixed-priority scheduling along with the overall configuration of the system. In the following section, we describe how to determine these system facts.

B. Determining System Facts

We infer system facts through a multistage process: Step one and two are the ABB-graph generation and the state-transition–graph analysis described in Section III-B. The former's OS-specific frontend is already capable of determining the most basic form of system facts, which are constant configuration parameters. For example, the number of priority levels (`MAX_PRIO`) or tasks (`NUM_TASKS`). To further obtain information on the runtime system states, we leverage the latter: The STG implicitly holds all relevant information that is required to provide guarantees or invariants for specific sites of system calls. Yet, this semantical knowledge is inaccessible to the WCET analysis. Depending on the initial system state of a given call site (ABB), a system call may lead to a number of different transitions and subsequent system states. For example, in Figure 2 the `SuspendTask` call in Task 1 generated transitions α, β, and γ that lead to the system states B_1, B_2, and B_3 respectively.

To solve this ambiguity, we, therefore, generate flow-specific system facts from the STG in the third and final step. These represent a sound aggregation of said possible system states, which can then be used in PLATINA annotations. For our problem domain of fixed-priority scheduling, we found a set representation of state variables such as obligatory and potential predecessors, successors, and occupied resources as purposeful. In Figure 2, the relevant obligatory set of successors for Task 1 is captured by the system fact `SUCC_SET`, which is calculated as the intersection of the ready queues of all system states (i.e., $B_{1\text{-}3}$) that are valid for ABB_2. This can be effectively seen as a must-analysis. In this case, the set is nonempty and we can guarantee that at least tasks T_2 and T_3 will be ready in this execution path. This system fact can then be used to tailor the loop bound (f_{LOOP}) by providing a lower bound on the queue index (`max(SUCC_SET)`) and thus the number of probing steps in `SuspendTask()`. During WCET analysis, this annotation is then evaluated over the specific set of system-fact values (e.g., task priorities) and thus, a concrete flow fact (e.g., loop bound of four) can be deduced for this specific execution path and call site. Since the system-facts model can be arbitrarily extended, further

aggregations and derivatives are possible. Examples for this are the variables `NEXT_PRIO` and `CUR_PRIO` in Figure 1, which are used to exclude the reschedule in `ResumeTask()` when it is not necessary. The complete set of system-fact values extracted from the STG defines the local system context, which is the system-state information associated with this STG node.

By introducing this system awareness, we can substantially reduce the analysis pessimism. We further detail the expressiveness of our annotation language in Section V-A.

C. Propagating System Facts through SWAN's Layers

The three layers in Figure 1 illustrate SWAN's fundamental layered data flow: at the core (middle) is PLATINA, a parametric annotation language for expressing system-awareness on the source-code level. The annotations tap into a system-facts layer (top) that holds all system-state–dependent information for a specific system. Finally, to achieve a system-aware WCET analysis (bottom) operating on the machine-code level, SWAN provides a *semantic-preserving transformation* that harnesses the flow information contained in the upper layers to eventually guide the hardware-dependent timing analysis operating on assembly level. We provide a conceptual description of those steps in the following before detailing their implementation in Section V.

Annotation Layer: With PLATINA, we attack our first challenge of formulating generic flow constraints by bridging the immanent semantic gap between application and kernel analysis by propagating system-state–dependent information (i.e., system facts) to the low-level WCET analysis in an automated fashion. Hence, we provide parametric (i.e., system-aware) annotations to address three challenging symptoms that so far prevent tighter bounds on generic OS operations: (1) Annotation of indirect function calls to aid the control-flow reconstruction. This is a key enabler for the subsequent propagation of system facts. (2) State-dependent annotation of branches in conditional executions to eliminate paths that become infeasible only in certain contexts. (3) System-aware annotations to bound computations by application and configuration knowledge. These building blocks form an expressive annotation language that allows associating the semantics of individual system facts with the actual execution flow. To reflect the complexity of control flows within the OS, PLATINA's annotation language allows programmers to combine multiple system facts and even reuse expressions by defining custom facts or functions. Furthermore, to foster long-term maintainability of annotations, we integrate them at the annotated program point within the code (e.g., C++) in a backward compatible manner.

System-Facts Layer: The system-facts layer parametrizes PLATINA's annotations by supplying context-specific values for referenced system properties. Therefore, this layer provides and stores static and application-specific configuration knowledge as well as system-state–dependent information, which we collectively refer to as system facts: the atomic entities on which the annotation language operates (cf. Section IV-A). Higher-level analysis approaches to incorporate scheduling semantics [9], [10] can engage at this level to derive and enrich the analysis with further system facts. During analysis, for each calling context of a given kernel operation, this layer is queried to determine the system-state–specific set of system-fact values, which are then used to calculate tight bounds.

WCET-Analysis Layer: The actual timing analysis is conducted on the machine-code level. From the high-level program-flow analysis, we can extract a partial STG [9] that spans all possible program paths between the relevant set of starting and terminating ABBs of the operation under analysis. Furthermore, the high-level analysis provides specific system-fact values for all individual system states along these execution paths. By evaluation, these specific system-fact values are then combined with the annotation expressions and lowered down to plain flow facts describing the given system state. Each occurrence of an ABB is thus instantiated within the analysis and specialized to its specific context [20]. Here, we rely on optimization-aware compilation using control-flow relation graphs [12] to transform the individual flow information from the bitcode level of the annotation language to the machine-code–level control-flow graph. Consequently, our lowering preserves annotation semantics, which ultimately allows us to infer specific facts on the execution flow from the contexts of system calls. For example, loop bounds that are specific to the number of runnable tasks or paths that are infeasible in the given context, such as excluding `reschedule()` in Figure 1. That way, we obtain system-aware and thus more accurate bounds for OS overheads while proving if the OS is real-time capable in the given setting.

D. Identifying Annotation Points for System Facts

While we have thereby created a truly system-aware WCET analysis and therefore successfully solved our first challenge from Section II-A, which means that the problem of whole-system analysis of operating systems is resolved on a conceptual level, this solution on its own is hardly practical for large legacy code bases due to the lengthy, unstructured, and labor-intensive annotation process involved as outlined in our second challenge (Section II-B). Consequently, the system facts and their knowledge need to be linked to the actual RTOS implementation, since flow-analysis tools are not capable of determining semantic knowledge based on the implementation, which is generally an unsolvable problem [21]. Thus, reconsidering the example in Figure 1, the developer has to provide the annotation what paths are feasible in which context and exclude infeasible paths via `guard` expressions, leveraging the system facts provided by these high-level flow analyses. To facilitate and ease this unavoidable process of manually annotating code and expressing best application-specific knowledge, we developed an *interactive annotation and visualization environment* (see Section IV-D).

Here, the difficulty actually lies within two separate yet related issues: (1) Navigating the source to effectively place the annotations and (2) providing faster feedback by cutting the cycle time between adding a new annotation, compiling the source code, and actually obtaining a bound or the next

problematic program point that requires the programmers attention. When performing a WCET analysis at the machine-code level, problematic program points appear as machine-code–level basic blocks whose execution frequency cannot be bounded. Mapping this back to high-level loops, and thereby effectively decompiling the binary, to find suitable loop bounds and annotating them, is a very tedious process. To make matters worse, the unbounded operation and the relevant annotation point do not necessarily coincide: In the case of system-aware infeasible paths, the problematic construct and the optimal annotation spot can easily be separated several levels deep into the call tree. Furthermore, when not tackling unbounded but merely unnecessarily pessimistic operations, there is no guidance at all into what program points would be beneficial to annotate. We tackle this problem by providing an interactive visualization of the analysis results that exposes the calculated execution frequencies and costs of the different program components in a fashion akin to a global control-flow graph across function boundaries, and partial mapping of the machine-code–level program points to source-level based on the debug information contained. By visually highlighting problematic program spots, we further guide the programmer through the complete annotation process in a problem-centric manner. A more detailed description along with an example of this visualization is given in Section V-C1. Furthermore, to make the annotation process even more time-efficient and interactive, we have eliminated the recompilation step from the analysis by providing the programmer with the capability of directly annotating the intermediate representation of the program without recompilation. When finished, the annotations can be integrated back into the source code in a guided (i.e., semi-automatic) manner. Additionally, this enables to interactively provide system facts and inspect the WCET result without recompilation: the same binary can be evaluated for arbitrary system facts and in multiple system contexts, which retains scalability and usability even for large implementations.

In summary, SWAN provides sound propagation of high-level, semantic, system-aware information throughout the compilation and analysis processes down to the level of the static WCET analysis. The parametric annotation mechanism allows obtaining tailored, system-aware timing bounds even for generic OS kernels. Thus, SWAN proves their real-time capability in a specific context with higher analysis accuracy and therefore tighter bounds compared to traditional decoupled WCET analysis. The visualizer and the interactive annotation environment further optimize the practical aspects of the annotation process by interactively guiding the programmer through the analysis in a problem-centric fashion.

V. IMPLEMENTATION & TOOL SUPPORT

We based SWAN on the T-Crest toolchain [22]–[24] and its WCET analyzer PLATIN, which we extended to implement the concepts described in the previous section. In the following, we detail two vital implementation aspects: (A) Cross-layer integration of our system-aware annotation language PLATINA and (B) our interactive visualization and analysis environment.

A. PLATINA *Annotation Language*

We designed PLATINA annotations to enable a universal formulation of system-aware constraints that can be parameterized by system facts. Therefore, we opted for (1) source-level annotations that seamlessly blend into common programming languages and toolchains, (2) a tailored language for the evaluation of complex system-state–specific expressions, and (3) a compiler-aware lowering of the resulting facts.

1) Language Integration: To minimize the analysis effort, we designed PLATINA annotations to interfere with neither non-real-time parts nor legacy compiler toolchains. Therefore, we introduced special pragma directives with the `#pragma platina` prefix, which are ignored by legacy compilers according to specification (e.g., for C99 [25]). Thus, we can ensure backward compatibility and easy integration with existing RTOSs and development processes. We identified three fundamental annotation types that are required to tackle the challenge of system-aware control-flow constraints:

```
#pragma platina (callee|guard|lbound) "expression"
```

First, `callee` annotations enable control-flow reconstruction by specifying call targets of indirect function calls. Moreover, `guard` annotations allow excluding paths that are irrelevant or infeasible only in a specific system context. Finally, `lbound` annotations facilitate to specify and tighten loop bounds in a system-state–dependent manner. In all cases, actual values are obtained by evaluating the expression over the set of system-fact values that are valid in the given system context, thus introducing system-awareness into our analysis. The result is a list of potential call targets (`callee`), a boolean indicating feasibility (`guard`), and the loop bound (`lbound`), respectively.

The final challenge is the annotation's placement within the source code to ensure a correct and traceable mapping to the annotated construct throughout the subsequent compilation and analysis steps. For the `callee` annotation, this is straightforward: it is placed at the line preceding the indirect call. In contrast, for `lbound` and `guard` this simple approach (i.e., ahead of `for` or `while` statements) conflicts with unstructured loops (e.g., by the `goto` statement) or macros (e.g.,`for_each_class`). However, both constructs are extensively used in critical parts of Linux, for example in the scheduler:

```
again:
  #pragma platina lbound "..." // goto-again loop
  for_each_class(class) {
    #pragma platina lbound "..." // for_each_class
    #pragma platina callee "[pick_next_task_rt,␣...]"
    p = class->pick_next_task(rq, prev); // ...
    if (p == RETRY_TASK) goto again;
  }
```

To solve these issues, we execute the following labeling scheme to firmly link PLATINA annotations with the control flow: an `lbound` annotation always correlates with the nearest loop whose flow passes the pragma instruction and specifies its iteration count. Likewise, a `guard` annotation marks all control-flow paths that pass the pragma's program point. That way, PLATINA annotations are universally applicable, backward compatible, and invariant of the syntactic program structure.

2) Expression Language: The key element to link system facts with a specific system context and to characterize their relation to the execution semantics is the ability to formulate and evaluate compound expressions. Therefore, we developed PEACHES, a tailored programming language that operates on the specific system-fact values produced by the high-level analysis, which are represented as global constants, and allows for complex calculations on those to derive system-state–specific flow facts for any given system context. These constant results then serve as inputs to the timing analysis.

As with the annotation types, we gave PEACHES sufficient expressiveness while keeping our approach simple. In addition to arithmetic and logical expressions, if-then-else constructs and built-in set manipulation, the language further enables the user to define custom helper functions and preaggregations of commonly used expressions on system facts. Yet, we tailored the language semantics of these constructs to this specific problem domain: For instance, we deliberately made PEACHES lack Turing completeness by limiting the types of admitted recursion in the annotation expressions. Based on the concept of total functional programming [26], [27], this ensures termination of the expression evaluation. Likewise, calculations with side effects are problematic as they may result in indeterministic results due to the unspecified evaluation order of compilation units. Therefore PEACHES's paradigm is based on a functional programming language (we borrowed syntax from Dhall [28], and Haskell [29]), which ensures an evaluation that is free from side effects and therefore invariant in evaluation order. Furthermore, our expressions and values are strongly and statically typed to facilitate type checking and validation ahead of the analysis. This eliminates the need for run-time error checking and handling during analysis, which is an enabler for our interactive analysis in Section V-C.

Notwithstanding said restrictions and simplifications, we made PEACHES extensible to meet future needs. For example, we support the definition of custom functions and the reuse of expressions. Likewise, the type system can be easily extended.

B. Compilation and Analysis Pipeline

Finally, we have to ensure the correct lowering of the annotations to assembly level irrespective of control-flow restructuring compiler optimizations. To tackle this issue, we leveraged LLVM [30] along with the T-Crest toolchain [22]–[24] to develop a tailored compiler (currently supports ARM and Patmos) that allows supports this annotation lowering.

Figure 3 illustrates the resulting framework and workflow. During the compilation process, PLATINA annotations are, along with other compilation-specific information, extracted and expelled into PLATIN's program meta-info language (PML) [23]. The latter serves as an exchange format between compiler and static analysis. The specific extraction process depends on the annotation type: `lbound` and `guard` annotations describe system-state–dependent flow information (i.e., system-state–dependent loop-bounds and infeasible paths) at the source-code level. Here, the challenge is in the ambiguous mapping of control flows on source-code and

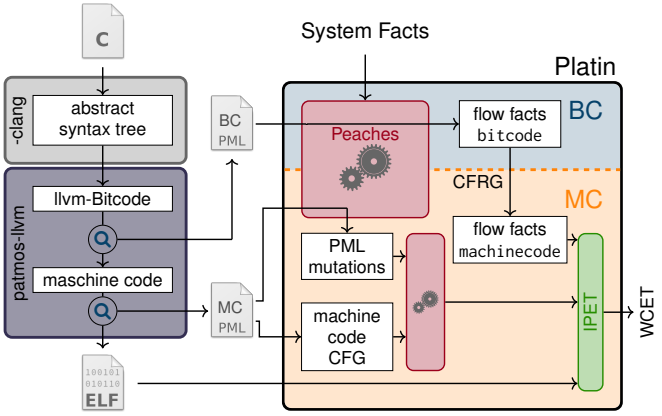

Figure 3. Visualisation of the annotation extraction in our compilation pipeline: Annotation extraction takes place at the bitcode (`lbound` and `guard`) and machinecode (`callee`) level and are emitted to PML files along with control-flow graphs and the CFRG, evaluated over the set of system facts and then integrated into the analysis as regular flow facts and CFG mutations.

binary level. SWAN leverages control-flow–relationship graphs (CFRGs) [12] to relate flows between high (e.g., llvm-bitcode) and low-level (e.g., machine code) control-flow graphs and to safely transform numeric flow facts over compiler optimizations. Consequently, we extract said annotations before compiler optimizations and perform the actual lowering within the static analysis. However, the latter requires constant numeric flow facts. Therefore, SWAN derives those flow facts by evaluating annotation expressions over system facts that apply to the given system context. For example, the `guard`-annotation in Figure 1 will evaluate to false and thus provide a flow fact that forces the execution frequency of the `reschedule()` operation to zero in the final IPET formulation. Likewise, in Figure 2 the system facts `MAX_PRIO` and `SUCC_SET` are evaluated to specific values for the current system context by calculating the guaranteed bound on the succeeding task's priority. The resulting loop bound of four is then expressed as a numeric flow fact, lowered utilizing the CFRG and expelled to the IPET formulation as a binary-level flow constraint. Simply put, system facts are transformed into common flow facts during lowering. As expressions are constantly reevaluated against the current system context in each analysis, the tailoring (e.g., path elimination for `guard` annotations) is only performed when applicable in the given context, which maintains soundness.

In contrast, `callee` annotations, which specify the set of targets of an unresolved call, do not end in a numeric flow fact and thus cannot be related as such by CFRGs. Instead, `callee` annotations are conveyed as an intrinsic call preceding the actual call instruction, which is translated into a pseudo instruction in the backend. As both instructions declare unmodeled side effects and effectively constitute reordering barriers, they pass further compilation steps untouched and adjacent to each other. In the final assembly, we discard this pseudo instruction, preserve its position, and later update the machine-code–level control-flow graph in the analysis: Based on the state-specific symbol list obtained from the annotation expression, the original call site now includes the relevant set

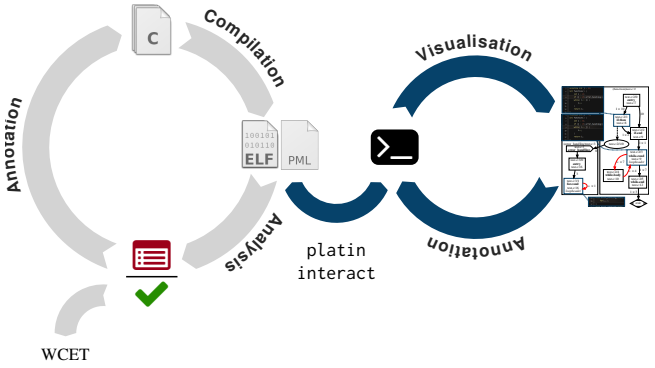

Figure 4. Interactive annotation cycle: After the initial compilation, annotations are inserted at the intermediate program representation displayed by the visualizer and later integrated back into the source code.

of successors. We then use this patched CFG as a basis for the final IPET construction.

Overall, using these two techniques, we can ensure a sound lowering of annotation expressions and system facts down to the assembly level, ready for the final WCET analysis.

C. Interactive Analysis Environment

Our second core contribution is SWAN's visualization and interactive annotation environment. It ultimately enables the developer to quickly identify system-dependent spots that require annotation and thus to cope with large implementations.

1) Visualization: SWAN's visualization transforms the program from the assembly-level analysis problem to a graph level that is more accessible. Therefore, we operate on the intermediate representation of the implicit path-enumeration technique (IPET) [31], [32] that is used internally by PLATIN. Thereof, SWAN generates a web-based interface to visualize the control-flow graph and enrich it with further information such as estimated WCETs and source-code locations. An example of the resulting view is shown in Figure 5. Technically, the visualizer intercepts both the transition costs that PLATIN determines for the individual transitions as well the execution frequencies obtained from the ILP. Blocks in the graph are labeled by their internal name (e.g., `test.c:2/0`) to facilitate a simple mapping of nodes in the control-flow graph and the source code. We further refine the graph by grouping blocks by their respective function and enriching them by semantic information and dedicated marking nodes (e.g., `loopheader`). The visualization grants further facilities, such as hovering blocks that reveal the corresponding code snippet based on the debug information, which supersedes switching between source locations and analyzer. Indeed, the usability of our visualization for source-level annotations heavily depends on the accuracy of the debug information and the accessibility of the overall graph structure. Fortunately, we can harness our optimization-aware compilation at this point: by initially compiling and visualizing the IPET-graph without optimizations (i.e., `O0`), we can obtain very accurate hints and good graph structure. As the optimization-aware compilation maintains soundness of the annotation, the code can be recompiled with the optimization to obtain timing information in the end.

2) Interactive Annotation: Finally, we developed an interactive variant of SWAN and the tool `platin interact` that allow for the shortcut shown in Figure 4. In this interactive mode, SWAN operates directly on the intermediate representation and allows for on the fly annotation, which eliminates the costly recompilation and reparsing steps. A read-evaluate-print-loop (REPL) command-line tool aids the developer in the annotation process and provides commands to add new annotations, modify existing annotations, change the currently assumed values for system facts, perform WCET analysis, and finally visualize the changes as part of the interactive session. For the two problematic spots presented in Figure 5, the annotation could, for instance, be performed with the following commands (assuming a suitable system fact or expression `J`):

```
annotate function/while.cond lbound "J"
annotate function/if.then guard "J␣<␣0"
```

In this example, the advantages of graph-like representations over mere source locations of analysis errors become apparent: It is ineffective to directly annotate the unbounded loop within the `error_handling` routine. Instead, we must exclude the entire path from invoking the error handling, iff we can prove that the error cannot occur given the current set of system facts. In the visualization, suitable annotation spots are easy to find by following the graph upwards and inspecting the relevant code snippets by hovering the mouse in the web-based interface. After annotation, the function can immediately be reanalyzed, updating the execution frequencies and thus providing direct visual feedback on the effects. On successful completion of

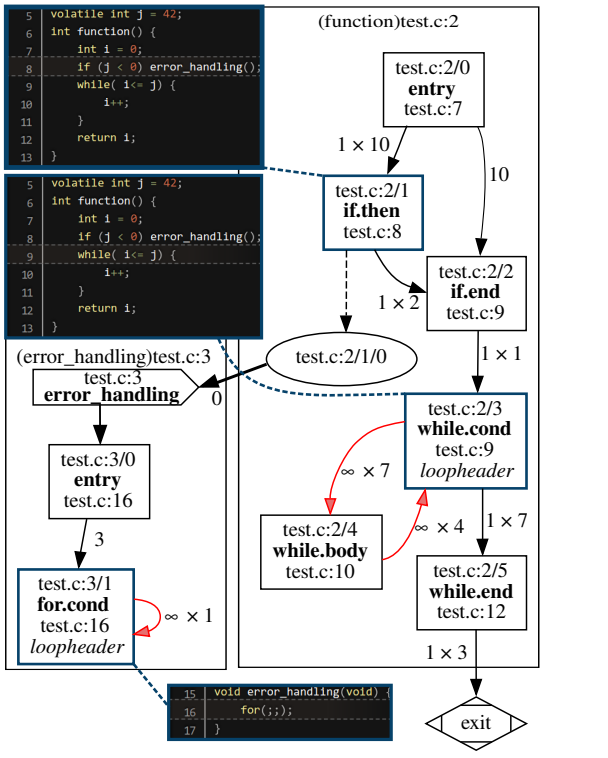

Figure 5. Example of the interactive annotation viewer (magnification of the visualization step shown in Figure 4)

the session, the tool finally assists the developer in integrating the changes into the code base in a semi-automatic fashion. Therefore, SWAN exercises all relevant source locations, which are obtained from debug information, and offers to insert and potentially adjust the individual annotations. Currently, this process is implemented for the VIM editor.

VI. EVALUATION

In this section, we present the experimental evaluation of SWAN. After detailing our setup, we demonstrate the effectiveness of our system-aware annotations by a case study on OS operations from FreeRTOS. Subsequently, we showcase the reduction in analysis pessimism on the example of a UAV flight-control system. Finally, we evaluate the overall performance and scalability of SWAN's infrastructure by the example of the Linux kernel.

A. Experimental Setup

As we intended SWAN to prove real-time capabilities of operating systems on a case-by-case basis, we are faced with the problem that quantitative experiments are inherently application-specific: crafting benchmarks with expensive paths that are, however, infeasible in a particular system context would allow for arbitrarily high figures for our solution. Instead, we based our reasoning on various qualitative experiments, which we adapted from established open source projects, benchmarks, and real-world systems. These experiments answer the following four fundamental questions: (1) Is our system-aware annotation approach capable of concisely expressing system-dependence in execution times by analyzing an example from the FreeRTOS [33] operating system? (2) Whether this dependence is commonly found in systems by examining benchmark programs from the publicly available TACLeBench suite [34]? (3) How our system-awareness quantifies for end-to-end response times in real-world applications with and without application code using a UAV flight-control system [9], [35]? (4) Is SWAN's interactive analysis, caching layer, and compilation toolchain effective even for large, complex code bases such as the Linux kernel and does it scale?

As discussed in Section III-A, our technique integrates into PLATIN's program-flow analysis and thus is decoupled from hardware analysis. Still, complex hardware features may induce additional overestimations in the final result. Therefore, we opted for a simple processor model to obtain an unbiased analysis of our technique, as it is common when benchmarking flow analyses [14]. We assume a simple single-core processor model (ARM Cortex-M4) without inter-instruction effects beyond pipelining and without caches.

B. Case Study: OS Operations

The following experiments evaluate our assumption that kernel operations can be annotated and analyzed with SWAN in a system-aware manner and that our approach reduces pessimism. We start with the scheduler as it is not only a core component of an RTOS but also a prime example of an application-defined, system-state–dependent operation that is used frequently by all tasks. The scheduler uses several data structures for bookkeeping and storing runtime information, for example, which tasks are runnable or blocked on what resource. These data structures lead to a strong interdependency between the application context and the WCET of scheduling operations. Figure 6 compares the static and the system-aware upper bound for the `vTaskSuspend` system call of FreeRTOS. This function suspends the current task and selects the next runnable task upon reschedule. It uses one queue per priority level and probes those in decreasing order to pick the highest-priority (runnable) task. In the following experiment, we configured FreeRTOS for a total of 42 priority levels, which translates to a static upper bound of 6498 cycles for the worst case, which assumes no runnable task in any of the queues and therefore requires 42 probing steps. With PLATINA's parametric annotations and by using system facts, we can accurately express this state dependency by combining the number of priority levels and the priority of the next task into a complex `lbound`-annotation (`MAX_PRIO - NEXT_PRIO`). With this system knowledge, the system call's WCET correctly reflects the system-state–dependent number of probing cycles, with a potential reduction in analysis pessimism between the best system-aware and the static bound of more than 178 %. We discuss the acquisition of system-state–dependent values by whole-system analysis in Section VI-C.

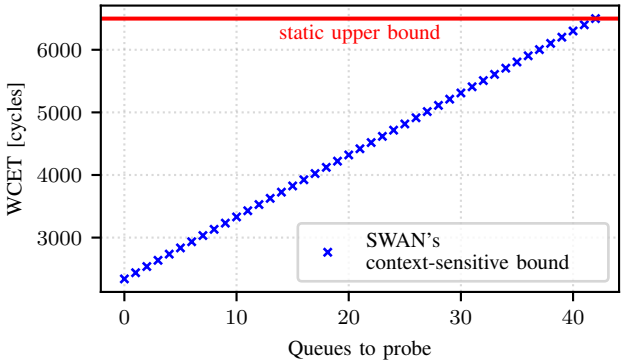

Figure 6. Constant vs. parametric annotations: WCET estimates for the `vTaskSuspend` system call in relation to the maximum number of task queues to be probed in a specific system context.

A further OS-related issue arises from the loss of analysis context (e.g., system-call parameters) by crossing the application-kernel boundary, which induces unnecessary pessimism to the system call's analysis. A typical pattern among kernel operation is the processing of memory objects of unknown sizes, such as copying data from user to kernel space or inter-process communication, which can benefit from our parametric annotations. Again, the potential improvements are application-specific, and, to the best of our knowledge, there are no generally available system-call–centric benchmarks for timing analysis that we could directly use to research the effects and set our approach into a known context. Instead, we surveyed TACLeBench [34], an application benchmark suite commonly used by the community and examined functions that correspond to said variable size idiom and thus can act as a realistic substitute. We found them all to process their work-

Benchmark	Function	Loop Iterations		Improvement
		Min	*Max*	
ammunition	`memset`	0	4	100 %
	`memmove`	2	4	50 %
	`memcpy`	2	6	67 %
md5	`memset`	128	208	38 %
	`memset_x`	16	64	75 %
	`memcpy`	0	55	100 %
pm	`memcpy`	44	256	83 %

Table I
Potential for parametric analysis in TACLeBench for the `fun(*a, *b, length)` idiom determined by loop iteration ranges

load iteratively, with the size parameter corresponding to the loop bound of the processing part. Consequently, we used the known costs of the latter to derive well-founded expectation values on the potential improvements that are achievable with SWAN. For a set of representative TACLeBench benchmarks, Table I gives the minimum and the maximum number of processing iterations (for all valid inputs) as well as the resulting improvement in analysis precision. The latter ranges from 38 % to 100 % (i.e., an overestimation of factor two).

In summary, our quantitative figures qualitatively substantiate that our approach to system-aware analysis is feasible and able to deliver significantly tighter bounds.

C. Case Study: Flight-Control System

To this point, we focused on SWAN's fundamentals and the two lower layers of Figure 1, respectively. To obtain a better notion of SWAN's applicability on real-world scenarios and to showcase the impact of high-level systems facts, we further conducted a case study on a flight-control system [35] to explore the potential of system-aware analysis. The benchmark includes signal processing, steering, and flight control and analyzes the temporal properties of a task group consisting of five tasks and an interrupt handler. Putting this setup into relation with other real-world real-time benchmarks, such as DEBIE [36], our used benchmark has a similar complexity and also consists of many system calls. Beyond its practical relevance, this benchmark is especially suitable for our evaluation, as it is already available [9] for an OSEK-based system, including an STG for fixed-priority scheduling. We ported the benchmark to FreeRTOS v10.0.1, which required only minor changes, generated the necessary system facts from the STG and pinned them the right spots in the kernel control flows using annotations. Thereof, we conducted comparative experiments: (1) *constant annotations* that always assume the worst system state, and (2) *system-aware annotations* that leverage system facts. For both settings, we evaluated two variants: OS overheads (a) without the application code in isolation and (b) with the application code, which we generated from the application's Simulink model. For the evaluated task-chain, the benchmark performs 14 system calls (in as many aggregated system contexts), which dispatch to six different scheduling-related system calls. Their implementation consists of 18 unique functions and 1041 assembly instructions across 192 basic blocks. The application introduces 18 additional functions containing 1875 assembly instructions.

Table II summarizes the results by the estimated individual WCETs and the overall WCRT. For the sake of simplicity, we derived the latter by pessimistic aggregation. The reduction in pessimism ranged between 0 % up to 60 % in our experiments. This variation stems from the types and usage patterns of system calls within the individual tasks. For example, the `SignalGatherFinishedTask` resumes both `SignalProcessing` tasks. Therefore, a static analysis must assume two reschedule operations accompanied by the probing of all priority-levels in the respective queues, which is the static baseline. In practice, however, just like in our running example from Figure 1, FreeRTOS compares the current tasks priority of 25 with the ones of the resumed tasks (22 & 21) and thus skips rescheduling altogether. By adding system-aware `guard` annotations and leveraging the respective system facts, SWAN's analysis is aware of this semantics and can identify the control-flow paths to be infeasible in this system context, which reduces pessimism by over 60 %. On the other hand, the lowest priority `SignalProcessingAttitudeTask` only processes the input and suspends itself. As it is always the last to run in the processing chain, there is no other runnable task in the queue. Consequently, no system fact could be exploited, which is why the improvement is zero. In total, we integrated five system-aware annotations into the scheduler, while the application required eleven constant, traditional flow facts as well as one application-specific system-aware annotation. Notably, the system-aware WCRT is lower than the sum of the task's WCETs. The reason for this is that `SignalGatherTimeoutTask` represents an optional member of the task group, which is only conditionally resumed. However, the respective path does not constitute the task's overall worst-case path. Again, using our system-aware analysis approach, we were able to analyze both settings and therefore obtain an even tighter yet still sound WCRT. Overall, we observed a substantial improvement of 40.7 % on OS-overhead estimates. Including the application WCETs, this translates to a reduction of 25.5 % on the overall WCRT. Furthermore, our experiments demonstrate that our concept of system facts is generalizable to incorporate semantic knowledge, for example from application models or OS–state-analysis approaches [9].

D. Development Support and Scalability Issues

Our final evaluation is devoted to our tooling infrastructure and the question of whether we can, with a focus on the interactive analysis, cope with larger code bases. Therefore, we based this set of experiments on the Linux kernel, version 4.1.39 using the llvm-linux patchset [37], and measured the processing times for the steps required for timing analysis of system calls: (1) Compilation of the kernel with our tailored `patmos-clang` compiler, which extracts the system-aware annotation information and builds the CFRGs. (2) Parsing of these outputs by PLATIN and constructing the analysis's intermediate representation. (3) Obtaining the symbol addresses by correlation with the actual binary. (4) Transformation and lowering of both system and static flow facts to the assembly level and finally (5) performing the actual WCET analysis of

Task	Operating System Only			Including Application Code		
	WCET (cycles)		*Improvement*	*WCET (cycles)*		*Improvement*
	Static	*System-Aware*		*Static*	*System-Aware*	
`SignalGatherInitiateTask`	20 342	13 214	35.041 %	29 336	22 208	24.298 %
`SignalGatherTimeoutTask`	20 220	13 092	35.252 %	24 717	17 589	28.838 %
`SignalGatherFinishedTask`	18 892	7545	60.062 %	30 464	19 117	37.247 %
`SignalProcessingActuateTask`	6694	4615	31.058 %	15 665	13 586	13.272 %
`SignalProcessingAttitudeTask`	6694	6694	0.000 %	15 665	15 665	0.000 %
WCRT	72 842	43 199	40.695 %	115 847	86 204	25.588 %

Table II
Static and system-aware WCET analysis results for the individual tasks of the copter benchmark, with and without the simulink generated application code

the `getcpu` system call. Table III gives the execution times for these steps based on a multi-socket 40-core Intel® Xeon® E5-4640 with 128 GB RAM.

In total, the benchmark consists of 19 773 functions, out of which we exemplary analyzed `Sys_getcpu`. The entire kernel totals 2 712 998 instructions, with the control-flow-relation graph mapping 227 920 bitcode-level basic blocks to 226 761 on the machine-code level. The kernel compilation takes up most of the time if not parallelized or stripped-down to an incremental rebuild. However, even then, the complete annotate-compile–analyze cycle takes over 16 min, which jeopardizes any iterative development process. The remaining overhead can be attributed to PML parsing and symbol correlation, which has to be repeated for every contextual change (i.e., individual system call). With our infrastructure and `platin interact` we were able to eliminate both steps. By smart caching and reuse of analysis artifacts, we can bypass recurrent compilation, parsing and symbol correlation, cutting turnover time by up to 99 % to fractions of a second in our experiments and the exemplary system. The only steps that have to be re-executed are the flow-facts transformation and the actual WCET analysis (highlighted in Table III), both of which directly evaluate new or changed annotations. These substantial improvements with SWAN enable convenient annotation and analysis even for large code bases, such as the Linux kernel.

VII. Discussion & Future Work

In the following section, we first discuss alternative approaches and future work in both phases of WCET analysis, that is path analysis (see Section VII-A) and hardware analysis (see Section VII-B).

A. Path Analysis

As the case study based on a UAV in Section VI-C shows, SWAN significantly reduces pessimism by exploiting system knowledge. For now, during the benchmark's analysis, most of the used system facts relate to the applications task set, its respective priorities and their interplay with the fixed-priority scheduling employed by the underlying OS. In the study, we obtained these system facts from a given STG of the real-time system. To obtain this representation, the SysWCET tool [9] implements a simulator of OSEK's fixed-priority scheduling strategy and a stack-based priority ceiling protocol [38] and beginning from the application's initial state, explores all reachable system states via explicit path enumeration. For example, with this approach, we can create the STG comprising around 40 000 system states within few seconds [20]. For comparison, the real-world flight-control system (see section VI-C) has around 10 000 states. A further factor of analysis time is ILP-solving time. Here the instantiation of ABBs for the different contexts introduces additional variables. To mitigate the problem of long analysis times, several heuristics and configuration options for solvers exist [9], [20], [39]. In this context of integrating system-wide semantics, the aspect of combining local and global data-flow analysis will become more relevant. Thread-local data-flow analyses are a well-explored topic in WCET analyses [40]. The knowledge on system-wide program flows now enables global control-flow graphs where combinations of global (e.g., value constraints of a global variable holding the task's priority) and local data-flow analyses are possible, which will enable future refinements of analysis pessimism.

Step	Runtime (hh:mm:ss.ms)		
	Max	∅	*Min*
Compiling, sequential	2:41:40.	2:40:39.	2:40:26.
Compiling, parallel	4:23.	4:17.	4:09.
Compiling, incremental	56.	55.	55.
Parsing PML Files	11:57.444	11:00.374	10:04.970
Read Symbol addresses	06:04.118	05:41.617	05:16.037
Flow Facts Transformation	0.154	0.127	0.104
Static WCET analysis	0.004	0.003	0.003

Table III
Timings of the steps in the annotation cycle, averaged over ten executions

B. Hardware Analysis

Although SWAN's primary focus is on path analysis with the PLATINA annotation language and the interactive mode to enable analysis of large code bases, the identified path-analysis results are vice versa utilizable to reduce pessimism in the hardware analysis: A challenging aspect of hardware-cost analysis is precisely accounting for delays due to reloading cache blocks after a preemption's code evicted blocks, which is known as cache-related preemption delays (CRPD) [16], [41]. Here, SWAN's path analysis and knowledge on paths are helpful, for example, in a fixed-priority real-time system using a stack-based priority-ceiling protocol [42]. With this resource protocol, tasks can have a higher (dynamic) priority during the path of holding a resource. Consequently, for the CRPD analysis, several preempting paths are not possible. SWAN is capable of expressing such system knowledge on potential preemptions and by limiting the number of possible preemptions and thus

helps to reduce the hardware-analysis pessimism, which is a promising aspect of future work. Generally, the exclusion of path interferences with application-specific knowledge is also helpful in research on multicore WCET analysis, which goes beyond our current system model of single-core processors. Nonetheless, SWAN is already applicable in scenarios where both the scheduling and the hardware resources are partitioned and a single-core equivalence is guaranteed [43].

VIII. Related Work

To the best of our knowledge, SWAN is the first approach to tackle the problem of proving the real-time capability of a generic OS through a system- and compiler-aware approach paired with the possibility of interactive WCET analyses.

Past attempts on static WCET analysis of operating systems share a large number of observations with our initial problem assessment: Colin and Puaut [44] were the first to pinpoint fundamental problems of OS WCET analysis, such as meaningful construction of global control-flow graphs over kernel boundaries. They identified indirect function calls, immanent in the syscall interface, as well as dependencies between application properties and kernel loops as crucial issues. Their approach was to modify the source code, restoring a statically analyzable implementation. Despite these tedious and non-generalizable measures, significant overestimation (avg. 86 %) remained. Later Sandell et al. [45] reported a large number of "uninteresting" kernel paths (e.g., error handling) in their analysis. Furthermore, they observed a mediocre performance of automatic data-flow analysis within the OS, which emphasizes the need for manual annotation. They used an annotation language with constant expressions at the assembly level to address the problems, which, however, involved a high degree of recurring effort with still unsatisfactory reduction of pessimism. In contrast to their approach, SWAN allows stating source-code annotations, which are propagated through an optimization-aware compiler infrastructure.

Among others, Schneider [8], [46] determined the dynamic reconfigurability and system-call interface, the tracing of call graphs, as well as the internal feedback between the RTOSs and the applications as further challenges towards realistic WCET estimations. He proposed an integrated WCET and scheduling analysis as a potential solution. In 2009, Lv et al. compiled a survey [3] on RTOS-analysis attempts, from which they derived a set of challenges. Like Schneider, they question the usefulness of single, global WCET estimates for individual operations but instead suggest a parametric analysis that captures specific WCETs for each invocation.

Since then, research focused on circumventing the problem by tailoring the OS to be static and more deterministic again. Undecidable artifacts are eliminated by application-specific source-code modifications: For example, in the seL4 kernel [2], preemption points in loops are added to limit the length of consecutive kernel execution. With SWAN, we chose the approach to avoid tailoring the OS, but provide support to state system-aware and OS-tailored annotations, which are propagated through the WCET-aware compilation framework.

Regarding annotations, there is a large body of related approaches, both on annotation languages [47]–[51] and tooling, that support these (context-sensitive) annotations [1], [11], [15], [22], [23], [52]. However, these approaches differ from SWAN in principle: SWAN focuses on integrating system facts, which are extracted from high-level analyses, into the timing analysis using parametric annotations, while those approaches deliver traditional (call-)context–sensitive annotations. A further major difference is the interactive annotation mode that also enables sound correlation between source and machine code. We identified when working with Real-Time Linux that large round-trip times of iterative analysis, annotation, and compilation require an effective tooling infrastructure to enable WCET analysis for increasingly complex systems.

Parametric WCET analysis itself (i.e., the extraction of formulas from existing source code) is also a well-explored research topic [53]–[56]. However, these approaches fail when analyzing entire systems with parametric inter-procedural and inter-thread dependencies. Building on this idea of parametric analyses, SWAN leverages this concept to enable expressing parametric and system-aware system facts across all layers in the real-time system (i.e., application, syscall interface, OS).

The necessity to include the OS semantic in static code analyzers has been gaining relevance in recent years. An indicator for this trend is the integration of OS interfaces into the commercial analyzer Astrée in order to prove the absence of runtime errors in concurrent, safety-critical software [10]. With SWAN, we provide a framework to prove the capability of a whole real-time system for its timeliness.

IX. Conclusion

In this paper, we presented SWAN, our approach to system-aware WCET analysis of RTOSs. The core concept is the idea of transporting system facts, that is, additional knowledge on the system configuration and system state available from the calling context, into the analysis of the OS. This information is used to eliminate unnecessary analysis pessimism, and thereby delivering tight and tailored bounds for specific operations. Its key element is PLATINA, a parametric annotation language to express dependencies between a system's control-flow and application states generically. Combined with system facts and an optimization-aware compiler and timing analysis, we can both prove real-time capabilities in a given setting and eliminate overly pessimistic constant bounds on RTOS overheads. In our case study using SWAN on a flight-control system, we were thus able to reduce analysis pessimism by 40 %.

Source code of SWAN*:* `gitlab.cs.fau.de/SWAN/`

Acknowledgment

We want to thank our anonymous shepherd for the work and helpful suggestions. This work is supported by the German Research Foundation (DFG) under grants no. SCHR 603/14-2, SCHR 603/13-1, SCHR 603/9-2, the CRC/TRR 89 Project C1, and the Bavarian Ministry of State for Economics under grant no. 0704/883 25.

REFERENCES

[1] AbsInt., "aiT worst-case execution time analyzers," absint.com/ait.

[2] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proc. of RTSS '11*, 2011, pp. 339–348.

[3] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, "A survey of WCET analysis of real-time operating systems," in *Proc. of ICESS '09*, 2009, pp. 65–72.

[4] OSEK/VDX Group, "Operating system specification 2.2.3," OSEK/VDX Group, Tech. Rep., 2005.

[5] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proc. of RTSS '06*, 2006, pp. 111–126.

[6] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[7] AUTOSAR, "Explanation of adaptive platform design," pp. 1–51, 2017.

[8] J. Schneider, "Why you can't analyze RTOSs without considering applications and vice versa," in *Proc. of WCET '02*, 2002, pp. 79–84.

[9] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, "SysWCET: Whole-system response-time analysis for fixed-priority real-time systems," in *Proc. of RTAS '17*, 2017, pp. 37–48.

[10] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand, "Taking static analysis to the next level: Proving the absence of run-time errors and data races with Astrée," in *Proc. of ERTS '16*, 2016, pp. 570–579.

[11] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener, "Embedded program annotations for WCET analysis," in *Proc. of WCET '18*, 2018.

[12] B. Huber, D. Prokesch, and P. Puschner, "Combined WCET analysis of bitcode and machine code using control-flow relation graphs," in *Proc. of LCTES '13*, 2013, pp. 163–172.

[13] T. Klaus, F. Franzmann, T. Engelhard, F. Scheler, and W. Schröder-Preikschat, "Usable RTOS-APIs?" in *Proc. of OSPERT '14*, 2014.

[14] C. Rochange, "WCET tool challenge 2014," talk held at WCET '14.

[15] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.

[16] S. Altmeyer, R. I. Davis, and C. Maiza, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, 2012.

[17] C. Dietrich, M. Hoffmann, and D. Lohmann, "Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis," *Trans. on Embedded Computing Systems*, vol. 16, pp. 35:1–35:25, 2017.

[18] F. Scheler and W. Schröder-Preikschat, "The real-time systems compiler: Migrating event-triggered systems to time-triggered systems," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1491–1515, 2011.

[19] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, "Towards system-wide timing analysis of real-time–capable operating systems," in *Proc. of ECRTS '18 WiP*, 2018.

[20] P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich, and W. Schröder-Preikschat, "Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems," in *Proc. of ECRTS '18*, 2018, pp. 24:1–24:25.

[21] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. of the AMS*, pp. 358–366, 1953.

[22] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard, "The T-CREST approach of compiler and WCET-analysis integration," in *Proc. of SEUS '13*, 2013, pp. 33–40.

[23] S. Hepp, B. Huber, D. Prokesch, and P. Puschner, "The platin tool kit – the T-CREST approach for compiler and WCET integration," in *Proc. of KPS '15*, 2015, pp. 277–292.

[24] M. Schoeberl et al., "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, pp. 449–471, 2015.

[25] ISO, *ISO/IEC 9899:2011 Information Technology — Programming Languages — C*. Int'l Organization for Standardization, 2011.

[26] D. Turner, "Total Functional Programming." *Journal of Universal Computer Science*, vol. 10, no. 7, pp. 751–768, 2004.

[27] A. Telford and D. Turner, "Ensuring Termination in ESFP," *JUCS - Journal of Universal Computer Science*, no. 4, 2000.

[28] G. Gonzalez, "dhall: A configuration language guaranteed to terminate," http://hackage.haskell.org/package/dhall-1.14.0, 2018.

[29] S. e. Marlow *et al.*, "Haskell 2010 language report," Tech. Rep., 2010, https://www.haskell.org/onlinereport/haskell2010/.

[30] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of CGO '04*, 2004, pp. 75–86.

[31] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *ACM SIGPLAN Notices*, vol. 30, 1995, pp. 88–98.

[32] P. Puschner and A. Schedl, "Computing maximum task execution times: A graph-based approach," *Real-Time Systems*, vol. 13, pp. 67–91, 1997.

[33] Real Time Engineers Ltd., *The FreeRTOS Reference Manual: API Functions and Configuration Options v9.0.0*, 2016.

[34] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *Proc. of WCET '16*, 2016, pp. 2:1–2:10.

[35] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat, "I4Copter: An adaptable and modular quadrotor platform," in *Proc. of SAC '11*, 2011, pp. 380–396.

[36] N. Holsti, T. Langbacka, and S. Saarinen, "Using a worst-case execution time tool for real-time verification of the DEBIE software," in *Proc. of DASIA '00*, 2000.

[37] Linux Foundation, "LLVMLinux kernel," http://git.linuxfoundation.org/llvmlinux/kernel.git/, 2015.

[38] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[39] Gurobi Optimization, Inc., "Gurobi optimizer reference manual," www.gurobi.com/documentation/8.1/refman.pdf, 2018.

[40] J. Blieberger, "Data-flow frameworks for worst-case execution time analysis," *Real-Time Systems*, vol. 22, no. 3, pp. 183–227, 2002.

[41] C.-G. Lee, J. Hahn, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," in *Proc. of RTSS '96*, 1996, pp. 264–274.

[42] T. P. Baker, "A stack-based resource allocation policy for realtime processes," in *Proc. of RTSS '90*, 1990, pp. 191–200.

[43] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "WCET(m) estimation in multi-core systems using single core equivalence," in *Proc. of ECRTS '15*, 2015, pp. 174–183.

[44] A. Colin and I. Puaut, "Worst-case execution time analysis of the RTEMS real-time operating system," in *Proc. of ECRTS '01*, 2001, pp. 191–198.

[45] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper, "Static timing analysis of real-time operating system code," in *Leveraging Applications of Formal Methods*, 2004, pp. 146–160.

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

[47] R. Kirner, A. Kadlec, P. Puschner, A. Prantl, M. Schordan, and J. Knoop, "Towards a common WCET annotation languge: Essential ingredients," in *Proc. of WCET '08*, 2008, pp. 53–65.

[48] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel, "WCET analysis: The annotation language challenge," in *Proc. of WCET '07*, 2007.

[49] J. Knoop, A. Kadlec, R. Kirner, A. Prantl, M. Schordan, and I. Wenzel, "WCET annotation languages reconsidered: The annotation language challenge," in *Proc. of the 25. Work. der GI-Fachgruppe Programmiersprachen und Rechenkonzepte*, 2008, pp. 93–103.

[50] N. Holsti et al., "WCET Tool Challenge 2008: Report," in *Proc. of WCET '08*, 2008, pp. 149–171.

[51] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec, "Beyond loop bounds: comparing annotation languages for worst-case execution time analysis," *Software & Systems Modeling*, vol. 10, no. 3, pp. 411–437, 2011.

[52] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: an open toolbox for adaptive WCET analysis," in *Proc. of IFIP '10*, 2010, pp. 35–46.

[53] S. Altmeyer, C. Hümbert, B. Lisper, and R. Wilhelm, "Parametric timing analysis for complex architectures," in *Proc. of RTCSA '08*, 2008, pp. 367–376.

[54] E. Vivancos, C. Healy, F. Mueller, and D. Whalley, "Parametric Timing Analysis," in *Proc. of LCTES '01*, 2001, pp. 88–93.

[55] B. Lisper, "Fully automatic, parametric worst-case execution time analysis," in *Proc. of WCET '03*, 2003, pp. 99–102.

[56] B. Huber, D. Prokesch, and P. Puschner, "A formal framework for precise parametric wcet formulas," in *Proc. of WCET '12*, 2012, pp. 91–102.