Cross-Kernel Control-Flow–Graph Analysis for Event-Driven Real-Time Systems *

Christian Dietrich, Martin Hoffmann, Daniel Lohmann

Friedrich–Alexander–Universität (FAU) Erlangen–Nürnberg, Germany

{dietrich,hoffmann,lohmann}@cs.fau.de

Abstract

Embedded real-time control systems generally have a dedicated purpose and fixed set of functionalities. This manifests in a large amount of implicit and explicit static knowledge, available already at compile time. Modern compilers can extract and exploit this information to perform extensive whole-program analyses and inter-procedural optimizations. However, these analyses typically end at the application–kernel boundary, thus control-flow transitions between different threads are not covered, yet. This restriction stems from the pessimistic assumption of a probabilistic scheduling policy of the underlying operating system, impeding detailed predictions of the overall system behavior. Real-time operating systems, however, do provide deterministic and exactly specified scheduling decisions, as embedded control systems rely on a timely and precise behavior.

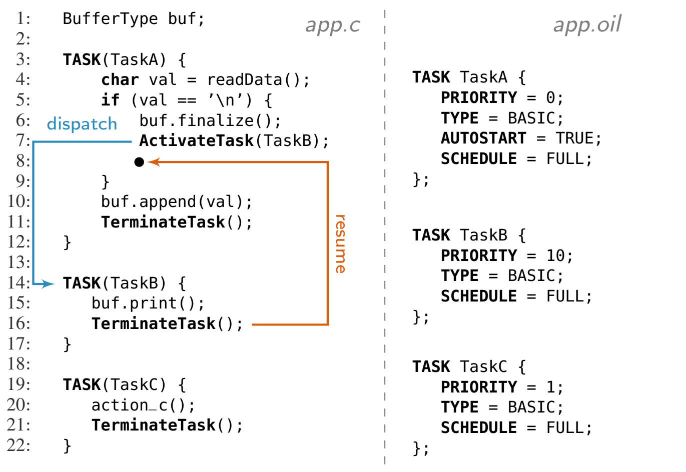
In this paper, we present an approach that incorporates the RTOS semantics into the control-flow analysis, to cross the application–kernel boundary. By combining operating system semantics, the static system configuration and the application logic, we determine a cross-kernel control-flow–graph, that provides a global view on all possible execution paths of a real-time system. Having this knowledge at hand, enables us to tailor the operating system kernel more closely to the particular application scenario. On the example of a real-world safety-critical control system, we present two possible use cases: Run-time optimizations, by means of specialized system calls for each call site, allow to speed up the kernel execution path by 33 percent in our benchmark scenario. An automated generation of OS state assertions on the expected system behavior, targeting transient hardware fault tolerance, leverages significant robustness improvements.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimizations; D.4.7 [*Operating Systems*]: Organization and Design—Real-time Systems and Embedded Systems

Keywords global control-flow graph, static analysis, OSEK, AUTOSAR, static system tailoring, whole-system optimization

* This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. LO 1719/1-3

LCTES '15, June 18–19, 2015, Portland, Oregon, USA.
Copyright © 2015 ACM 978-1-4503-3257-6//...$15.00.
http://dx.doi.org/10.1145/2670529.2754963



```
BufferType buf;

TASK(TaskA) {
    char val = readData();
    if (val == '\n') {
        buf.finalize();
        ActivateTask(TaskB);
        •
    }
    buf.append(val);
    TerminateTask();
}

TASK(TaskB) {
   buf.print();
   TerminateTask();
}

TASK(TaskC) {
   action_c();
   TerminateTask();
}
```

```
TASK TaskA {
   PRIORITY = 0;
   TYPE = BASIC;
   AUTOSTART = TRUE;
   SCHEDULE = FULL;
};

TASK TaskB {
   PRIORITY = 10;
   TYPE = BASIC;
   SCHEDULE = FULL;
};

TASK TaskC {
   PRIORITY = 1;
   TYPE = BASIC;
   SCHEDULE = FULL;
};
```

Figure 1: A small OSEK system with three tasks. `TaskA` receives data and fills a buffer, which is processed by `TaskB`. `TaskC` is never activated. The right hand side shows the according, static system configuration.

1. Introduction

Embedded real-time control systems are special-purpose systems: The built-in computers are dedicated to specific, predefined tasks [14, 7]. Hence, it is possible (and common practice) to tailor both, the hardware and system software of an embedded system to its specific needs in order to keep per-unit hardware costs as low as possible [6].

In the software, the "special-purposeness" of an embedded control system manifests in the large amount of implicit/explicit static knowledge we have available already at compile time: The structure of the application code is typically static by nature. With respect to predictability, standards for embedded software development, such as MISRA-C [9], favor `static` data over stack-based or heap-based memory allocation, prohibit the use of function pointers, and suggest to use constants whenever possible.

Thanks to *whole-program analyses* (WPAs), modern compilers can extract and exploit this static structure of the application to perform a great deal of inter-procedural optimizations, such as constant folding, caller-site inlining, or elimination of dead code and data. Such optimizations become particularly effective if they are applied per thread [22], that is, if the compiler is made aware of the OS-managed control flows of the application and their respective entry points [8]. This is generally possible in embedded real-time systems, as the set of control-flows is also static: The *real-time operating system* (RTOS) itself is tailored towards the specific application; all HW/SW events are prioritized and mapped to a finite set of threads, *interrupt-service routines* (ISRs), semaphores or other system objects. This makes it possible to allocate all system objects at compile-time in static arrays and address them by their constant index value. The automotive OSEK/AUTOSAR [17, 1] RTOS standards, for instance, suggest this technique to keep the RAM overhead as low as possible. Figure 1 exemplifies such an OSEK-based system consisting of three tasks.

1.1 Problem Statement

Nevertheless, even a control-flow–sensitive WPA cannot provide a complete picture, as it does not cover control-flow transitions between *different* threads. These transitions are executed and managed by the kernel scheduler on behalf of a syscall (e.g., posting a semaphore) and, thus, outside of the semantics of the programming language. The common assumption here is that the kernel might switch at any time to any other thread, so the compiler cannot derive constraints about inter-thread control-flow transitions.

However, while this pessimistic assumption is true for operating systems that employ probabilistic scheduling of a dynamically changing set of threads (e.g., Linux, Windows), the scheduling decisions of an RTOS are generally deterministic. In Figure 1, for example, the `AUTOSTART` thread `TaskA` sets thread `TaskB` ready (`ActivateTask(TaskB)`, line 7). As the static priority `10` of `TaskB` is higher than the static priority `0` of `TaskA`, we know *at compile-time* that at this point the kernel of an event-driven RTOS will *always* dispatch to `TaskB`. Furthermore, as `TaskB` is only activated from `TaskA` (and itself does not activate any other task), we also know that upon termination of `TaskB` (`TerminateTask()`, line 16) the execution will *always* continue in `TaskA`. A compiler that is aware of these facts could optimize the code to not invoke the kernel scheduler at these points. Even further, the compiler could inline the user-code of `TaskB` into `TaskA` and completely eradicate the respective system calls and the – then dead – `TaskB` object. Similarly, such compiler could detect that thread `TaskC` is also dead (the kernel will never dispatch to it), so it could also be eradicated. Eventually, the system from Figure 1 will be collapsed into a single thread, so that even the scheduler is no longer needed.

This basic example is, of course, simplistic. It does, for instance, not contain any ISR that may activate a task at any time. Nevertheless, even with ISRs it is typically possible in an event-triggered real-time system to derive *some* knowledge about inter-thread transitions – in order to exploit this knowledge for truly global cross-kernel control-flow system optimizations.

1.2 About This Paper

In this paper, we describe our approach to construct a *global control-flow graph* (GCFG) of a static event-triggered real-time system. In addition to an ordinary (control-flow–sensitive) *control-flow graph* (CFG), the GCFG also incorporates the RTOS semantics, including interrupts and synchronization primitives, to model the control flow even across multiple threads and kernel invocations.

We show how, once obtained, the GCFG can be employed to speed up kernel activations in our benchmark system by 33 percent. It can also be employed to harden the kernel against transient hardware faults. Our software-based measure reduces the silent data corruption count by 49 percent for an already hardened system. The presented optimizations are novel and only enabled by the whole-system view of the GCFG. Further measures only applicable on the system level, like checking for specification conformity are possible. The GCFG information enables the lifting of optimizations known from the function level, like dead code elimination, pointer-alias analysis, or constant propagation, to the system level.

2. System Model

To achieve sound GCFG analyses, the underlying RTOS has to provide four basic properties: First, a deterministic scheduling policy, as for example fixed-priority preemptive scheduling. Second, all system objects must be declared before run time, either in some dedicated configuration file, or unambiguously in the application code itself. Third, system-service calls must be explicit, that is, indirect invocations via function pointers are not allowed. Finally, system objects must be referenced with compile-time constant identifiers or link-time constant addresses.

In practice, these requirements are already fulfilled or easy to achieve for event-triggered real-time control systems – they are basically a technical consequence of predictability and, thus, already mandated by the relevant coding standards of the domain. A large class of systems that fulfills them out of the box are OSEK/AUTOSAR-based automotive control systems. Without loss of generality, we therefore describe our approach in the following on the example of the system model mandated by OSEK-OS [17].

In our current implementation, the GCFG is constructed for a single core (OSEK-OS specifies a single core system). However, our approach would also work for multicore systems with strictly partitioned scheduling, such as AUTOSAR 4.0 [1].

2.1 Overview of OSEK-OS

OSEK-OS [17] has been the dominant industry standard for automotive RTOS for the last two decades. Originally intended for single-core single-application systems, it has been extend for spatial and temporal isolation and multi-core support in AUTOSAR-OS [1], but the core API and concepts remained unchanged. So all of the following equally holds for AUTOSAR-OS–based systems.

OSEK specifies terminology and the API for a completely statically configured event-triggered RTOS. For a specific automotive application, all system objects and their configuration have to be declared at compile-time in a domain specific language, the *OSEK Implementation Language* (OIL) [16]. From this specification, the concrete RTOS instance is typically derived by a generator.

At run-time, the application manipulates the operating-system state by invoking *system services*, which influences the system behavior (Table 1 gives a short overview).

Control-Flow Abstractions: ISRs and Tasks

OSEK offers two main control-flow abstractions: ISRs and *tasks* (traditionally called threads). ISRs are activated by the hardware and fall into two classes: *category-1* ISRs, which are not allowed to call system services; and *category-2* ISRs, which are synchronized with the kernel. Tasks have a statically assigned priority, are allowed to use all system services, and are invoked according to a fixed-priority preemptive scheduling policy.

On each new activation, tasks start from the very beginning until their (self-) termination. Each task is configured to be either non-preemptive (enforcing run-to-completion semantics) or fully preemptive (see `SCHEDULE = FULL` in Figure 1). Preemption points can be either *synchronous*, for example caused by an explicit activation of a higher priority task (e.g., `ActivateTask(TaskB)`, line 7), or *asynchronous*, if a higher priority task is activated inside an ISR. Recurring, periodic or aperiodic, task activations can be triggered with the help of statically configured *Alarms*.

Synchronization Primitives: Global Lock and Resources

Inter-task synchronization can be realized either by a coarse-grained global interrupt lock, or more fine-grained *Resource* objects. Based on a *stack-based priority-ceiling protocol*, OSEK resources ensure mutual exclusion while preventing deadlocks and priority inversion. Through the acquisition of a resource, a task raises its *dynamic* priority to the *ceiling* priority of the resource – the highest static

System Service	Arguments	Brief Description
`ActivateTask`	`TaskID`	Task – `TaskID` – is activated. If the current task is preemptable, an immediate rescheduling takes place.
`TerminateTask`	–	The current task terminates itself. An immediate rescheduling takes place.
`ChainTask`	`TaskID`	The atomic combination of `ActivateTask(TaskID)` and `TerminateTask()`.
`GetResource`	`ResID`	Acquires the resource identified by `ResID`.
`ReleaseResource`	`ResID`	Leaves the critical region associated with the resource `ResID`. The dynamic priority of the calling task is changed and a reschedule takes place for preemptable tasks.
`DisableAllInterrupts`	–	Disables all interrupts
`EnableAllInterrupts`	–	The inverse operation to `DisableAllInterrupts`.

Table 1: (Incomplete) list of system services provided by the OSEK API. Not all control flows are allowed to invoke all system services.

priority of all tasks that can obtain the resource, according to the OIL file. The OSEK specification further defines four conformance classes (BCC1, BCC2, ECC1, ECC2), which describe minimum requirements about optional features provided by the system. In contrast to *basic* tasks (BCC1/2), *extended* tasks (ECC1/2) are allowed to use *Event* objects to block and enter a waiting state. In this work, we target the conformance class BCC1 (i.e., only basic tasks and only one task per priority) plus the aforementioned resource concept. In principle, the other conformance classes could be supported as well, however, we have not yet implemented this.

Sources of Information

With these system objects at hand, we can construct real-time systems, composed of ISRs and tasks, which are activated by external or software signals and coordinated using interrupt blocks or resources. The OIL file statically defines and configures all objects, providing *coarse-grained* application knowledge. To achieve more *fine-grained* knowledge on the overall system behavior, a detailed analysis of the kernel–task interaction is necessary.

3. Fine-Grained Interaction Knowledge

One piece of fine-grained knowledge about the application is interaction knowledge: How does the application interact with the kernel and what is the kernel's reaction. In this section we describe the *global control-flow graph* (GCFG) and present methods to extract it from the application's structure and the system configuration.

3.1 Global Control-Flow Graph

In many modern compilers, CFGs [2] are the vehicle to capture the program logic of single functions. CFGs are directed graphs with *basic blocks* (BBs) as nodes and a single *entry node*. The functions' code is partitioned into BBs, where the code in one BB can only be executed linearly. From a high-level perspective, an edge in the CFG between two BBs has an execution-order semantic; in every execution trace two BBs can only follow each other, iff there is an edge in the CFG.

We develop the GCFG semantic from the observation that the CFG expresses the BB execution order *within* a function. With a function call, control is transferred from the caller's to the callee's CFG. The *interprocedural control-flow graph* (ICFG) is formed by connecting all BBs in a program; it captures the execution order on program level and respects control transfers caused by function calls. From the operating-system perspective, the ICFG expresses the execution order on task (or thread) level. By rescheduling, the operating system switches control between two tasks, and therefore between their ICFGs. The GCFG is one level higher; it expresses the execution order on the system level. Iff there is an edge in the global control-flow graph between two basic blocks, they may be executed directly after each other on the real hardware. Nevertheless, like regular CFGs, also the GCFG can include infeasible paths.

In Figure 2, an example GCFG is shown for the application from Figure 1. `TaskA` has been assigned a low, while `TaskB` has a high

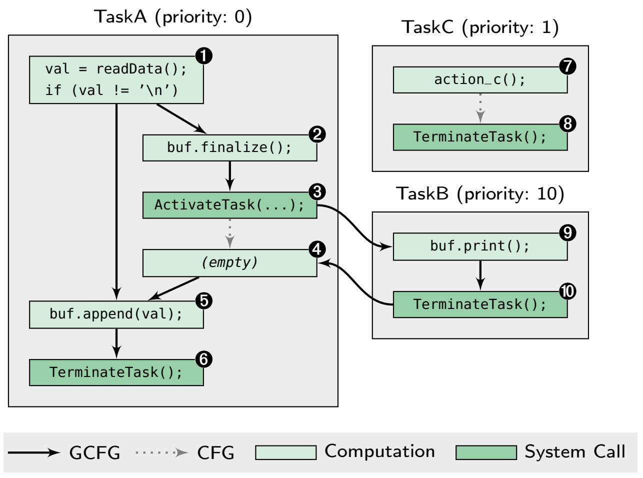


Figure 2: The GCFG representation of the system shown in Figure 1. Dotted lines are part of the local CFG, but not part of the GCFG. The "dead" `TaskC` is not part of the GCFG at all.

priority. When the application does not interact with the kernel, the GCFG corresponds to the CFG execution order (e.g., ❶ $\rightarrow$ ❷, ❶ $\rightarrow$ ❺). Any system-service invocation (❸, ❻, ❽, ❿) requires the kernel to react. In case of `ActivateTask` (❸), a task with higher priority is activated. According to the OSEK specification, the kernel scheduler chooses `TaskB` and dispatches to its entry block. After `TaskB` has terminated itself with `TerminateTask` (❿), the execution of `TaskA` is resumed. Here, we can observe, that edges, present in the CFG (❸ $\rightarrow$ ❹), are not necessarily part of the GCFG: Block ❹ cannot execute directly after block ❸ on the underlying hardware.

For the construction of the GCFG, we have to answer two questions: (1) How to partition the application code into blocks? (2) What edges have to be drawn between these blocks?

For the code partitioning, we use an adaptation of the *atomic basic block* (ABB) concept introduced by Scheler and Schröder-Preikschat [20]. An ABB is a control-flow super structure that subsumes one or more *basic blocks* (BBs) forming a *single-entry–single-exit* (SE-SE) region; it has *exactly one* distinguished entry BB and one exit BB. Besides these two blocks, no BB has a preceding or succeeding block outside of the ABB region. Every BB is member of exactly one ABB. As an adaption of the original ABB concept, we construct and connect the ABBs differently, for the whole application at once:

1. A function that contains a system call is a *system-relevant function*. Each function that calls a system-relevant function is a system-relevant function itself.

2. We iterate over all basic blocks of all functions in the application: each basic block that contains system calls and/or calls to system-relevant functions is split directly before and after those locations into subsequent parts (see ABBs ❷, ❸ and ❹ in Figure 2).
3. Depending on their content, we assign a type to each basic block: system-call block, function-call block, or computation block.
4. We collect adjacent *computation* blocks into single-entry–single-exit regions.
5. Each system-call block is an ABB, which contains a single system-call; each function-call block is an ABB, which contains a single function call; and each SE-SE region of computation blocks is an ABB.
6. Within a function, the ABBs are connected into a local control-flow graph corresponding to the connections of their entry and exit BBs.

After this construction, we have a *local* ABB-graph for each function within the application code. By the distinction of system-relevant functions, calls to system-*irrelevant* functions and subsystems are fully subsumed into computation ABBs. This subsumption not only reduces the number of blocks we have to consider, but also sharpens the focus on the application logic that is visible to the operating system. Interaction with the kernel is only possible in system-call blocks. Each system-call block has only computation blocks as direct neighbors.

In Figure 2, each green block represents an ABB. In the example, `readData()` is no system-relevant function, therefore block ❶ is not split before and after the function call, subsuming the internal logic of `readData()`. The empty ABB ❷ is the result of the split operation that was performed because of the `ActivateTask` system call.

3.2 System-State Enumeration

To construct the GCFG, we combine three sources of information: (1) The *system semantics*, as defined by the OSEK specification. (2) The static *system configuration*, which is specified in the OIL file, and (3) the *application logic*, described by the local ABB graphs.

As a combination method, we present the *system-state enumeration* (SSE). Briefly explained, the SSE computes all possible system states ahead of time and creates a *state-transition graph*. The resulting states are partitioned into groups depending on the ABB they are currently executing. A GCFG edge between two ABBs exists, iff at least one state in the state group of the source ABB has an edge to a state in the state group of the target ABB.

Abstract System State Representation

The basis of the SSE analysis is the *abstract system state* (AbSS) representation, which subsumes all relevant behavioral information of the system for single points within the control flow. Figure 3 depicts a single system state for the system from Figure 2. Each task declared in the OIL file is assigned a record with fields capturing its current task state and dynamic priority. For OSEK, each task can also hold zero or more resources, which are used to calculate the dynamic priority. The resume-point field contains the ABB to be executed next in the context of the task. Preempted tasks will continue their execution at this point. The resume point of the currently running task is the next block to be executed in the system context; the *next ABB*. The *return stacks* store return ABBs, which are pushed on function calls. *Interrupt block* indicates whether interrupts (ISR2s) are currently enabled, which is a system-wide information.

SSE Algorithm: Basic Concept

The enumeration of all system states is achieved by the repetition of a step function until a fix point is reached (no new AbSSs appear). The step function pops one state from a working stack, calculates all subsequent states, inserts edges into the AbSS graph, and pushes the follow-up states onto the working stack, if they were newly discovered. The calculation of follow-up states is based on three functions:

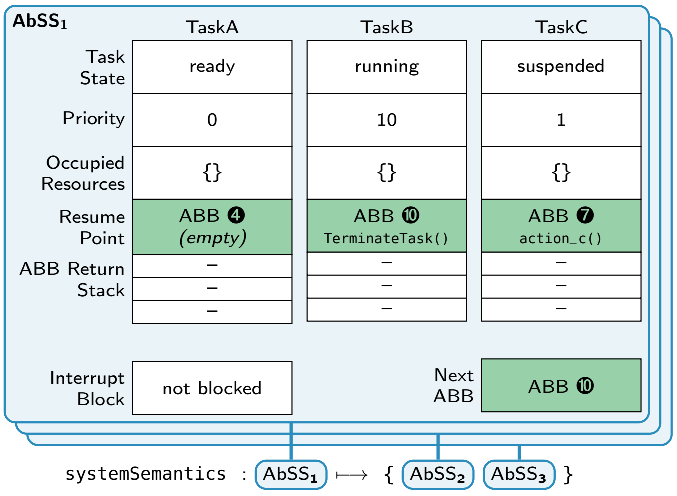


Figure 3: A detailed view on an *abstract system state* (AbSS). The `systemSemantics` function maps an input AbSS to a set of follow-up states.

$$\texttt{systemSemantics} : State \longmapsto \{State\}$$
$$\texttt{schedule} : State \longmapsto State$$
$$\texttt{execute} : State \longmapsto \{State\}$$

The `systemSemantics` function maps the input state to a set of follow-up states and is composed of the `execute` and the `schedule` function:

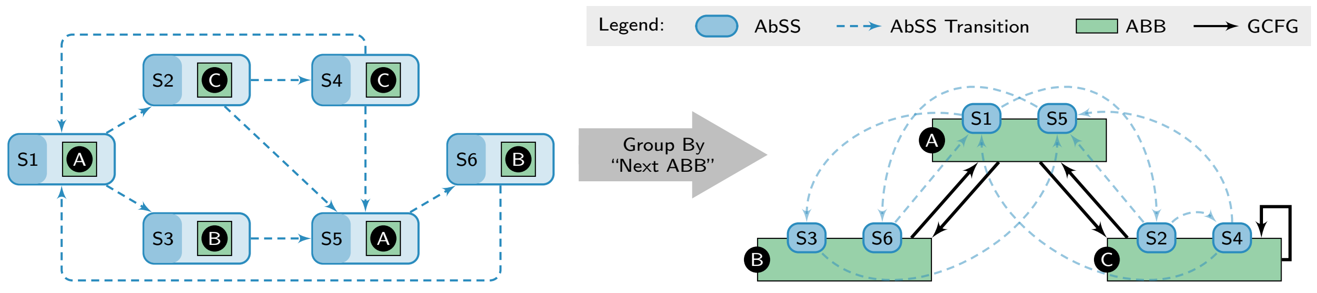
$$\texttt{systemSemantics}(x) \rightarrow \{\texttt{schedule}(y) \mid y \leftarrow \texttt{execute}(x)\}$$

The `schedule` function updates the dynamic priorities, according to the resource occupation states, and chooses the next running task, according to the scheduling rules mandated by OSEK. The `execute` function captures the influence of executing the next ABB. For each of the three block types, different rules apply:

For system-call blocks, all non-terminating system calls set the resume point of the currently running task to the computation block following in the local CFG. In Figure 3, the resume point of `TaskA` was set by the `ActivateTask` in ABB ❸. Afterwards, `execute` transforms the input state according to the system-call type and arguments. This is enabled by the constraint that all system calls must be constant in location, type, and their arguments. In the example, `execute` evaluates the `TerminateTask` system call and returns a single state: `TaskB` is *suspended* and its resume point is reset to the task's entry block (❾). The `schedule` operation marks `TaskA` as the running task and ABB ❹ will be executed next.

Function-call blocks push their single CFG-successor block onto the ABB return stack. When the execution of the function reaches a computation block without CFG-successors (exit node), an ABB is popped from the stack and used as the resume point.

Although computation blocks seem harmless, their `execute` semantic is the most complex. While all system-call and function-call blocks have a single successor, due to the ABB split operation, computation blocks may have *several* successors. For every CFG successor, `execute` emits a single follow-up state where the next ABB is set to the successor block.



(a) State-Transition Graph

(b) Derived Global Control-Flow Graph

Figure 4: The abstract system states in the state-transition graph are partitioned into state groups according to their "Next ABB" field. A GCFG edge links two ABBs, if any state Sn in the ABB's state group has at least one successor in the other's group.

Furthermore, interrupts (alarms and ISR2s) occur only in computation blocks. With function-call and system-call blocks, we capture only the uninterruptible, atomic moment of control transfer between system-relevant functions or tasks. Therefore, all asynchronous signals are handled within computation blocks.

Interrupt Handling

In order to support interrupts in the system analysis, we create a virtual task for every ISR defined in the OIL description. These tasks are configured as non-preemptable tasks, with fully disabled interrupts, and with the highest possible priority in the system. Therefore, our ISRs cannot be nested, which is one possible implementation according to the OSEK specification. For each declared alarm, we create an ISR containing a single `ActivateTask()` call.

The activation of an interrupt that is synchronized with the kernel can be treated like an asynchronous system call made by the hardware. If interrupts are enabled in the input state, `execute` emits one follow-up state for each ISR and alarm, where the virtual ISR task is set to ready. The resume point of the interrupted task is not changed; the interrupt will return exactly to the same computation block. Afterwards, `schedule` will always jump to the entry of the handler function, which is executed in a run-to-completion semantic.

With firing every interrupt source in every computation block we are on the safe side, if no information is available about timing, minimal inter-arrival times, and execution times of computation blocks or interrupt handlers. By leaving the resumption point untouched, we capture multiple activations of a single interrupt and activations of multiple interrupts. Nevertheless, this approach has the drawback of a significant state-explosion.

To ease this shortcoming, we provide the possibility to give additional coarse-grained information about the system configuration. The developer can declare groups of tasks that are used to handle a single physical event. Until not all tasks in the task group have finished their execution, the interrupt that activates this group cannot fire again. Providing this information is a qualitative statement about the execution time; the deadline of task group's execution is shorter than the minimal inter-arrival time of the activating interrupt. As future work, quantitative timing information, like the block *worst-case execution times* (WCETs) and precise interrupt timings, could be used to rule out some interrupt activations from the analysis.

Final GCFG Construction

After the stepping function has reached a fix point, we have enumerated all possible system states. Based on the resulting state graph, we can construct the GCFG by partitioning all system states into state groups depending on their *next ABB* field. In each group, all states will execute the same ABB next. We add GCFG edges between the ABBs Ⓐ and Ⓑ if any state in group Ⓐ has at least one successor in the state group of ABB Ⓑ. The GCFG edge expresses: After Ⓐ has executed, it is possible to execute Ⓑ next.

Figure 4a shows a state-transition graph for a system consisting of three ABBs. This graph is the direct result of the SSE stepping function. In Figure 4b, the state groups are drawn next to their ABBs; S3 and S6 belong to the state group of ABB Ⓑ. Since a state–state transition exists between S1 and S3, we insert a GCFG edge between ABB Ⓐ and ABB Ⓑ. ABB Ⓒ has a self-loop, since S2 can directly be followed by S4.

The fusion of all states within an ABB state group represents the expected system behavior at the entry of the respective ABB. The resulting predictive abstract system state is a union of the individual (task) information fields of each involved AbSS. If a field provides different values in different states, we insert a "no information" marker leading to an imprecise state for this task field; otherwise the task's state is unambiguous at this point. To give an example: if a task is marked as ready in S3, but suspended in S6, the predictive AbSS cannot provide information about this task; the task's state is not predictable at ABB Ⓑ. On the other hand, if a task is denoted as suspended in each AbSS of a group, it is surely known to be suspended on entry of the corresponding ABB. The resulting fine-grained prediction of the system behavior, finally, leverages various cross-kernel optimization strategies.

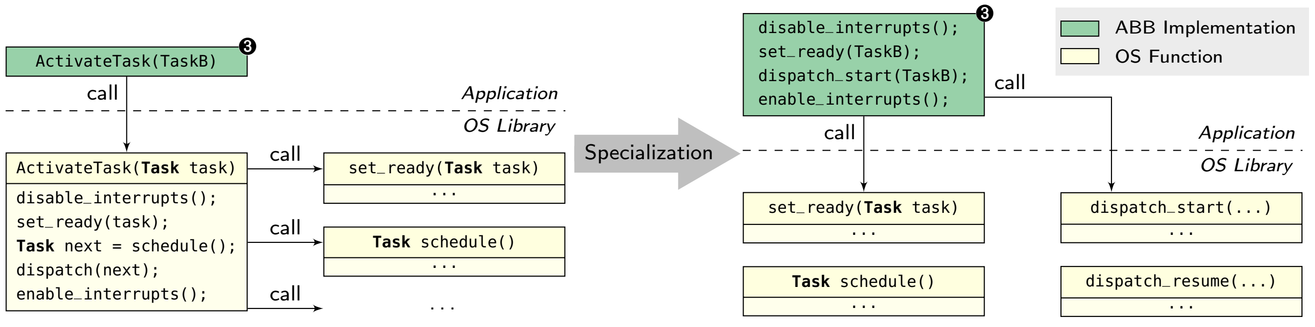
4. Application Scenarios

With the system analysis, we have gained two pieces of fine-grained knowledge about the interaction between application and kernel: First, GCFG edges, with system-call blocks as sources, represent all possible scheduling decisions after returning from the system call. Second, the predictive system states, computed for every ABB, describe the system before the block is executed. With this fine-grained information, we can optimize the whole system towards different non-functional properties. In the following, we present two different optimizations that are enabled by the fine-grained information. The measures are discussed in detail next; the evaluation follows in Section 5.

4.1 System-Call Specialization

Most OSEK implementations are shipped as library operating systems. The operating-system library, which might be tailored with the coarse-grained knowledge from the OIL file, is linked against the application to generate the system image. Each system service is a function in this library and a system call boils down to a function call into the operating system library (see Figure 5a). This approach forces the system-service implementation to be generic, since the OS developer has no control where the function is called.

With fine-grained interaction knowledge at hand, we can tailor the system calls more specifically to the application behavior in order to speed up the kernel execution paths. Instead of calling the generic system service, we insert a specialized service at the call site. This decoupling enables us to use the interaction knowledge for selecting the minimum necessary functionality at that point.



(a) Common OSEK Library OS

(b) Specialized System Calls

Figure 5: System-call specialization uses the GCFG to extract possible scheduling decisions and generate a tailored system service instances for each call site. In our example, `TaskB` is known to be the highest priority task at ABB ❸ and can be dispatched without invoking the scheduler.

In Figure 5b, the `ActivateTask` system call from the running example is specialized. Since ABB ❸ has only one direct successor at `TaskB`, we do not have to call the scheduler; we already know the result in all cases and can directly dispatch `TaskB`. Even more, from the fact that all GCFG edges lead to the entry block of `TaskB`, we will never resume but always start `TaskB` from the beginning. As the references to system objects are inserted as constants, the compiler might even inline the dispatching mechanism, here. In cases, we do not have enough information to insert a specialized version of the system service, we fall back to call a default implementation.

Not only scheduler invocations can be avoided. Other, complex system operations can be substituted by mechanisms that only update the system state with a pre-calculated result of the operation. For example, if the dynamic priority of a task is unambiguous after a `ReleaseResource` system call, we do not have to determine it at run time, but can update the OS state with a constant value. This might even boil down to a single memory write to a constant address.

However, even if the result of a scheduling operation cannot be calculated completely at compile time, it might be worthwhile to insert a *partial specialization* of the schedule operation. From the GCFG edges we can tell all possible scheduling outcomes. This information can then be used to tailor a scheduling operation that checks only for potentially runnable tasks. As their number is typically much lower then the number of all tasks, this particularly pays off if the scheduler's computational complexity depends on the number of tasks (e.g., $O(n)$). The downside of this tailoring is an increased code memory use, due to the extensive specialization.

4.2 Assertions on the Predicted System State

Besides the kernel execution time, also the resilience against transient hardware faults is a nonfunctional property of the operating system; can the kernel detect, or even recover from, a bit flip in its data structures? Caused by shrinking hardware structure sizes and lower operating voltages, the problem of transient hardware faults becomes increasingly important for automotive and other safety-critical control applications.

Software-based dependability measures allow for selective and resource efficient robustness improvements. Generally, such measures, as for example triple modular redundancy and checksumming, are dynamic by nature; they check integrity by comparison of dynamically computed values. With fine-grained interaction knowledge at hand, we have compile-time information about the dynamic behavior of the application. Therefore, we can derive constraints that must hold for all possible execution paths. We enforce these constraints with run-time assertions at each system call: The predictive system states express the knowledge we have about the system before

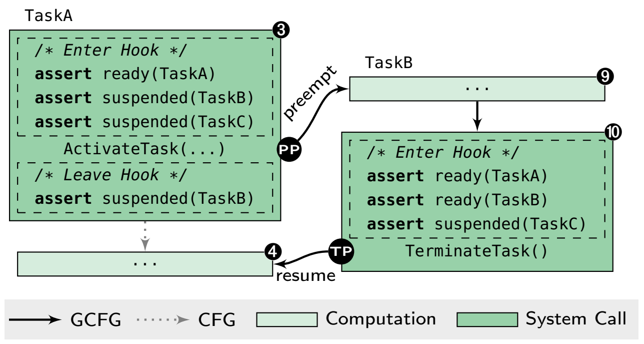


Figure 6: Kernel enter and leave hooks, which are executed atomically with the system-call, provide assertions on the system state.

an ABB is entered. Checkable pieces of the system state are, for example, the task state or the resumption points of preempted task.

These assertions not only allow to detect corruptions in the kernel memory, but also errors in the control-flow. Undesired, faulty jumps beyond the next expected system call, or even into another task's control-flow, are detectable, if the predicted system state does not match the current kernel state. This allows to collect different types of constraints for each system-call block, and generate code that is executed *atomically* with the system call. We achieve atomicity by substituting the invocation at system-call site with a code sequence that is enriched by *kernel enter* and *kernel leave hooks*. Figure 6 depicts such hooks, as well as the assertions on the tasks' states for ABB ❸ and ABB ❿ of our running example. We extract the constraints for a system call by inspecting different system states for enter and leave hook. The enter hook is filled with the predictions of the system-call block itself. Since the leave hook is executed only after the preemption point (PP), we use the predictive system state of the system-call's local CFG successor. In the example (Figure 6), the enter hook is filled with constraints from ABB ❸, while the leave hook uses the predictions from ABB ❹.

The independent collection of constraints for enter and leave hooks leads to duplicate assertions. We can avoid double checks during a kernel activation to save run time and code size. Each synchronous kernel activation consists of the system-call's enter hook, the system call itself, and a leave hook of the resumed task. In the example, we return control from the termination point (TP) to the preemption point (PP) of `TaskA`. Therefore, the `TerminateTask()` activation consists of enter hook ❿, `TerminateTask()`, and leave hook ❸. We eradicate all assertions from leave hooks which are surely checked in all resuming enter hooks. In the example, we do not check `TaskA`'s and `TaskC`'s task state in leave hook ❸, since it is already checked in enter hook ❿.

4.3 Further Use Cases

Besides the system-state asserts, we developed a control-flow monitoring mechanism on the system level to detect faulty execution paths. Shortly sketched, we use a system-wide dominance analysis [13] to identify control-flow regions in the GCFG. These regions can only be entered through a single system call; a constraint we can enforce at run time. Furthermore, we are planning to encode the state-transition graph as a finite state machine, which implements exactly the scheduling behavior of the OS for each particular application. Another direction of future research is the improvement of WCET analysis by the fine-grained interaction knowledge.

5. Evaluation

As evaluation platform, we use the existing *d*OSEK operating-system generator. It is designed as a dependable operating system that is resilient against transient hardware faults in memory and registers [10]. The generative approach of *d*OSEK is a perfect fit for the presented analysis and optimizations. *d*OSEK is available in two basic configurations: *unprotected* and *protected*. Only the protected variant includes dependability measures against transient hardware faults. The presented analyses and optimizations were integrated into *d*OSEK, which is available as free software.

5.1 Scenario

Our evaluation scenario is based on a realistic system workload considering all essential OS services. We use a setup resembling a real-world safety-critical embedded system in terms of a quadrotor helicopter control application. The tasks are activated both periodically and sporadically by an interrupt. Inter-task synchronization is realized with OSEK resources and a watchdog task, observing the remote control communication.

Checkpoint markers replace the application logic, since we are only interested in the *interaction* between application and OS. The substitution does not change the GCFG or the analysis, but only touches the contents of the computation blocks. In total, the scenario consists of eleven tasks, three periodic interrupts (alarms), one sporadic interrupt, and one resource. The first analysis step emits 91 ABBs, and 53 system-call blocks.

5.2 SSE Effectiveness: GCFG Sparseness

With our system analysis, we gather fine-grained knowledge about the interaction between application and the operating system. We will quantify the amount of knowledge we can gather with different methods by comparing their predictive power regarding all system execution sequences. The higher the predictive power of a method, the more *impossible* execution sequences are sorted out.

For each method, all possible execution sequences are combined in a GCFG with ABBs as nodes. The lesser edges this graph has, the lesser execution sequences are possible and the higher is the predictive power. For example, the least informative GCFG, with the worst predictive power, is the fully connected graph. It proposes that every block–block transition is possible; every ABB can be followed by any other ABB.

Table 2 shows different degrees of knowledge we can gather for the evaluation scenario. The fully connected graph consists of 91 nodes and 8,281 edges including self loops. Its transitions can be constructed without any knowledge of the application logic nor the system's configuration. If we include the application logic and the system-call locations, we get a sparser graph: Every computation block can only be followed by its CFG successors or an ISR entry block, each system-call block can resume to every computation block. For the scenario, this reduces the number of edges to 2,809.

GCFG Edges	w/o Annotation	w/ Annotation
No Information	8,281	
+ Application Logic	2,809	
+ System Configuration	1,642	1,580
System-State Enumeration	373	304

Table 2: Sparseness of different GCFGs with 91 ABBs. With more detailed application knowledge, we can construct sharper GCFGs.

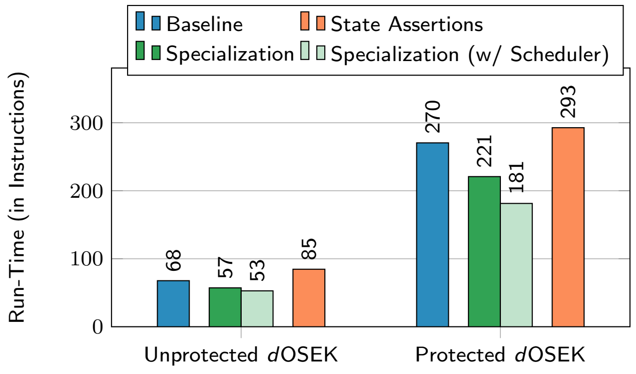


Figure 7: Average kernel runtime of *d*OSEK system calls (n=420).

The system configuration, like it is specified in the OIL file, reduces the number of edges even further: Every computation block can followed by its CFG successors and those ISR entry blocks that are activatable in this block. Each system-call block that terminates a task can proceed to every computation block of another task. Non-terminating system calls can dispatch to their CFG successor and blocks of higher priority tasks. Without the qualitative information about the interarrival time of interrupts, which was discussed in Section 3.2, the resulting graph has 1,642 edges. With the annotations, which prohibit the reactivation on an interrupt within an annotated task group, the graph has 1,580 edges.

With the presented SSE method, lesser edges are emitted as possible block–block transitions. The analysis results in 373 edges and removed 95.5 percent of all possible block–block transitions from the fully connected graph. With additional information about interrupts, the graph size reduces even further to 304 edges.

The user-supplied annotations do not only decrease the number of edges, but also reduce the analysis time significantly. With task-group annotations, the analysis, which is implemented in the Python scripting language, drops from 453.32 seconds to 1.62 seconds. The resulting state-transition graph shrinks from 2,068,143 states and 2,497,541 transitions to 16,263 states and 18,513 transitions.

5.3 Runtime of Specialized Kernel Fragments

The presented method increases the amount of application knowledge significantly. But what influence can we take on the nonfunctional properties of the system? First, we quantify the time the *d*OSEK kernel takes to execute. Therefore, the system executes for three hyperperiods on an IA-32 emulator, while, at the same time, an execution trace is recorded. During the benchmark, 420 system calls are issued.

Figure 7 shows the average time the system remains in the kernel for a system call. We measure the time in instructions, although this number is not linearly correlated to execution time on modern pipelined, out-of-order processors. Nevertheless, instruction counts

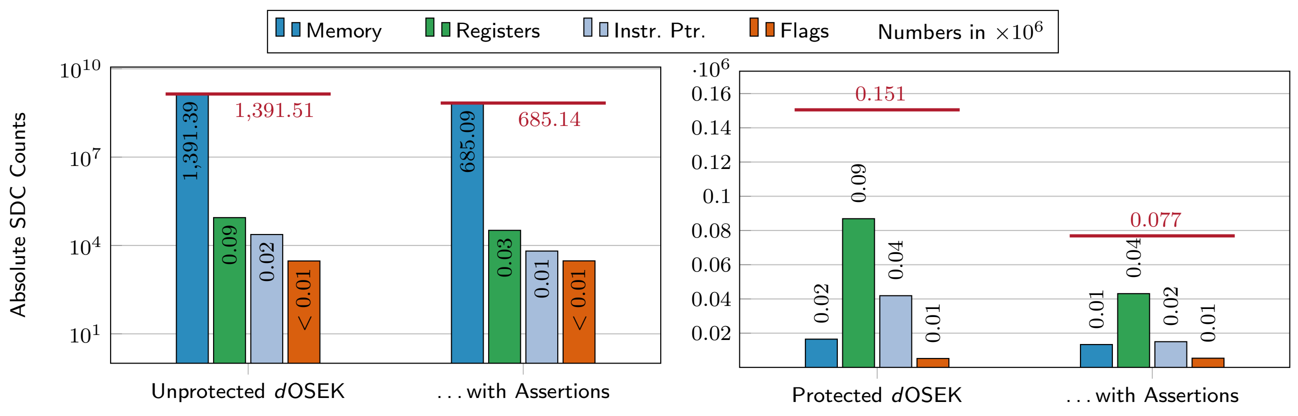


Figure 8: Absolute SDC Counts. Influence of inserting state assertions on the unprotected *d*OSEK (logarithmic scale, left) and the protected *d*OSEK (linear scale, right).

are more comparable over different hardware versions. Further, many of the mentioned CPU features are not yet available for embedded platforms, which are mainly used for real-time systems.

The unprotected *d*OSEK needs, in average, 68 cycles for a system call. If we enable the system-call specialization without support for instantiating a partial scheduler, we gain 16 percent. Additional support for partial schedulers, which are instances of the scheduling function that check not all tasks for readiness, we reduce the number of cycles by 22 percent.

The improvement for the protected *d*OSEK is even larger, since the protected kernel operations are much more expensive than in the unprotected system. It starts on average with 270 cycles for the unmodified, but protected, *d*OSEK system. Specialization, without touching the scheduler, reduces the kernel time by 18 percent. If incorporating partial scheduler instances, we decrease the run time by 33 percent, compare to the unmodified version.

Inlining the system service into the system-call sites has an impact on the code size. *d*OSEK always creates an inlined instance of the system service for each call site. This inlining increases the resilience against hardware faults, by avoiding function calls within the kernel execution. For the benchmark, baseline *d*OSEK requires, in average, 75 bytes per system-call site for the unprotected, and 189 bytes for the protected variant. Compared to this, the system-call specialization reduces the code size for each system call to 66 bytes (unprotected), respectively 161 bytes (protected).

Of course, when integrated into a commonly developed library-based OSEK, the system-call specialization will in general *not* reduce the code size. Nevertheless, the code overhead per system-call site will be in the same range as for *d*OSEK.

5.4 SDC Count Decrease

The enrichment of the system with system-state assertions is a measure against transient hardware faults. Therefore, we used the FAIL* [21] fault injection framework for an extensive injection campaign on the presented evaluation scenario. We used a single-event, single-bit fault model, which can, for example, be caused by transient hardware faults due to radiation or voltage fluctuations. According to our fault model, a single-bit flip occurs at one point in time in one location that is visible on the instruction set architecture. As locations, we do not only examine the memory, but also general-purpose registers, flag register, and the instruction pointer.

The benchmark scenario, which was augmented with checkpoint markers, runs for three hyperperiods, while, at the same time, visits 172 checkpoints. It is the operating system's task to adhere this checkpoint sequence, even in the presence of hardware faults. If the kernel cannot provide the correct activation order or corrupts the application data, and is not able to detect the fault, we record a *silent data corruption* (SDC). If the fault was not benign, but the kernel detected it, we hand over control to the application; the fault is *not* counted as a SDC.

With FAIL*, we executed the system $3.95 \cdot 10^6$ times with the deterministic BOCHS [12] IA-32 emulator. Into each execution, we injected a single single-bit flip into the operating systems' memory or registers and observed the system's reaction. Due to this focus on the OS execution, the replacement of the application logic does not influence the results. For the benchmark, the injected faults cover the *entire* effective fault space, which consists of $6.1 \cdot 10^{10}$ faults. Over all versions, 96.6 percent of the fault space was either benign or detected by the systems.

In Figure 8, we present the results of the injection campaign in absolute SDC counts for four different configurations. We start with the unprotected baseline *d*OSEK with $1{,}391.51 \cdot 10^6$ SDCs for the scenario. Nearly all of these SDCs stem from faults in the main memory, since this variant does not protect the OS state at all.

The system-state assertion optimization inserts 748 assertions into 48 system-call sites; 639 assertions were introduced into enter hooks and 109 assertions into the leave hooks. The insertions add in average 203 bytes of code to a system-call site and increase its runtime by 25 percent (see Figure 7). By spending these overheads, we reduce the SDC count by 51 percent. These improvements originate mostly form the protective nature of the measure on the OS state in memory. In total, faults in registers, instruction pointer, and flags are negligible for the unprotected *d*OSEK.

The protected *d*OSEK has several magnitudes better starting conditions. For the baseline, the system reveals $0.15 \cdot 10^6$ SDCs for the benchmark. Due to encoded operations, memory SDCs dropped significantly, while the other fault locations remain on the same level as in the unprotected system.

If we insert the same number of assertions into the protected *d*OSEK, we can further reduce the number of SDCs by 49 percent. Here, the improvements stem from reducing register faults that occur during the kernel operation. The system-state assertions are able to detect incorrect results directly after they have been written back to memory in the kernel leave hooks.

6. Discussion

One of the main challenges in analyzing OSEK systems with the presented approach is the size explosion of the state-transition graph, which is generated by the system-state enumeration. Potentially, a system can have an exponentially higher amount of states compared to the number of tasks and basic blocks. Therefore, measures to ease this exponential burden are crucial. Furthermore, they will, as we believe, also give insightful design principles for real-time systems in general.

6.1 Design Recommendations to Tackle the State Explosion

With the *atomic basic block* (ABB) abstraction, we reduce the number of blocks in the system significantly. Subsuming blocks that do not interact with the kernel sharpens the focus on the application–kernel interaction and abstracts from the application's microstructure. Whole library hierarchies and algorithms can be hidden within a single ABB. As a general design principle, we further recommend to avoid system calls that modify the kernel state, deep in the call hierarchy. Although deeply buried system calls do not impede our analysis, they still result in many split basic blocks and thus complex and confusing GCFGs. Often, such hidden system calls reveal surprising side effects and activation sequences. In order to reduce the complexity of the analysis – and thereby of the entire system – we suggest to plan the real-time application in large computation blocks, which are not sliced by synchronous syscalls.

6.2 Interrupt Handling

The other main drivers of state explosion are interrupts. Interrupt requests fork the state-transition graph in every computation block. Nevertheless, in real-time systems, interrupts normally are *not totally* unpredictable. In order to maintain analyzability, developers already determine minimal and maximal inter-arrival times. Furthermore, parts of the application are synchronized with these signals. With the task-group annotation, we let the developer express this knowledge about signal–signal-handling causality. In all cases, the system architect should answer some questions: What is the handling task group for a physical signal? Where does the handling end, and how are different groups synchronized?

It is also conceivable to take further knowledge of the peripheral-device behavior regarding interrupts into account. Here, a *logic of actions* on application level could be derived. For example, it could be defined, that a "send buffer empty" interrupt can only be triggered within a specific time after the associated `SendMessage()` function was invoked.

6.3 Scalability of the Analysis

As a rough estimation, we can assume that the SSE will scale linear in run-time to the number of system states. Of course, we cannot deduce how large a system may be, to remain manageable. Nevertheless, we already can handle real-world scenarios without having further assistance by the developer. As a topic of further research, methods from the symbolic-execution community could be applied to cut down on the analysis run time, and other methods could be developed to construct the global control-flow graph.

6.4 Threats to Validity

The major threat to validity of the experimental findings is that they are based on single case study only. The flight control of a quadrotor flying vehicle is a real-world safety-critical system. Therefore, we consider it representative for these kinds of applications, not only in its size, but also regarding operating system interaction.

As threat to the approach in general, we consider the limitations that were put on the applications. System calls must be fixed in their locations, types, and arguments. We forbid the invocation of system-calls through pointers with variable arguments. Nevertheless, we consider this flexibility of kernel API usage unnecessary and undesirable for (safety-critical) real-time systems, where predictable behavior and analyzability is mandatory.

Current restrictions, like event support, multiple tasks per priority as well as multiple task activations are no conceptual problem for our analysis. Event support is currently under development. Multiple task per priority can be mapped to the existing model by letting all tasks of one priority share a common resource to serializes their execution. Multiple task activations can be modeled by an activation queue within the AbSS, finally leading to a larger state graph.

Another aspect is the necessity of an unambiguous description of the OS behavior to derive an accurate `systemSemantics` function, which is the key to our analysis. This dependency on a clear specification, or at least description, seems inconvenient in the first place. However, embedded real-time operating systems generally provide enough information to deduce the necessary information. Regarding safety-critical systems, the use of a strictly specified operating system is often prescribed, anyway. Here, our approach can even support the development process: Incorrect API usage within the application code, violating the specified OS semantic, is immediately uncovered by the GCFG analysis.

7. Related Work

Bertran et al. [5] proposed a global view on the interaction between operating system and application. They constructed a global control-flow graph for a complex embedded system, which was built on top of Linux. System-call entry points and library entry points were connected to the corresponding call sites. On this GCFG, dead code elimination in terms of removing uncalled system calls and unreferenced library functions resulted in a reduced code size of the system image. In contrast to this work, their analysis was flow-insensitive and did not take the semantic of system calls into account. Basically, they extended the CFG into the kernel, but not out of it.

Barthelmann [3] makes use of the static semantics of an OSEK system to minimize the task contexts to be saved at specific preemption points. A static analysis reveals an *interference graph* describing mutual preemptions of basic blocks, based on the tasks' static priorities. With this knowledge, an optimized *inter-task* register-allocation is performed including context-switch code generation. Similar to our approach, application knowledge is used to influence a non-functional property of an OSEK kernel. In contrast to our work, the scheduling semantic of OSEK was used in a flow-*in*sensitive manner. That means, the interference graph includes superfluous preemptions that are actually impossible according to a GCFG analysis. This approach corresponds to the "System Configuration" case from Table 2. Nevertheless, the paper describes a first approach of a generative whole-system optimization taking both the application and the operating system into account.

The OSEK semantic also found attention in the area of formal methods and verification: Waszniowski and Hanzálek [23] designed a model of the OSEK standard targeting the UPPAAL model checker. They modeled all components as timed automata, also taking inter-process communication (OSEK events) into account. Their main focus was the verification of different application properties and schedulability analyses. Huang et al. [11] modeled OSEK as communicating sequential processes (CSP). The application subtasks were modeled without considering the internal application structure; interrupts were excluded entirely. With this model, they could verify different properties of their OSEK system, like deadlock freedom and freedom of priority inversion. Regarding our approach, these models could provide a more formal definition of the `systemSemantics` function.

System specialization was already discussed by the operating-system community for general-purpose operating systems. Pu, Massalin, and Ioannidis [19] developed the Synthesis kernel, which included a code synthesizer that produced optimized code paths at run time for often invoked system calls, like for example `read` or `write`. Due to manual implementation of code templates, which are then filled by the synthesizer, huge performance benefits arouse from shorter kernel execution paths. In comparison to the dynamic Synthesis system, our approach of the system-call specialization also takes the in-depth application knowledge into account but is executed off-line. Pu et al. also mention the problem of code-size explosion. McNamee et al. [15] used *Tempo*, a partial evaluator for C programs, and a set of specialization predicates to identify functions automatically for specialization within the kernel. In their approach, the specialization was also done dynamically at run time, omitting detailed static application knowledge.

Several approaches towards control-flow monitoring were developed for application logic. Benso et al. [4] employ regular-expression automata to check the correctness of executed basic block sequences of an application. Oh, Shirvani, and McCluskey [18] presented an approach with *software signatures*. Each basic block is assigned a unique number; when the basic block is entered or left, a global variable is xor'ed with this number. Without control-flow errors, the global variable contains always exactly the unique number of the currently executed basic block. Yau and Chen [24] divided the control-flow graph into loop-free regions. For each region a database of possible paths is encoded and checked during the execution. All mentioned approaches only consider the control-flow graph of a single function or task, but could be extended to catch control-flow errors on the ABB and GCFG level.

8. Conclusion

Real-time systems include a large amount of concealed static knowledge about their dynamic behavior. With the presented methods, we make this knowledge accessible for OSEK-like systems and exploit it to optimize nonfunctional system properties. The system-state enumeration computes the *global control-flow graph*, which covers all possible system execution paths, by combining application logic, the system configuration, and the operating-system specification. We employ the fine-grained interaction knowledge to inline *specialized system services* into the system-call sites. Furthermore, *system-state asserts* check statically derived constraints at run time. With these applications of our fine-grained knowledge, we could speed up the kernel execution path by 33 percent and decrease the soft-error vulnerability by 49 percent.

Source Code and Raw Data

The analysis source code, which is published as free software under GPLv3+, is available at github.[1] The results that lead to the numbers in this paper were calculated by an automated experiment workflow. The raw results are available at our website.[2]

[1] https://www.github.com/danceos/dosek

[2] https://www4.cs.fau.de/Research/dOSEK/data/lctes15/

References

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

[2] Frances E. Allen. "Control Flow Analysis". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[3] Volker Barthelmann. "Inter-Task Register-Allocation for Static Operating Systems". In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES '02)*. (Berlin, Germany). New York, NY, USA: ACM, 2002, pp. 149–154. ISBN: 1-58113-527-0. DOI: 10.1145/513829.513855.

[4] A Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri. "Control-flow checking via regular expressions". In: *10th Asian Test Symposium 2001 (ATS '01)*. (Kyoto, Japan). Washington, DC, USA: IEEE, 2001, pp. 299–303. DOI: 10.1109/ATS.2001.990300.

[5] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. "Building a Global System View for Optimization Purposes". In: *W'shop on the Interaction between Operating Systems and Computer Architecture (SCA-WIOSCA '06)*. (Boston, USA). Washington, DC, USA: IEEE, 2006.

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

[7] Jim Cooling. *Software Engineering for Real-Time Systems*. AW, 2003. ISBN: 0-201-59620-2.

[8] Christoph Erhardt, Michael Stilkerich, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study". In: *JTRES '11: 9th Int. W'shop on Java Technologies for real-time & embedded systems*. (York, UK). New York, NY, USA: ACM, Sept. 2011, pp. 96–105. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043927.

[9] *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*. Oct. 2004. ISBN: 0-9524156-2-3.

[10] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. "dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel". In: *21st IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '15)*. Accepted. Washington, DC, USA: IEEE, 2015.

[11] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, and Jianqi Shi. "Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP". In: *5th Int. Symp. on Theoretical Aspects of Software Engineering (TASE'11)*. (Xi'an, China). Washington, DC, USA: IEEE, 2011, pp. 142–149. DOI: 10.1109/TASE.2011.11.

[12] Kevin P. Lawton. "Bochs: A Portable PC Emulator for Unix/X". In: *Linux Journal* 1996.29es (1996), p. 7.

[13] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *ACM Trans. Program. Lang. Syst.* 1.1 (1979), pp. 121–141. ISSN: 0164-0925. DOI: 10.1145/357062.357071.

[14] Peter Marwedel. *Embedded System Design*. Heidelberg, Germany: Springer, 2006.

[15] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. "Specialization Tools and Techniques for Systematic Optimization of System Software". In: *ACM Trans. Comp. Syst.* 19.2 (May 2001), pp. 217–251. ISSN: 0734-2071. DOI: 10.1145/377769.377778. URL: http://doi.acm.org/10.1145/377769.377778.

[16] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2014-09-29. OSEK/VDX Group, 2004.

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

[18] N. Oh, P.P. Shirvani, and E.J. McCluskey. "Control-flow checking by software signatures". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 111–122. ISSN: 0018-9529. DOI: 10.1109/24.994926.

[19] Calton Pu, Henry Massalin, and John Ioannidis. "The Synthesis Kernel". In: *Computing Systems* 1.1 (1988), pp. 11–32.

[20] Fabian Scheler and Wolfgang Schröder-Preikschat. "The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems". In: *13th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '10)*. (Carmona, Spain). Washington, DC, USA: IEEE, May 2010, pp. 34–41. ISBN: 978-0-7695-4037-5. DOI: 10.1109/ISORC.2010.11.

[21] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. "FAIL*: Towards a Versatile Fault-Injection Experiment Framework". In: *25th Int. Conf. on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*. (Munich, Germany). Ed. by Gero Mühl, Jan Richling, and Andreas Herkersdorf. Vol. 200. Lecture Notes in Informatics. Gesellschaft für Informatik, Mar. 2012, pp. 201–210. ISBN: 978-3-88579-294-9.

[22] O. Shivers. "Control Flow Analysis in Scheme". In: *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '88)*. (Atlanta, GA, USA). PLDI '88. New York, NY, USA: ACM, 1988, pp. 164–174. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54007.

[23] Libor Waszniowski and Zdeněk Hanzálek. "Formal Verification of Multitasking Applications Based on Timed Automata Model". In: *Real-Time Systems* 38.1 (Jan. 2008), pp. 39–65. ISSN: 0922-6443. DOI: 10.1007/s11241-007-9036-z.

[24] S.S. Yau and Fu-Chung Chen. "An Approach to Concurrent Control Flow Checking". In: *IEEE TOSE* SE-6.2 (1980), pp. 126–137. ISSN: 0098-5589. DOI: 10.1109/TSE.1980.234478.