Levels of Specialization in Real-Time Operating Systems

Björn Fiedler, Gerion Entrup, Christian Dietrich, Daniel Lohmann
Leibniz Universität Hannover
{fiedler, entrup, dietrich, lohmann}@sra.uni-hannover.de

***Abstract*—System software, such as the RTOS, provides no business value on its own. Its utility and sole purpose is to serve an application by fulfilling the software's functional and nonfunctional requirements as efficiently as possible on the employed hardware. As a consequence, every RTOS today provides some means of (static) specialization and tailoring, which also has a long tradition in the general field of system software.**
However, the achievable depth of specialization, the resulting benefits, but also the complexity to reach them differ a lot among systems. In the paper, we provide and discuss a taxonomy for (increasing) levels of specialization as offered by (real-time) system software today and in the future. We argue that system software should be specialized as far as possible – which is always more than you think – but also discuss the obstacles that hinder specialization in practice. Our key point is that a deeper specialization can provide significant benefits, but requires full automation to be viable in practice.

I. Introduction

While the domain of real-time control systems is broad and diverse with respect to both, applications and hardware, each concrete system has typically to serve a very specific purpose. This demands specialization of the underlying system software, the real-time operating system (RTOS) in particular: An "ideal" system software fulfills exactly the application's needs, but no more [19]. Hence, most system software provides built-in static variability: It supports a broad range of application requirements and hardware platforms, but can be specialized at compile-time with respect to a specific use case. Historically, this has led to the notion of system software as *program families* [25], [14] as well as a myriad of papers from the systems community that demonstrate the efficiency gains by specializing kernel abstractions to the employed application, hardware, or both. Examples include [27], [6], [20], [26].

A. System Software Specialization

Specialization (of infrastructure software) for a particular application–hardware setting is a process that aims to improve on nonfunctional properties of the resulting system while leaving the application's specified functional properties intact. If the application employs an RTOS with a specified API and semantics (e.g., POSIX [2], OSEK/AUTOSAR [4], ARINC [3]), a specialized derivative of the RTOS does no longer fulfill this API and semantics in general, but only the *subset* used by *this concrete application* and hardware. If successful, this specialization leads to efficiency gains with respect to memory footprint, hardware utilization, jitter, worst-case latencies, robustness, security and so on; it increases the safety margins or makes it possible to cut per-unit-costs by switching to a cheaper hardware. For price-sensitive domains of mass production, such as automotive, this is of high importance [8].

This work was partly supported by the German Research Foundation (DFG) under grant no. LO 1719/4-1

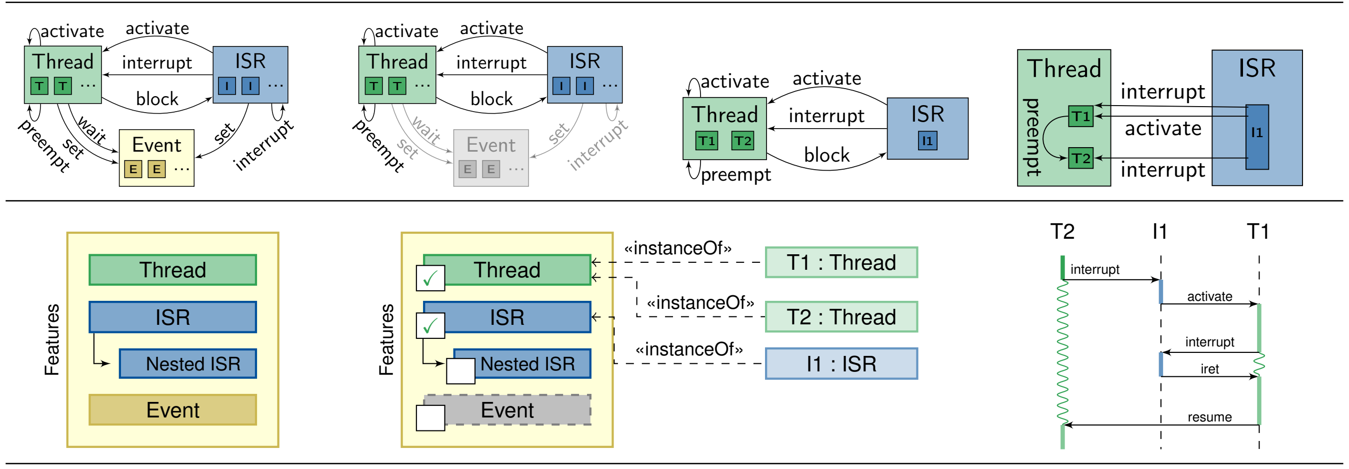
Intuitively, any kind of specialization requires knowledge about the actual application: The more we know, the better we can specialize. In the domain of real-time systems (RTSs), we typically know *a lot* about our application and its execution semantics on the employed RTOS: To achieve real-time properties, all resources need to be bounded and are scheduled deterministically. Timing analysis depends on the exact specification of inputs and outputs, including their inter-arrival times and deadlines; schedulability analysis requires that all inter-task dependencies are known in advance – and so on.

Even though all this knowledge should pave the road to a very rigorous subsetting of the RTOS functionality, this rarely happens in practice. Part of the problem is that the specialization of the RTOS typically has to be performed manually by the application developer or integrator, which adds significant complexity to the overall system development and maintenance process. We are convinced that automation is the key here, as most of the required knowledge could be extracted by tools from the application's code and design documents – the RTOS specialization should become an inherent part of the compilation process, like many other optimizations.

Another part of the problem is, however, that static specialization itself is only rarely understood. This holds in our observation for both, RTOS users and RTOS designers, both of which typically have been educated (and tend to be caught) in the mindset and APIs of general-purpose operating systems, such as Linux or Windows. So while every system software provides *some* means for static specialization and tailoring, the rigorosity at which this (a) could be possible in principle, (b) is possible in the actual RTOS provisioning, and (c) is employable by users in practice, differs a lot.

B. About This Paper

Our goal with this paper is to shed some light on the aspects and the fundamental levels of specialization that are provided by system software today and, maybe, in the future. We claim the following contributions: (1) We provide a classification for specialization capabilities on three increasing levels (Section II). (2) We discuss the challenges and benefits of system specialization by examples from the literature (Section III). (3) We show, on the example of a small experiment with FreeRTOS [5], the potential of different specialization levels, even for an RTOS API that is supposed to "look like POSIX" (Section IV).



(g) Generic	**(a) Abstractions**	**(b) Instances**	**(c) Interactions**
RTOS API/standard, like POSIX [2], OSEK/AUTOSAR [24], or ARINC [3]. Supports any application.	Like Linux or eCos [22], specialized for applications that employ threads and ISRs, but not events. Supports a class of applications.	Like ERIKA [1], specialized for an application that employs well-defined threads T1 and T2 and ISRs I1. Supports a specific application.	Like dOSEK [10], specialized for the concrete interactions that happen in a specific application. Supports a concrete implementation.

Fig. 1: Levels of RTOS specialization. From left to right, each level further constraints how the application may use the kernel.

Many aspects about specialization we describe in this paper are based on our own experience with the design, development, and employment of highly configurable application-tailorable system software. We apologize for the (shameless) number of self-citations, but felt that leaving them off would not have contributed to the accessibility of the paper.

II. A Taxonomy of Specialization Levels

In this section, we give a taxonomy of system specialization and the different levels specialization can reach. In short, a generic RTOS (g) can be specialized by (a) removing complete abstractions (e.g., threads or a specific syscall), (b) make instances fixed (e.g., there are only threads T1 and T2), and (c) make interactions fixed (e.g., only T1 waits on event E1). We examine these terms at the example of RTSs, which we specify for the purpose for this paper as follows:

A (hard) real-time system *RTS* consumes time-labeled input events $\vec{\mathrm{I}}$ and produces observable, time-labeled output events $\vec{\mathrm{O}}$, while fulfilling strict timing constraints between both event streams. An implementation $RTS^{\mathrm{A}}_{\substack{\mathrm{RTOS}\\ \mathrm{HW}}}$ of the abstract *RTS* consists of a concrete application *A* that runs, mediated by a concrete RTOS implementation *RTOS*, on a concrete hardware *HW*. We encapsulate the specification and timing requirements of the *RTS* in an equality operator $\overset{RTS}{=}$ that compares two outputs.

$$RTS(\vec{\mathrm{I}}) = \vec{\mathrm{O}} \quad \overset{RTS}{=} \quad RTS^{\mathrm{A}}_{\substack{\mathrm{RTOS}\\ \mathrm{HW}}}(\vec{\mathrm{I}})$$

Every *correct* implementation of *RTS* produces an output stream that is equal, under the *RTS* specification, to the outputs of the abstract/ideal *RTS*. Therefore, we derive: Every specialized implementation ${RTS^{\mathrm{A}}_{\substack{\mathrm{RTOS}'\\ \mathrm{HW}}}}$ has to be a correct implementation of *RTS* and the observable outputs must not change with respect to the specification of the real-time system.

However, not every $RTS^{\mathrm{A}}_{\substack{\mathrm{RTOS}\\ \mathrm{HW}}}$ is a specialized implementation. Specialization is the process of reducing flexibility from one or more system components of an already existing implementation. For real-time systems, it can take place in the application *A*, the *RTOS*, or/and the hardware *HW*. For the rest of the paper, we focus on the specialization of the RTOS, while application and hardware remain unchanged.

The specialized $RTOS'$ fulfills all requirements of the specific application that runs on top and works on the specified hardware. However, this $RTOS'$ does not necessarily provide the correct semantics to execute an alternative A' or correct instructions to execute on an alternative HW'. Therefore, RTOS specialization always depends on the application that uses the RTOS and the targeted hardware.

In the following we exemplify this by a simple RTOS that supports only three abstractions: Threads, interrupt service routines (ISRs) and Events. Figure 1 (g) shows the whole range of functions provided by our example RTOS as an *interaction graph*. Nodes are system abstractions that are provided by the RTOS standard; edges are interactions between them. The generic RTOS (i.e., the respective standard) provides the illusion that abstractions can be instantiated arbitrarily often and all instances (nodes within nodes) can interact according to their abstraction. For example, every ISR can activate every thread.

When we specialize our generic RTOS, we (a) remove abstractions, (b) make instances fixed, and (c) forbid concrete interactions. The shrunk interaction graph reflects the reduced flexibility of the specialized RTOS. We define three levels of specialization, which subsequently need more information about the actual interaction graph of the application and remove more flexibility. Every level is a true superset of the previous one.

Specialization of Abstractions: remove complete abstractions and types of interactions.

Specialization of Instances: number and identity of instances become fixed; dynamic instantiation is not possible.

Specialization of Interactions: interactions are constrained to concrete instances instead of (generic) abstractions.

The following sections describe the levels in detail and outline the information is needed to reach the respective level. If we specialize a RTOS implementation to a certain level, it only ensures that applications with the corresponding interaction graph are executed correctly. For all other applications, the result is undefined. The effects of the specialization levels (Figure 1 (a)-(c)) are examined using the following example application code:

```
BoundedBuffer bb;             Thread T1 { // priority: 2
                                while(data = bb.get())
ISR I1 { // priority: ∞           handleSerial(data);
  data = readSerial();        }
  bb.put(data);               Thread T2 { // priority: 1, autostart
  activate(T1);                 while (true)
}                                 handleADC(readADC());
                              }
```

The nonpreemptable ISR reads serial data into a bounded buffer, which is handled by the higher-priority worker thread T1. The background thread T2 continuously reads analog data and handles the result. For compactness reasons, we ignored the lost wake-up problem between I1 and T1.

A. Specialization of Abstractions

Specialization on the level of abstractions is the most generic one that is commonly used to select the availability of RTOS features. The needed knowledge to conduct this specialization is confined to the list of used abstractions, which could be derived from code or explicitly listed in a configuration. This kind of specialization is possible in most operating systems. For instance, Linux, eCos and FreeRTOS provide support to be specialized on the level of abstractions. The example application employs only threads and ISRs, while events are not used at all. Therefore, the RTOS specialized on abstractions (Figure 1 (a)) avoids everything event related. Furthermore, we can safely forgo the nesting of ISRs and, therefore, remove the “interrupt” interaction between ISRs.

B. Specialization of Instances

One level deeper, specialization of instances means to specify the concrete instances of each abstraction and their properties. In addition, knowledge about these concrete instances is necessary. For threads, this could be their name, priority, stack size, periodicity and initial activation state. Some RTOS specifications, such as OSEK, already require this information in a configuration file. For others this information may be gathered out of the source code. An instance-level specialized RTOS looses the capability to create system objects at run time. All instances need to be specified statically at compile time.

In an OSEK implementation like ERIKA [1], the OSEK Implementation Language (OIL) file [23] describes all system objects of the application and their properties. For our example application this would be two threads, namely T1 and T2 and one ISR, namely ISR1. The priority of ISR1 is ∞ and T1 and T2 have the priorities 2 and 1. In Figure 1 (b), only the three concrete instances (T1, T2, I1) remain in the interaction graph, while the interactions are still attached to the abstractions.

C. Specialization of Interactions

The most extensive specialization takes place at the level of interactions. Here, we limit the concrete interactions between the system-object instances rather than abstractions. By limiting interactions, we can derive optimized kernel paths, like removing dead code branches (e.g., syscall parameter checking). In essence, we take the viewpoint of an optimizing whole-system compiler that knows the RTOS semantics and thereby could, for instance, derive scheduling decisions already at compile time. To optimize the RTOS on this level, we have to know of all concrete interactions of our applications. This can be done by static code analysis or examination of external-event timing constraints to derive possible invocation sequences.

For our application, we can derive that there is no inter-thread activation, no interrupt blockade, and only T1 can preempt T2. Furthermore, we know that I1 can only activate T1, while it potentially interrupts both threads. This results in Figure 1 (c) contain just these interactions.

D. Summary

In summary, by specialization of the RTOS kernel we remove flexibility from the kernel implementation by restricting the possible run-time interactions of the application already at compile time. This can take place on the (subsequently stricter) levels of *(a) Abstractions*, *(b) Instances*, and *(c) Interactions*, which, in turn, subsequently cut of more from the unneeded RTOS functionality.

III. Specialization: Benefits and Challenges

In our experience, the less-is-more philosophy (i.e., it is a good thing to *reduce* flexibility) tends to be counter-intuitive for many software engineers and in any case it is arguable. In the following, we discuss some benefits and challenges of specialization in general and with respect to the different levels.

A. Benefits

Memory footprint reduction is the most obvious benefit – and still the driving factor for industries of mass production, such as automotive [8]. It is not a coincidence that OSEK (and later AUTOSAR) were designed for specialization on the instance level from the very beginning. The compile-time instantiation of kernel objects and their management in preallocated arrays instead of linked lists facilitates significant RAM savings. In [17], the transformation of an RTS from the abstraction-level specialized eCos [22] to the instance-level specialized CiAO [21] reduced the RAM footprint by half.
But also abstraction-level specialization alone can pay off, if applied systematically: The specialization of a Linux system running typical appliances, such as a LAMP server or an embedded media player, can reduce its code size by more 90 percent compared to a standard kernel [30], [28].

Security and safety improvements are less obvious, but a corollary from memory footprint reduction: What is not there can neither break nor be attacked or exploited and does not need to be later maintained in this respect. For instance, specializing the mentioned LAMP server on the level of abstractions did not only reduce its code size, but also cut the number of relevant entries in the CVE database[1] by ten percent [30]. The instance-level specialization of the RTS in [17] also increased its robustness regarding bit flips by a factor of five.

[1] https://cve.mitre.org

Further significant improvements in this respect are possible if one specializes down to the level of interactions, for instance, by inserting control-flow assertions [11].

Better exploitation of hardware by a direct mapping of RTOS abstractions. Modern μ-Controllers are not only equipped with an increasing number of cores, but also large arrays of timers, interrupt nodes and so on. Nevertheless, most RTOS implementations still multiplex a single hardware timer and IRQ context. In Sloth [16], [15] the specialization on instance-level is the prerequisite to map system objects at compile-time directly to the available hardware resources, which results in minimal kernel footprints and excellent real-time properties.
If specialization of the hardware itself is also an option, a kernel specialized on interaction-level could even be placed directly into the processor pipeline [12].

Reduction of jitter and kernel latencies is a further benefit of memory footprint reduction and the better exploitation of hardware. Intuitively, removing code, state, and indirection in the control and data flows of the kernel also reduces noise caused by memory access and cache pressure and increases determinism. Shorter kernel paths and the direct mapping of kernel objects to hardware yield a direct benefit on interrupt lock times and event latency.

Analyzability and testability is both improved as well as impaired (see below). In principle, any reduction of possible kernel states and execution paths increases determinism and makes it easier to analyze, test, and validate the kernel against the RTS specification. The model that is required for instance-level specialization can directly be used for static conformance checking to find, for instance, locking protocol violations. If specializing on interaction level, the underlying interaction model [11] further paves the path to whole-system end-to-end response-time and energy-consumption analysis [13], [31].

B. Challenges

However, specialization does not come for free. It depends on a very deep understanding of your RTS on the systems level, as well as the ability and willingness to express its properties and demands towards the RTOS. In our experience, deep specialization remains a hopeless attempt if the configuration of the RTOS is mostly based on experience and manual labor of the RTS developer. Full (or at least nearly full) automation of specialization by tools is the key to success.

You have to know what you need and this is probably the major challenge. In practice the burden is on the developer – and this already hits its limits when specialization takes place on the level of abstractions: Recent versions of Linux (4.16), but also smaller RTOSs like eCos, provide an unbearable number of configuration options (more than 17000 in Linux, respectively 5400 in eCos). Hence, most developers have long ago stopped specializing more than necessary and employ, in the case of Linux, a one-size-fits-all standard distribution kernel instead.
To be viable in practice, the RTOS configuration has to be derived automatically: In fact, the 90 percent code savings in Linux mentioned above were only achievable by an automatic specialization approach that measures the required features on a standard distribution kernel in order to derive a tailored configuration [30], [28]. Schirmeier and colleagues suggested automatic detection of required eCos features (level of abstractions) by static analysis of the application source [29]. However, they also identified limits of their approach when the decision about an abstraction (e.g., the need for a costly priority inheritance protocol in the mutex abstraction) depends on information only available on the instance or interaction level (i.e., who accesses a particular mutex at run time).
Hence, for the developer automatic configuration becomes actually easier with instance- or interaction-level specialization. As she has to think about the employed system objects anyway, specifying the requirements on the instance level is closer to the application and more natural, while the configuration tool can automatically derive the necessity of, for example, a priority inheritance protocol in mutex objects. OSEK, which is specialized on instance level, automatically derives the priority of the resource objects specified in the OIL file [23], [24].
If interaction-level information is required, a manual provisioning would become completely intractable. However, in this case static analysis of the application source code is even more promising than on the feature level: Programming is the act of writing down desired interactions between instances, which are technically expressed by syscalls, and we can use static analysis to extract these interactions. For example, Bertran et al. [7] analyze all libraries and executables of a concrete Linux system and remove system calls that cannot be activated. Furthermore, with a complete and flow-sensitive analysis of the application's execution across the syscall boundary we could retrieve a complete interaction model [11]. This, however, has exponential overhead if indeterminism by external events needs to be considered. Hence, the analysis needs to be constrained by further information that is commonly not expressed in the source code, such as event-activation frequencies.

You have to be able to express what you need is therefore another challenge – and unfortunately in many cases the RTOS interface even hinders the expression of instance-level developer knowledge [18]: Most RTOSs adhere to (or at least mimic) a POSIX-style API with dynamic allocation and instantiation of a conceptually arbitrary number of system objects at run time. This mindset stems from interactive multi-user systems (UNIX), but has to be considered as a strong misconception in the world of real-time systems – for both sides, developers and users of an RTOS. The already mentioned reductions in the kernel's memory footprint when switching from the POSIX-like eCos to the OSEK-like CiAO in [17] are rooted in the kernel-internal overhead of implementing an interface that favors (unneeded) dynamic instantiation. So, if the RTOS employs such a "flexible" syscall interface, more additional information has to be provided by the developer to enable instance- and interaction-level specialization.

Testability and certifiability is in our opinion becoming the most significant obstacle towards systematic specialization of system software. With the advent of autonomous driving features, the industry is facing new challenges with respect to functional safety; ISO 26262 and ASIL D demand the employment of a certified RTOS. While in principle the certification of a less flexible system should make this easier (see discussion of the respective benefit in the previous section), existing certification procedures mostly follow a certify-once-and-never-touch-again philosophy that is fundamentally the opposite of application-specific specialization. The certification of an RTOS kernel is extremely expensive, so vendors shy away from the even higher costs of certifying a kernel generator. However, without a certified generator, each specialized kernel instance

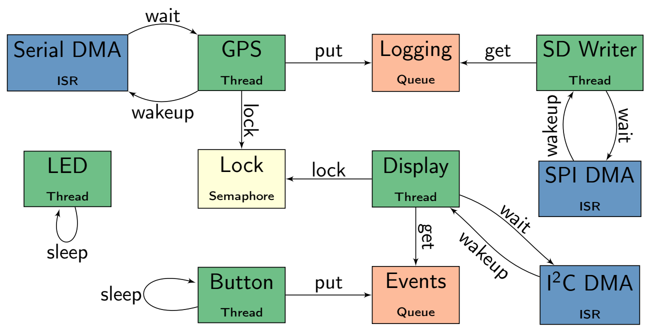


Fig. 2: Interaction Graph for GPSLogger

would have to be certified individually. In the extreme case (full interaction-level specialization) this would be necessary for every change of the application implementation. Hence, certified RTOSs, such as RTA-OS (ETAS), MICROSAR OS (Vector), or tresos Safety OS (EB) offer not more, but significantly less room for specialization.
So one has either to forgo the benefits of specialization or to swallow the bitter pill of certifying a complete kernel generator, which is highly unrealistic. A more promising direction could be to make a virtue out of necessity and extend the (automatic) specialization to the certification process as well: We do not need to validate the specialized kernel against the full RTOS specification, but only to those parts and interactions that are actually used on the concrete RTS. If the interaction model could be assumed as sound and complete, it can be employed with model checkers to automatically validate the generated kernel instance. [9]

C. Summary

Despite very high improvements regarding many nonfunctional properties, RTOS specialization is performed only half-hearted in practice, as explicit configuration puts to much burden on the developer. This is partly caused by unsuitable UNIX-inspired syscall APIs and misconceptions about “what the OS is and provides”. Hence, deep specialization requires automation to remove the burden of having to understand and know the details from the developer. The analysis of the application’s requirements interactions as well as the generation of a fitting RTOS instance has to be provided by tools.

Nevertheless, also with existing RTOSs implementations that offer a less-than-ideal API, significant savings are achievable. In the following, we exemplify this by re-analyzing an existing application running on FreeRTOS from the viewpoint of our taxonomy.

IV. An Experiment with FreeRTOS

Our example is the a freely available GPSLogger[2] application, which uses FreeRTOS [5] to orchestrate its threads. The system runs on a “STM32 Nucleo-F103RB” evaluation board that is equipped with a STM32F103 MCU. It is connected to a graphical display (I^{2}C), a GPS receiver (UART), a SD card (SPI), and two buttons (GPIO). The application consists of 5 threads, 3 ISRs, 2 blocking queues, and one binary semaphore. Due to a broken SD card library, we replaced the SD card operations with a `printf()`.

[2]https://github.com/grafalex82/GPSLogger

In Figure 2, we extracted the interaction graph for this application manually from the source code. For compactness reasons, we omitted some interactions from the figure (i.e. preempt). The inter-process communication is mainly done with blocking message queues. However, the GPS thread and the display thread bypass the kernel for the transferred data and use a shared memory region that is protected by a binary semaphore. For most IO operations, GPSLogger uses a pattern where one thread blocks passively until one DMA ISR signals the completion of a data transfer. However, for the button thread, GPSLogger uses active polling with a passive sleep. While the employment of full-blown queues is overkill to transmit small datagrams in 1:1 interactions, it is the primary abstraction offered by FreeRTOS.

a) Specialization of Abstractions: FreeRTOS provides abstraction-specialization capabilities by using conditional compilation and C preprocessor macros. However, there is no formal or semi-formal feature model, like it is provided by Linux KConfig or the eCos configuration tool, but the configuration is placed in a header file. As another specialization, unreachable functions are automatically removed by the linker, as the build system uses function- and data sections in combination with link-time garbage collection.

At this specialization level, the resulting binary uses $91{,}084$ bytes for code and $18{,}328$ bytes of mutable RAM. The kernel takes $60{,}426$ cycles of startup time, before the first task starts. Startup times were measured 100 times and the standard deviation always was below 35 cycles.

b) Specialization of Instances: For the instance level, we removed the dynamic system-object allocation in favor of statically allocating them in the data section. These system objects include the thread stacks, the thread control blocks, queues, and the ready list. FreeRTOS, since version 9.0.0, supports that the user provides a statically allocated memory to hold system objects and, thereby, gets rid of the special FreeRTOS heap. With static allocation, we use 112 more bytes of code, but save 856 bytes of RAM and $6{,}598$ cycles of startup time compared to baseline. The increase in code size stems from the additional parameter of the static object-initialization functions.

As a second step, we removed the dynamic initialization of stacks, thread-control blocks, and the scheduler. Instead, we initialized their values and pointers statically such that the data section already contains a prepared memory image to start FreeRTOS. Compared to baseline, the statically initialized GPSLogger saves 344 bytes of code and $7{,}327$ cycles of startup time. The RAM usage is equal to the variant only with static memory allocation.

c) Specialization of Interactions: After carefully examining the GPSLogger, we came to the conclusion that a interaction-level specialization that is restricted to the RTOS is not possible here. From the FreeRTOS API usage it is hard to tell why a specific API was used, since it hindered the expression of the developer’s intention (see also Section III).

However, we extend the scope of the specialization to the application. From the interaction model (Figure 2), we know that the LED thread does not interact with any other thread as it only periodically blinks the LED. Furthermore, toggling a GPIO pin takes far less cycles than the thread-management overhead. Therefore, we can safely inline the GPIO toggling into the timer ISR and remove the LED thread, including its stack and TCB. Compared to baseline, the system becomes 512 bytes of code and 1,616 bytes of RAM smaller. The startup time decreases by 9,397 cycles.

V. Conclusions and Future Work

In this paper, we described a taxonomy of specialization for real-time systems and define three levels of specialization that successively remove (unneeded) flexibility from the system. On the abstraction level, instance, and interaction level, we can remove abstractions, make instances fixed, and forbid concrete interactions. Furthermore, we discussed the benefits and challenges introduced by specialization. Although specialization yields significant improvements of nonfunctional properties, manual specialization has long outgrown engineers capabilities and is thus mostly applied on the coarse-grained abstraction level. Therefore, we argue that specialization on deeper levels requires automation to reach it's full potential.

To illustrate our taxonomy, we (manually) specialized an example application on the three specialization levels. Although the application was not designed with specialization in mind, we were able to extract the actually required interaction graph and, in consequence, were able to specialize the system to show improved nonfunctional properties. Therefore, we plan to integrate automated analysis and specialization into the build process and the compiler toolchain. If once automated, all levels of specialization can be generated and compared at compile time to choose the variant with the best nonfunctional properties for the specific use case.

References

[1] ERIKA Enterprise. http://erika.tuxfamily.org, visited 2014-09-29.

[2] Portable operating system interfaces (POSIX®) – part 1: System api – amendment 1: Realtime extension, 1998.

[3] AEEC. Avionics application software standard interface (ARINC specification 653-1), 2003.

[4] AUTOSAR. Specification of operating system (version 5.1.0). Technical report, Automotive Open System Architecture GbR, 2013.

[5] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd, 2010.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *15th ACM Symp. on Operating Systems Principles (SOSP '95)*. ACM Press, 1995.

[7] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. Building a global system view for optimization purposes. In *2nd Work. on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)*. IEEE Computer Society Press, 2006.

[8] Manfred Broy. Challenges in automotive software engineering. In *28th Intl. Conf. on Software Engineering (ICSE '06)*. ACM Press, 2006.

[9] Hans-Peter Deifel, Christian Dietrich, Merlin Göttlinger, Daniel Lohmann, Stefan Milius, and Lutz Schröder. Automatic verification of application-tailored OSEK kernels. In *17th Conf. on Formal Methods in Computer-Aided Design (FMCAD '17)*. ACM Press, 2017.

[10] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Cross-kernel control-flow-graph analysis for event-driven real-time systems. In *2015 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*. ACM Press, 2015.

[11] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. *ACM TECS*, 16(2), 2017.

[12] Christian Dietrich and Daniel Lohmann. OSEK-V: Application-specific RTOS instantiation in hardware. In *2017 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '17)*. ACM Press, 2017.

[13] Christian Dietrich, Peter Wägemann, Peter Ulbrich, and Daniel Lohmann. Syswcet: Whole-system response-time analysis for fixed-priority real-time systems. In *Real-Time and Embedded Technology and Applications (RTAS '17)*. IEEE Computer Society Press, 2017.

[14] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5), 1976.

[15] Wanja Hofer, Daniel Danner, Rainer Müller, Fabian Scheler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Sloth on Time: Efficient hardware-based scheduling for time-triggered RTOS. In *Real-Time Systems (RTSS '12)*. IEEE Computer Society Press, 2012.

[16] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *Real-Time Systems (RTSS '09)*. IEEE Computer Society Press, 2009.

[17] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *ISORC'14*. IEEE Computer Society Press, 2014.

[18] Tobias Klaus, Florian Franzmann, Tobias Engelhard, Fabian Scheler, and Wolfgang Schröder-Preikschat. Usable RTOS-APIs? In *10th Annual Work. on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '15)*, 2014.

[19] Butler W. Lampson. Hints for computer system design. In *9th ACM Symp. on Operating Systems Principles (SOSP '83)*. ACM Press, 1983.

[20] Jochen Liedtke. On μ-kernel construction. In *15th ACM Symp. on Operating Systems Principles (SOSP '95)*. ACM Press, 1995.

[21] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX Annual Technical Conf.* USENIX Association, 2009.

[22] Anthony J. Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[23] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2014-09-29.

[24] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29.

[25] David Lorge Parnas. On the design and development of program families. *IEEE Trans. on Software Engineering*, SE-2(1), 1976.

[26] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 2014.

[27] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1), 1988.

[28] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. Automatic feature selection in large-scale system-software product lines. In *13th Intl. Conf. on Generative Programming and Component Engineering (GPCE '14)*. ACM Press, 2014.

[29] Horst Schirmeier, Matthias Bahne, Jochen Streicher, and Olaf Spinczyk. Towards eCos autoconfiguration by static application analysis. In *1st Intl. Work. on Automated Configuration and Tailoring of Applications (ACoTA '10)*, CEUR Work. Proceedings. CEUR-WS.org, 2010.

[30] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Doreanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Automatic OS kernel TCB reduction by leveraging compile-time configurability. In *8th Work. on Hot Topics in System Dependability (HotDep '12)*. USENIX Association, 2012.

[31] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. In *30th Euromicro Conf. on Real-Time Systems 2018*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. to appear.