Tailorable System Software

Maßschneiderbare Systemsoftware

Der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

als

HABILITATIONSSCHRIFT

vorgelegt von

Dr.-Ing. Daniel Lohmann

Erlangen — 2013

Als Habilitation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Tag der Einreichung: 11. August 2013

Fachmentorat	Universitätsprofessor Dr.-Ing. habil. Wolfgang Schröder-Preikschat, Friedrich-Alexander-Universität Erlangen-Nürnberg
	Universitätsprofessor Dr.-Ing. habil. Klaus Meyer-Wegener, Friedrich-Alexander-Universität Erlangen-Nürnberg
	Universitätsprofessor Dr.-Ing. Olaf Spinczyk, Technische Universität Dortmund
Gutachter	Directeur de Recherche Dr. habil. Gilles Muller, INRIA/LIP6, Paris
	Universitätsprofessor Dr. rer. nat. Hermann Härtig, Technische Universität Dresden

Abstract

System software, such as the operating system, provides no business value of its own. Its *sole* purpose is to serve the concrete application's needs – that is, to map the functional and nonfunctional requirements efficiently to the functional and nonfunctional properties of the hardware. Efficiency calls for specific, tailored system software; reusability demands generic solutions. To overcome this dilemma, most system software provides built-in static variability: It can be tailored at compile time with respect to a specific application–hardware use case.

In the case of Linux v3.2, this static variability is reflected by nearly $12{,}000$ configurable features that control the inclusion and exclusion of $28{,}000$ source files with $84{,}000$ conditional (`#ifdef`) blocks. Variability by means of thousands of features imposes challenges for both system-software developers, who have to implement and maintain variability, as well as application developers/administrators, who have to understand the impact of all these features in order to configure a tailored variant.

Over the last four years, my research has focused on methods and techniques to improve the design, implementation, and maintenance of static variability in highly tailorable system software. My central contributions in this respect are: (a) The CiAO approach, which employs language techniques to achieve excellent *up-tailorability* of embedded system software (towards the requirements of a specific application). (b) The SLOTH approach, which employs generative techniques to achieve *down-tailorability* of embedded system software (towards better exploitation of modern commodity hardware). (c) The VAMOS approach, which employs cross-language analysis techniques and holistic variability modeling to improve on the long-term maintainability of multi-paradigmatic variability implementations in existing large-scale system software, such as Linux.

This research has been carried out in collaboration with seven doctoral researchers and master students from my research group, four of which have already defended.

Contents

1. Introduction

System software is computer software that is designed to operate and control a computing hardware and to provide a platform for the execution (and partly also creation) of application software on this hardware. Examples include, first and foremost, the operating system, but also the compiler, the database engine, middleware, network stack, Java virtual machine, and so on. System software "maps" (in a general sense) the high-level application functionality to the imperative hardware machine.

System software provides no business value of its own. Its *sole* purpose is to ease the development, integration, and operation of applications – that is, to provide the "right" set of abstractions for a particular application use case: The functional and nonfunctional requirements of the application have to be mapped *efficiently* to the functional and nonfunctional properties of the hardware. The term *efficiency* here refers to further nonfunctional properties of the resulting system, such as memory footprint, throughput, event latency, robustness, jitter, and so on. The "ideal" system software does not impair these properties by abstractions and policies that do not serve the application's needs [141]. Between the application and the hardware, the effects of system software should be as "thin" as possible (Figure 1.1). The avoidance of unnecessary overheads is particularly important for the cost-sensitive domain of embedded systems.

Efficiency calls for specific, tailored system software for each concrete application–hardware use case. On the other hand, the broad variety of applications and hardware platforms demands generic, reusable solutions. Thus, system software designers are caught "between a rock and a hard place": broad reusability versus case-specific efficiency.

1.1. The Case for Tailorable System Software

To overcome this dilemma, most system software provides built-in **static variability**: It supports a broad range of application requirements and hardware platforms, but can be tailored at compile-time with respect to a specific use case. Historically, this has led to the notion of system software as *program families* [145, 144].

With the rising number of static variation points and their dependencies (in the source code of Linux v3.2, we can find more than $112{,}000$ variation points) this has been complemented by explicit models of variability known from the domain of software product lines (SPLs) [138, 100, 131, 45]: The intended variability is expressed as a **feature model** – a set of (configurable) **features**, which reflect a mandatory, optional, or alternative functionality offered by the implementation. Features typically carry **constraints** that define how they depend on or conflict with other features. In the case of Linux v3.2, the intended variability is described as nearly twelve thoussand ($11{,}863$)

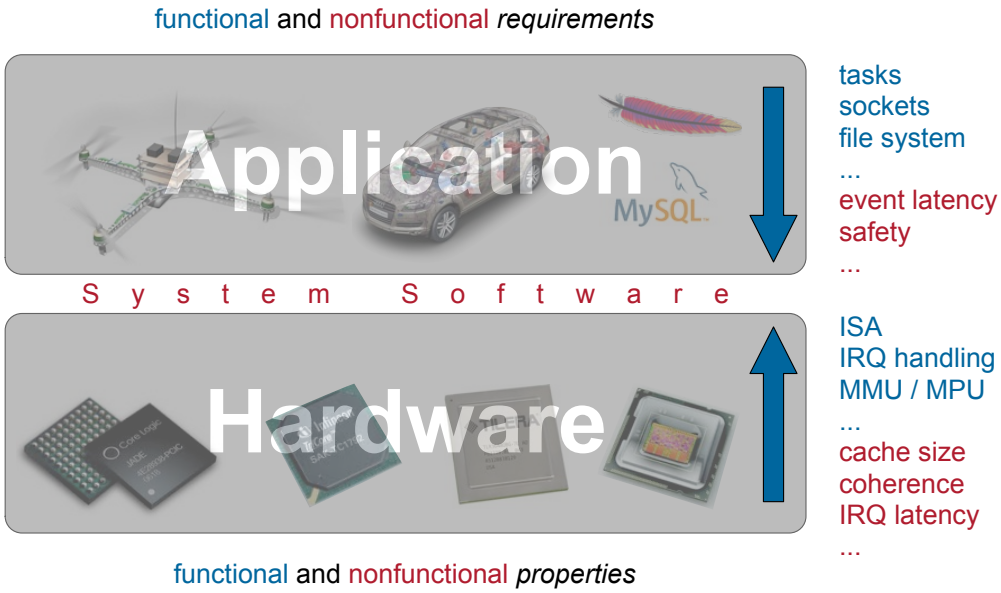


Figure 1.1.: System software: between a rock and a hard place
The "ideal" system software is (a) resuable and provides (b) an as-thin-as-possible layer that exactly maps the functional and nonfunctional requirements of the application – and nothing more – to the functional and nonfunctional properties offered by the hardware.

features and their constraints in a language called KCONFIG [9, 28, 27]; KCONFIG front ends load this model to let users select and deselect features to come up with a valid **configuration** that describes a concrete Linux **variant**.

Linux is just one example of system software that offers tailorability by configuration – although the largest one I am aware of. Table 1.1 lists a small selection of open-source/research system software, with a focus on the operating-system domain: The eCos [2] real-time operating system (RTOS), for instance, also provides more than 5,400 features, which are configured with (and checked by) ECOSCONFIG [92, 21]. All other projects listed in Table 1.1 have imported the KCONFIG infrastructure of Linux. It is remarkable, though, that even very small and young projects (compared to Linux), such as BUSYBOX [109] or COREBOOT [30], and research projects, such as CiAO [C20⋆, C12⋆],[1] CiAO/IP [C5⋆] or L4/FIASCO [118] already offer hundreds to thousands of features.

Variability by means of thousands of configurable features imposes big challenges for both system-software developers, who have to implement and maintain variability, as well as application developers/administrators, who have to understand the impact of all these features in order to configure a tailored variant.

[1] References to own publications, such as [J7, C25, W24], are prefixed by a one-letter acronym indicating the type of the publication, including: Journal, Conference, Workshop (peer-reviewed). The complete list can be found in the Personal Bibliography on pp. 51ff. References to key papers of this habilitation treatise are additionally marked by an asterisk, such as [C20⋆].

Project	Description	Version	# Features
Linux	Operating system	v3.2	11821
eCos	Library operating system for embedded systems	3.x (hg 3216)	5466
BUSYBOX	UNIX tool suite for embedded systems	1.20.1	879
COREBOOT	Firmware (BIOS/UEFI)	4.0 (13.10.2012)	1140
L4/FIASCO	μ-kernel operating system	svn 38	1255
CiAO	Library operating system (AUTOSAR OS)	svn 1829	578
CiAO/IP	TCP/IP stack for IPv4	svn 1829	96

Table 1.1.: Examples for statically configurable system software

1.2. Contributions

Over the last four years, my research has focused on methods and techniques to improve the design, implementation, and maintenance of variability in highly tailorable system software. In this respect, I have been conducting research in the areas of (Figure 1.2):

1. Tailorable System Software (Goal)
2. Software Product Lines (Method)
3. Languages and Generators (Technique)

Tailorable system software is the target domain (Goal). For this purpose, system software is conceptually understood as a software product line; the functional and nonfunctional properties are modelled as optional and mandatory features and constraints (Method). The resulting variability eventually has to be made explicit; it has to be implemented (and maintained) in the code in a way that facilitates modularity and separation of concerns without sacrificing nonfunctional properties, such as run-time and memory efficiency or predictability (Technique).
In the end, however, it is all about the goal: researched methods and techniques have to show their benefits by application to real system software; for instance, by evaluating them in the design, implementation, and maintenance of operating systems and network stacks. The central contributions of my research in this respect are:

(a) The CiAO approach [C20⋆, C12⋆, C5⋆], which employs language techniques to achieve excellent *up-tailorability* of embedded system software (towards the requirements of a specific application).

CiAO is Aspect Oriented

CiAO is a *constructive* analysis, design and implementation approach for the development of fine-grained configurable system software using aspect-oriented programming (AOP). It has been evaluated in the development of event-triggered automotive RTOS, compilers, and network stacks for resource-constrained embedded systems.

(b) The SLOTH approach [C19⋆, C10⋆, C7⋆], which employs generative techniques to achieve *down-tailorability* of embedded system software (towards better exploitation of modern commodity hardware).

The name features both the lazy animal breed and the deadly sin of laziness.

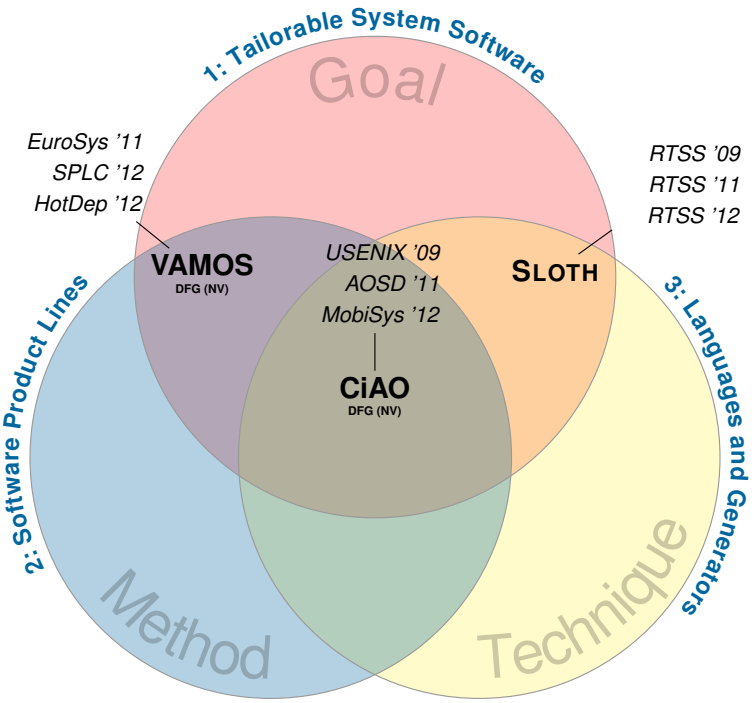


Figure 1.2.: Research areas with central research projects and key papers

SLOTH is a *constructive* approach for the development of very thin system software that is aggressively and automatically tailored by generators towards platform-specific hardware particularities. It has been evaluated in the development of event-triggered and time-triggered automotive RTOS.

Variability Management in Operating Systems

(c) The VAMOS approach [C14⋆, C6⋆, W5⋆], which employs a holistic variability model to improve on the long-term maintainability of multi-paradigmatic variability implementations in existing large-scale system software, such as Linux.

VAMOS is an *analytical* approach based on the automatic cross-language extraction and checking of variability information. It has been evaluated by providing tool support to find variability defects and bugs in Linux, increase the coverage of static checkers in Linux, BUSYBOX, and L4/FIASCO, and the automatic tailoring of Linux for a particular application use case.

This research has been carried out in collaboration with seven doctoral researchers and master students from my research group. I have been the scientific leader and conceptual contributor.

1.3. Papers of This Treatise

This document is a cumulative habilitation treatise. Out of my 82 peer-reviewed publications (see Appendix A.2 on pp. 51ff), I have selected the following 9 papers (full texts are provided in Appendix B on pp. 59ff) as the key contributions of my research:

CiAO

USENIX '09 pp 61 ff	Lohmann, Hofer, Schröder-Preikschat, Streicher, and Spinczyk. "CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems" (Acceptance rate: 16%)	[C20⋆]
AOSD '11 pp 75 ff	Lohmann, Hofer, Schröder-Preikschat, and Spinczyk. "Aspect-Aware Operating-System Development" (Acceptance rate: 23%)	[C12⋆]
MobiSys '12 pp 87 ff	Borchert, Lohmann, and Spinczyk. "CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack" (Acceptance rate: 18%)	[C5⋆]

SLOTH

RTSS '09 pp 101 ff	Hofer, Lohmann, Scheler, and Schröder-Preikschat. "Sloth: Threads as Interrupts" (Acceptance rate: 21%)	[C19⋆]
RTSS '11 pp 111 ff	Hofer, Lohmann, and Schröder-Preikschat. "Sleepy Sloth: Threads as Interrupts as Threads" (Acceptance rate: 21%)	[C10⋆]
RTSS '12 pp 123 ff	Hofer, Danner, Müller, Scheler, Schröder-Preikschat, and Lohmann. "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS" (Acceptance rate: 22%)	[C7⋆]

VAMOS

EuroSys '11 pp 135 ff	Tartler, Lohmann, Sincero, and Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem" (Acceptance rate: 15%)	[C14⋆]
SPLC '12 pp 149 ff	Dietrich, Tartler, Schröder-Preikschat, and Lohmann. "A Robust Approach for Variability Extraction from the Linux Build System" (Acceptance rate: 33%)	[C6⋆]
HotDep '12 pp 159 ff	Tartler, Kurmus, Heinloth, Rothberg, Ruprecht, Doreanu, Kapitza, Schröder-Preikschat, and Lohmann. "Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability" (Acceptance rate: 42%)	[W5⋆]

Regarding the order of authors: The first author did most of the actual implementation and experimental work (usually a doctoral researcher), in close collaboration with me as the responsible post-doc and scientific project lead. As such, I am listed as second author on most papers, which can be considered as the "classical post-doc position". Since 2012, however, I have been deliberately shifting my name on papers to the last position for all research projects for which I am the scientific project lead.

1.4. Structure of This Treatise

The remainder of this document is structured as follows: In the next chapter

I provide a short introduction into and classification of implementation techniques for tailorable system software. This is followed by chapters for the three research projects in which I have investigated such techniques:

In these chapters, I provide a brief overview of the respective projects, their underlying design and implementation techniques, and a classification of the role of each paper that is part of this habilitation treatise. Each chapter is *logically* followed by the reprints of the respective papers, which constitute the body and main contributions of this treatise. For the sake of easy (partial) printing and reading, however, I have moved the reprints to Appendix B and continue with:

In the last chapter, I discuss some general aspects of my work, derive ideas for further research, and conclude my work. I close with:

In the appendix I provide the general and personal bibliographies (including a complete list of own papers) and reprints of the papers selected for this cumulative habilitation treatise.

2. Towards Tailorable System Software

It has long been known that tailorability has to be considered as a first-class design goal from the very beginning [142] and it is no surprise that most early work on program families [148, 147, 145, 146, 144] focuses on system software. Technically, a system software has to provide static variation points to be tailorable at compile time – well-defined points in the program structure that, depending on a compile-time decision, influence the resulting binary code.

2.1. Design for Static Variability

Variability as a system property includes two separated – but related – aspects: *implementation* and *configuration*.

Implementation (Model Level). System software developers provide variation points in the code – in most cases by means of conditional compilation and the C Preprocessor (`CPP`) [136, 88, 37, 23] (fine-grained variability) or the build system [40, 13] (coarse-grained variability). Other implementation approaches for static variability are (typically language-supported) component models [106, 107, 89, 137, 134, 123, 84, 128, 133, 110, 90] or domain-specific generators [100, 104, 18, 80].

Besides the actual variation points, system software developers furthermore have to provide an interface to users or system integrators to exploit the thereby offered variability.

Configuration (Instance Level). Application developers or system integrators configure the system software to derive a concrete variant that fits their purposes. The provided interface for this process varies from (1) the completely manual (un-)commenting of `#define` directives (unvalidated) or component aggregation (validated by the rules of some type system) over (2) explicit variability models supported by feature models and validating configuration tools [22, 73, 50] up to (3) the (semi-)automatic configuration based on application analysis [44] or a specification written in some domain-specific language (DSL) [76, 104, 80, 100].

The configuration interface defines the **configuration space**, which is usually intended to cover all variability of the **implementation space** defined by the variation points of the actual implementation, but at a higher level of abstraction: Instead of single (i.e., *extensional*) variation points, system users can deal with configuration options – more

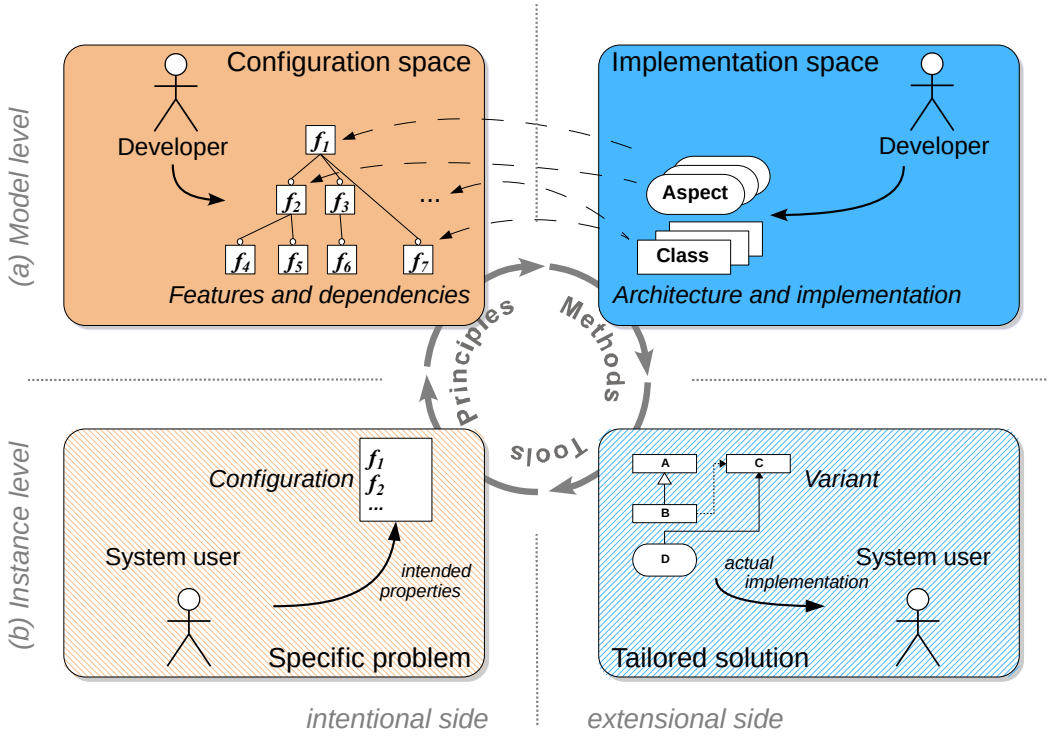


Figure 2.1.: The model of configurable system software

(a) *System developers* provide variation points in the implementation (*implementation space*) together with an (abstract) interface to access the thereby offered variability in a controlled manner (*configuration space*). **(b)** *System users* specify for their *specific problem* a concrete instance via this interface to derive a *tailored solution*.

or less abstract (i.e., *intentional*) features;[1] they select from these features (explicitly or implicitly) to get a **tailored solution** for their **specific problem**.

Figure 2.1 presents the overall picture of this process and model, which is to be supported by principles, methods, and tools. This is – with a focus on the implementation

[1] It should be noted that the understanding of features as (implementation-specific) *configuration options* is in stark contrast to the early SPL literature, which understands the concept of features more from the viewpoint of requirements, that is, focuses on *problem variability*: Features and feature models (as a means to structure variability) are understood as intentional constructs that describe commonalities and differences within a specific problem domain (the *problem space*), obtained by a process called domain analysis [139, 138, 131, 97]. However, especially in the open-source world, system software is developed implementation-driven [9]: Configurable features rarely stem from a domain analysis process, but generally describe technical (i.e., explicit) implementation options and their constraints. They are often implemented *before* they turn into features. Feature models have more and more been employed as a *tool* to specify the configuration space and its constraints: In the systems-software community, this led to the high number of features exemplified in Table 1.1. In the SPL community, this in turn has led to the (arguable) recognition of system software as (large) "classical" SPLs [45, 22, 8] – the differences between the *software variability* (implementation/configuration space) and *product-line variability* (problem space) have somewhat blurred [43].

space – the goal of my research around CiAO, SLOTH, and VAMOS.

The general challenge is the *efficient*, *consistent*, and *maintainable* mapping from elements of the configuration space (such as features and their constraints) to the entities of the implementation space (such as conditional blocks and their presence conditions[2]): The *granularity* of a feature in the implementation may scale from coarse-grained (inclusion or exclusion of a complete module or file) to fine-grained (inclusion or exclusion of a single statement – and even below [17]), or even be crosscutting (require many variation points [C25, 94]). For static tailoring, feature implementations should be bound at compile time and thus not induce an extra run-time overhead just by the fact that they are configurable. Features often interact with each other [32], so that many derivatives have to be provided for their implementation.

All this is to a high degree a matter of the chosen implementation approach, which has a large impact on achieving *granularity*, *efficiency*, *consistency*, and *maintainability*.

2.2. Implementation Approaches for Tailorable System Software

In general, the approaches to implement static variability in system software can be classified into three different classes (Figure 2.2):

Decompositional Approaches. The implementation of optional and alternative features is intermingled in the source code with each other and the base system. At compile time, *annotation-based filtering techniques*, such as conditional compilation with `#ifdef` statements and the CPP, are used to filter out the source lines that do *not* implement the concrete variant. This is, in a broader sense and with focus on consistency and maintainability, the scope of the VAMOS approach.

Compositional Approaches. Optional and alternative features are implemented as a set of fine-grained and loosely coupled implementation components in a "syntactically rich" implementation language (e.g., C++ classes or templates). Each component (ideally) implements a single feature only. A concrete variant is derived by *composition* of feature components; the validity of the composition is guarded by the type system of the implementation language. This is, with focus on granularity, efficiency, and maintainability, the scope of the CiAO approach.

Generative Approaches. The code of optional and alternative features is not provided in the compiler's target language (such as C), but generated out of templates by a problem-specific generator. A concrete variant is derived as an instance of the *meta model* implemented by the generator. The meta model basically provides a problem-specific type system for feature composition; the instances are often described in a DSL. This is, with focus on granularity and efficiency, the implementation technique applied in the SLOTH approach.

[2] The *presence condition* is the condition that has to be fullfiled for the respective block to be included, such as the existence of a certain macro

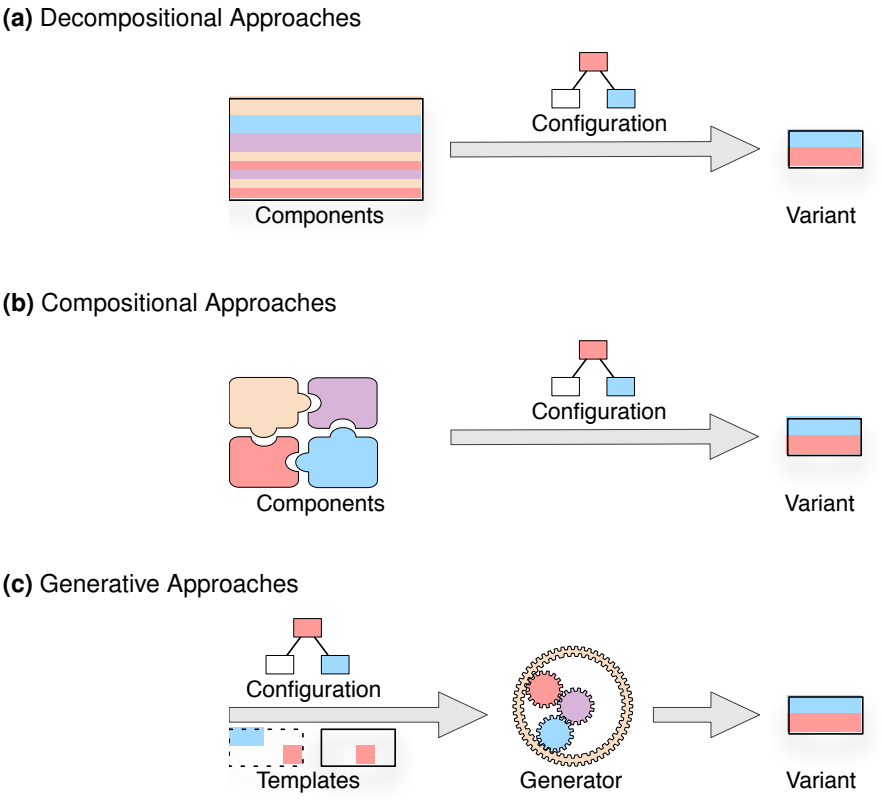


Figure 2.2.: Classification of implementation approaches used in configurable system software
(a) All variation points are contained in a single implementation and described by *annotations* (e.g., `#ifdef` blocks) before compilation to derive a concrete variant. **(b)** Variation points are provided as *(feature) components* that can be composed according to the *type system* to derive a concrete variant. **(c)** Variation points are provided by an active *generator*, which employs the given configuration together with domain-specific knowledge to derive a concrete variant.

For pragmatic reasons, real-world system software typically employs approaches of multiple classes, even though in various degrees and qualities: For instance, variation points are not only implemented with the CPP using conditional compilation (decompositional), but also with CPP macro expansion (compositional). Some variation points are bound at link time by symbol-level linking (compositional), and so on. Nevertheless, most system software employs one of them as the dominant implementation technique. In the following, I briefly discuss issues and variations of each approach and its application to tailorable system software.

2.3. Decompositional Implementation of Variability

The application of preprocessors, especially the CPP, is considered to be *the* dominant implementation approach for static variability in system software: Most system software written in C [143] or C++ [126] also employs the CPP [88, 37, 23] for this purpose.

Basically, a preprocessor filters the source code before it is passed to the compiler by evaluating preprocessor statements for conditional compilation[3] (`#ifdef CONFIG_OPTION` ... `#else`... `#endif` blocks); the presence condition of such a block (such as the feature flag `CONFIG_OPTION`) *annotates* which feature the respective code block belongs to.

The advantages of annotation-based static variability are *efficiency* and *granularity*: The feature-specific part of the code is directly injected into the source code before compilation without any indirection via a (language-specific) run-time concept (such as virtual method invocation). Hence, the approach is inherently free of any run-time overhead for binding feature-related parts of the code. Because preprocessing is just text processing on the source code (outside of the "burden" of a type system), feature-dependent parts can be specified on statement level or even below [17], which technically facilitates very fine-grained feature interactions.

The disadvantages of this flexibility are *consistency* and *maintainability*: Neither the presence conditions nor the results of including or excluding feature-related parts of the code are checked for declarative completeness, consistency, and type conformance; if there are many and fine-grained annotations ("`#ifdef` hell"), the code becomes difficult to understand and maintain [136, 122, C25]. Approaches to mitigate these disadvantages by better tool support are an active field of research [103, 34, 29, 46, W15, 24, 16, 10] – and also part of the VAMOS project [C2, C17, C14⋆, W11].

Also considered by VAMOS is the fact that in most real-world system software (e.g., Linux), only *fine-grained* decompositional variability is implemented with the CPP (and sometimes other source-code preprocessors, such as [1, 85, 62]), whereas the implementation of *coarse-grained* variability is delegated to the build system [13, 39, W3, C6⋆], which conditionally includes or excludes complete translation units from being preprocessed, compiled, and linked.

2.4. Compositional Implementation of Variability

A wide-spread direct counterpart to the source-code–based decompositional implementation of variability by CPP and conditional compilation is *source-code patching*: Optional or alternative features are provided as source-code patches, which are applied by the PATCH tool to augment the source-code before compilation. As preprocessing, source-code patching is basically untyped text processing on the source code, so it shares the same advantages regarding *efficiency* and *granularity* and disadvantages regarding *consistency*

[3] Most preprocessors, including the CPP, also provide *macro expansion* as a second mechanism, which belongs, depending on the expressive power of the expansion mechanism, to the class of compositional or generative approaches.

and *maintainability*; better, "semantic" patch languages that can incorporate context into the patching process [55, 54, 36, 19, 63, 60] aim to mitigate the disadvantages. In practice, patches are mostly used as a transitional or organisational means for compositional variability for features that are developed and maintained "out-of-tree" [9].

Most other approaches for compositional variability aim at type-safe compositions: Variation points are implemented on the base of syntactical constructs of the implementation language (such as classes, methods, fields); the underlying type system guards the composition of valid variants. A large research community investigates methods and languages to support compositional variability [114, 99, 72, 61, 25], only few of which, however, have actually been applied to larger pieces of system software.

The use of compositional variability in system software is mostly based on the abstractions provided by some general-purpose programming language, such as Oberon modules [137], Modula 3 generics (SPIN [133, 128]) C++ objects (Choices [134], K42 [57, 52]), C++ classes (PURE [110]), C++ templates (EPOS [112, 96]), Java classes (JX [90]), or – as also investigated In the CiAO project – aspects (BOSSA in Linux [80], AspectC in FreeBSD [94, 82]; TOSKANA in NetBSD [51]; AspectC++ in PURE [91, W34], COMET [78, 77, 59], IP-TN [56], eCos [C25, 4], CiAO [C20⋆, C12⋆, J1], and CiAO/IP [C5⋆]). Other approaches are not tied to a particular programming language, but instead employ a language-independent (binary) component model, such as plain function libraries (exoKernel [129]) or Microsoft's COM [116], with components as the elementary building blocks. Examples for the COM approach include THINK [89], KOALA [106], and Flux/OSKit [123]. On the border to generative approaches and DSLs are component models that are supported by domain-specific general-purpose programming languages, such as NesC for TinyOS [64, 84, 42, 66].

Compositional variability based on strongly typed languages or component models bears clear advantages regarding *consistency* and *maintainability*: Consistency and validity of concrete variants is to a large degree ensured by the type system; a well-defined module and interface concept provides for easy extensibility (optional features), substitutability (alternative features), and separation of concerns: in the ideal case, each feature of the configuration space can be mapped to a distinct module or component in the implementation space (Figure 2.1).

The *efficiency* of binding features, on the other hand, depends on the actually employed mechanisms and how well they match the needed binding abstraction. With C++ objects or COM components, for instance, feature substitutability (abstraction) is implemented by virtual functions (mechanism), which bears an inherent run-time and memory overhead [127]. Choices and K42 also aim at run-time configurability, so they use a mechanism to bind feature modules that matches the intended abstraction. If, however, the goal is static variability, the mechanism provides more than actually needed; the overhead of "componentization" ([107]) can become significant [J8]. Flux/OSKit partly mitigated this overhead by additional language and tool support [107] that (re-)facilitates cross-component optimizations, such as inlining. The general message, however, is that *efficiency* is hampered if the employed binding mechanism provides more than the

actually needed abstraction.[4] The implementation of static variability calls for purely static binding mechanisms.

The possible *granularity* of feature interactions in the implementation is restricted to the granularity of the syntactic entities offered by the language or component model – in most cases to function/method level. This is in stark contrast to decompositional variability using the CPP, where features may augment the implementation on statement level (or even below). The implementation of finer-grained variability with compositional approaches is possible, but often leads to either code duplication (e.g., a method is duplicated to provide a variant that differs only in a single statement) or artificial "micromodularization" [3] (e.g., the single statement is factored out into an own method that does not necessarily reflect an abstraction) – both of which imply follow-up costs with respect to *consistency*, *maintainability*, and *efficiency*. Ideally, the granularity of the syntactical entities of the employed language or component model matches the granularity of feature interactions in the implementation.

In system software – especially with respect to optimizing nonfunctional properties –, these interactions often are (a) fine-grained and (b) bear an *inherently* crosscutting character [C20⋆, W29, W26, 78]. This makes, despite all stated critique regarding negative effects on modular reasoning [81, 74, 70, 58, 41, 34], the syntactical concepts provided by AOP [125, 95], especially *pointcut* and *advice*, so attractive as a means for the efficient compositional implementation of static variability in this domain. This is the scope of the CiAO approach.

2.5. Generative Implementation of Variability

With generative implementation techniques, the elements of the implementation space (Figure 2.1) become *active*: Instead of mapping features and their constraints to implementation fragments in the employed programming language or component model (such as `#ifdef` blocks or COM components), they are mapped to generators – meta programs (developed in a Turing-complete language) that interpret the configuration to *generate* (parts of) the tailored system's implementation at build time [100].

Generative approaches are beneficial if feature interactions are (a) very fine-grained and complex (context-sensitive) in the implementation, or (b) have to be instantiated over many variation points. In system software, this is often the case when dealing with hardware peculiarities. A typical example is the implementation of IPC or syscall mechanisms [121]; the respective stubs that marshal parameters over protection boundaries have to be instantiated for each specific IPC or syscall. In order to provide for *efficiency*, the code generation has furthermore to be highly optimized with respect to platform

[4] An example for the efficiency issues caused by "having too much" in a fundamental binding mechanism is local IPC in μ-kernel operating systems: Mach's [140] IPC mechanism, for instance, was optimized for remote IPCs by network transparency and portability – and performed badly if only local IPC was needed [124]. In L3 and L4 [135, 130], in contrast, the IPC mechanism was less flexible and optimized for local situations by a platform-dependent implementation in assembler, which results in a much better perfomance for local IPC [124].

particularities [102]. In a sense, the respective generators implement operating-system–specific *active libraries* [120, 101, 100] that know how to map certain mechanisms of the system software (such as a syscall) efficiently to hardware. This is also the approach behind SLOTH.

Other examples from the domain of system software include the implementation of capabilities [20], cache coherence protocols [111], and device drivers [115, 108, 104, 75, 33]. Not as close to hardware, but also suggested have been generative approaches for protocol stacks [86, 119, 18] and thread schedulers [68, 87, 80]. These examples for generative implementation of static variability are mostly motivated by benefits regarding *consistency* and *maintainability* – which are achieved by the use of a domain-specific language (DSL) as an input language. Compared to general-purpose languages, DSLs typically provide fewer and more declarative mechanisms, so that software development and maintenance can take a higher, domain-related abstraction level and thus becomes easier and less error prone [100, 68, 67]. In other cases, the DSL is embedded into a general purpose language and extends it by domain-specific abstractions [115]. Embedded DSLs and generators are often implemented with C++ template meta-programming [132, 100, 93] (also in combination with AOP [C29, C27, 4]), a technique that also has been applied to system software [20, 86, 112, 96].

Generative approaches bear the risk that the complexity of variability implementation is just shifted from the target software to the generator and languages, so that the difficulties regarding *consistency* and *maintainability* remain – just on another level. To mitigate these problems, reusable generator frameworks and modularization concepts have also been suggested for language and generator design [79, 65, 31].

For embedded system software, such as the RTOS, even the complete specification in higher-level DSLs and subsequent generation has been suggested [105]. SLOTH, which implements the automotive OSEK/AUTOSAR operating-system standards [48, 49, 69, 98], also features complete generation. The expressive power of the configuration language, however, is very similar to other OSEK/AUTOSAR implementations [76]. The goal behind the generative implementation of variability in SLOTH is to provide efficiency and portability by whole-system optimization and platform-specific back ends for the efficient mapping of kernel mechanisms to hardware.

2.6. Summary

Variability as a system property includes two separated – but related – aspects: *implementation* and *configuration*. The general challenge is the *efficient*, *fine-grained*, *consistent*, and *maintainable* mapping from elements of the configuration space (such as features and their constraints) to entities of the implementation space.

Technically, a piece of system software has to provide (many/good) static variation points to be tailorable at compile time. This is, first and foremost, a question of the software design *method* – static variability has to be considered as a first-class design goal from the very beginning. Additionally, the chosen implementation *technique* has a large impact on achieving granularity, efficiency, consistency, and maintainability in

the resulting system. Common in system software are *decompositional*, *compositional*, and *generative* implementation approaches, all of which have specific advantages and disadvantages.

To exploit the advantages and mitigate the disadvantages by methods, techniques, and tools is the objective behind CiAO (compositional), SLOTH (generative), and VAMOS (decompositional, multi-paradigmatic) – with a strong focus on the *goal*: System software that offers fine-grained static tailorability with respect to its functional and, especially, nonfunctional properties.

3. The CiAO Approach

The dominant implementation technique for variation points in system software is the CPP [88, 37, 23], despite all the disadvantages with respect to understandability and maintainability ("`#ifdef` hell") this approach is known for [136, 122]. In my PhD thesis [T1], I evaluated aspect-oriented programming [125, J7] as an alternative technique [C25] and came up with the analysis and design method of *aspect-aware system-software development*. This method leads to a much better separation of concerns and it facilitates fine-grained feature implementations and the configurability of even fundamental system policies without giving up on run-time efficiency and memory thriftiness.

The result of this research – and also the starting point for my further research activities – was the CiAO operating system [C20⋆, C12⋆, J1]. CiAO implements the automotive OSEK/AUTOSAR standards for completely statically configured[1] RTOSs [48, 49, 69], but provides a much better configurability and tailorability towards specific application requirements than leading commercial implementations. Besides operating-system kernels, we have also applied aspect-aware system-software development to the domains of compilers [C18, J4] and network stacks for resource-constrained embedded systems [C5⋆].

3.1. CiAO Goals

The primary goal of the CiAO approach is configurability of all functional and nonfunctional properties [B1, W29]. This includes fine-grained tailorability and composability of optional features [J8], but also configurability of even fundamental, "architectural" policies, like synchronization [W25] or isolation in time and space [W24, W17] in an OS kernel or byte ordering and checksumming in an IP stack [C5⋆].

3.2. Implementation Approach: Aspect-Aware Development

Static variability in CiAO is implemented using a *compositional* approach based on AOP [125, 95] and the strongly typed AspectC++ language [J7]. The basic implementation idea is to achieve a strict decoupling of policies and mechanisms in the implementation by using aspects as *the* primary composition technique: CiAO components are sparse and do not directly interact with each other; they are glued together and extended by aspects, which thereby implement policies and optional features. For fundamental "architectural"

[1] *Completely statically* here means that all OSEK system entities (tasks, events, resources, ...), their priorities, and possible interactions (which tasks can access which resource) are known at compile time.

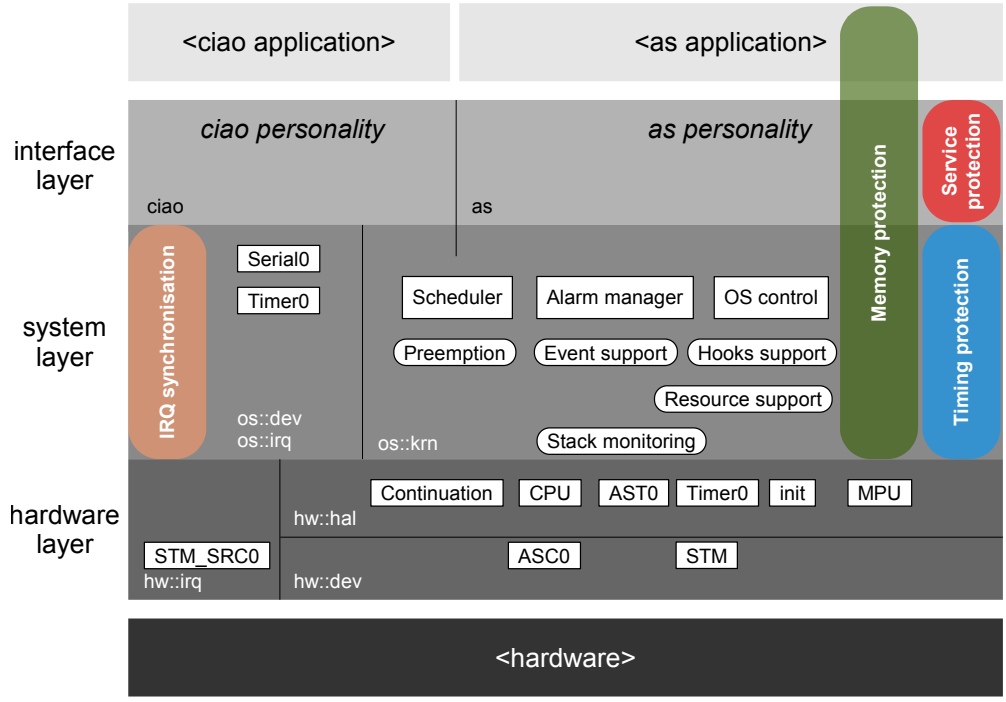


Figure 3.1.: CiAO software structure

Depicted are the three functional software layers of CiAO with a selection of (logical) sublayers, abstractions, and aspects (depicted with rounded corners). Configurable architectural properties, such as *Memory protection*, *Service protection*, or *IRQ synchronization* are modeled as aspects and may have an effect across multiple layers.

system policies, such as synchronization or protection, this binding may even take place across multiple functional layers [144] up to the application (Figure 3.1).

The fundamental language concept of AOP in this context is *advice*, by which it becomes possible to let optional features and cross-cutting policies *specify themselves* how they are to be integrated into the base program (Figure 3.2). As advice-based binding is inherently loose (if the addressed join point is not present, the binding is silently dropped) it also provides a solution for the implementation of interacting optional features [C20⋆], which are difficult to tackle with other compositional approaches [32]. Advice code is inlined in AspectC++ [J7], so separation of concerns and loose coupling in the *design* do not imply an binding overhead in the *implementation* of the resulting system [144])– when refactoring `#ifdef`-based variability into aspects, the resulting binary code typically remains the same [C25].

Aspect-aware development thereby leads to a very fine-grained composability, especially with respect to policies and optional features – no functionality shall be considered mandatory [142]. The identification and decomposition of optional features and cross-cutting policies is guided by the structured concern impact analysis (CIA) process [W23, C12⋆] and a set of predefined class and aspect roles [C20⋆, C5⋆].

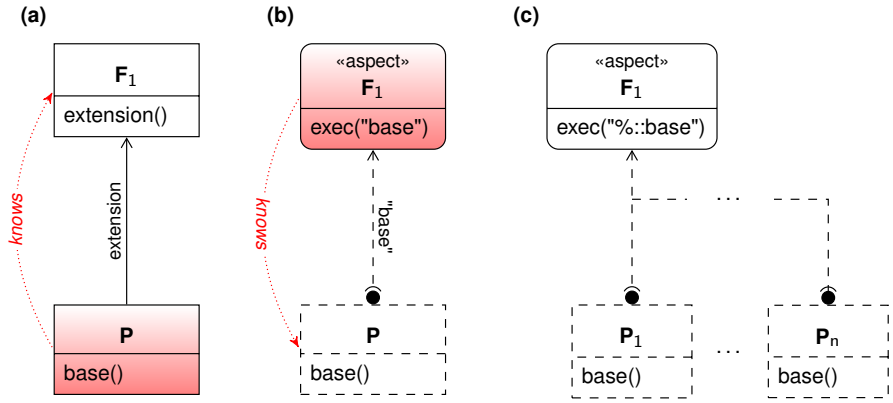


Figure 3.2.: Loose coupling of (interacting) optional features by advice-based binding
Optional feature F_1 shall extend product-line component P, so `F1::extension()` needs to be invoked from `P::base()`. **(a)** With traditional programming paradigms (e.g., OOP), P knows and *has to know* F_1 (i.e., its name and signature) to invoke `F1::extension()`. **(b)** With advice, F_1 *integrates itself* into P (i.e., in the execution of `P::base()`) if P is present. **(c)** Advice-based binding with quantification: F_1 integrates itself into *all* present (matching) feature components $P_{1...n}$.

3.3. CiAO Results

We have applied aspect-aware development to the domains of embedded operating-system kernels (CiAO [C20⋆, C12⋆]), compilers (PUMA [C18, J4]), and network stacks for embedded systems (CiAO/IP [C5⋆]). In all cases, the resulting tailorability leads to significant advantages with respect to nonfunctional properties compared to existing leading solutions:

In comparison to a leading commercial implementation[2] of the OSEK OS standard [69], CiAO achieves significantly lower latencies for system services (up to 70% less) and an excellent scalability of the memory footprint, mainly because of its better tailorability with respect to the actual application's requirements. However, even if the amount of features of a CiAO variant is "artificially enriched" to provide the same amount of unneeded functionality than the commercial OSEK, CiAO still performs better in most cases [C20⋆].

For PUMA, which is an open-source C/C++ transformation framework also used in the implementation of the AspectC++ weaver [J7], the goal was not so much performance and memory efficiency, but separation of concerns with respect to optional features, namely, tailoring towards the large variety of C/C++ language dialects and extensions (GNU, VisualC++, AspectC++, ...). The PUMA C++ parser implementation, for instance, requires only ten percent of the lines of code of the GNU C++ parser [J4, C18].

CiAO/IP, the most recent project, provides an aspect-oriented IPv4 TCP/IP stack for sensor nodes and other resource-constrained embedded systems. The leading implemen-

[2] part of the BMW and VW/Audi standard cores

tations in this field were *micro-IP (uIP)* and *lightweight-IP (lwIP)* [83]. By its excellent tailorability towards specific application requirements, the CiAO/IP TCP/IP stack outperforms lwIP and uIP in terms of code size (up to 84% / 88% less than uIP / lwIP for a UDP sender on an AVR), throughput (up to 587% / 33% higher than uIP / lwIP for a TCP sender on IA-32 with Gbit Ethernet) and energy consumption (up to 63% lower than uIP on AVR for a TCP sender) [C5⋆].

3.4. CiAO Key Papers

In the following, I briefly classify the role of the three related key papers, which are part of this cumulative habilitation treatise. Reprints of these papers are available in Appendix B.

USENIX '09 pp 61 ff — Lohmann, Hofer, Schröder-Preikschat, Streicher, and Spinczyk. "CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems" (Acceptance rate: 16%) [C20⋆]

The USENIX '09 paper introduces, on the example of the CiAO operating system, AOP and aspect-aware development as an alternative to CPP-based configuration *from a systems perspective*: It shows why "`#ifdef`-hell" appears to be inevitable in configurable system software, introduces the three fundamental design principles of aspect-aware develpment and the resulting aspect roles to overcome these problems, and evaluates the scalability of CiAO with respect to performance and memory consumption.

AOSD '11 pp 75 ff — Lohmann, Hofer, Schröder-Preikschat, and Spinczyk. "Aspect-Aware Operating-System Development" (Acceptance rate: 23%) [C12⋆]

The AOSD '11 paper complements the USENIX '09 paper by summarizing aspect-aware development from a *software engineering perspective*: It describes our longer-term experiences with using AOP vs. object-oriented programming (OOP) as a compositional mechanism in this domain and how the complete approach – the process of CIA, aspect-aware design and development – has emerged from that. CIA and aspect-aware design are evaluated and discussed on the examples of AUTOSAR OS and CiAO with respect to separation of concerns.

MobiSys '12 pp 87 ff — Borchert, Lohmann, and Spinczyk. "CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack" (Acceptance rate: 18%) [C5⋆]

Finally, the MobiSys '12 paper generalizes the applicability of the approach by applying it to a different domain: network stacks. It exercises the complete product-line development process (domain analysis, CIA, aspect-aware design and implementation) on the example of a TCP/IP stack and comes up with a new aspect role for layered system designs. The resulting CiAO/IP stack

is evaluated and compared with the state of the art regarding scalability (throughput, memory, energy consumption) and separation of concerns.

The work on CiAO arose out of my PhD [T1], helped by Olaf Spinczyk and two diploma students: Wanja Hofer developed the OSEK/AUTOSAR personality; Jochen Streicher developed the configurable memory protection. The work on CiAO/IP was conducted with Christoph Borchert (TU Dortmund) during his Masters and first PhD year, helped by Olaf Spinczyk.

4. The SLOTH Approach

While CiAO provides an excellent *up-tailorability* of system software towards the specific requirements of the application, it offers relatively few options [C21] to tailor it with respect to specific hardware properties. Many features of modern μ-controller hardware densely interact with each other; the resulting constraints make it difficult to express them as configurable features and to exploit them in system software. Instead, for the sake of platform independence, operating system designers try to abstract (arguably too early and too much [141]) from hardware as soon as possible by provisioning of a hardware abstraction layer (HAL), which defines an abstract machine the kernel is built on (cf. [38, 47]).

SLOTH kernels go the other way round [C19$\star$, C10$\star$, C7$\star$]: The goal here is to *embrace* the specific features of modern *commodity off-the-shelf hardware* as much as possible for the implementation of standard operating-system services: As CiAO, SLOTH kernels implement (most of) the automotive OSEK/AUTOSAR RTOS standards [48, 49, 69, 98], but "let the hardware do all the work" [O5].

4.1. SLOTH Goals

The major goal of SLOTH is to embrace platform-specific hardware particularities instead of blindly abstracting from them – in order to use them in an (unorthodox) way for the efficient implementation of core operating-system services, such as scheduling and dispatching.

A key concept in this realm is to implement *all* control flows (software-triggered, hardware-triggered, time-triggered) as interrupt handlers and let the hardware, namely the interrupt subsystem, do the scheduling and dispatching work (Figure 4.1). Hence, SLOTH implies some hardware requirements with respect to the interrupt subsystem: (1) It has to offer as many interrupt sources and priority levels as there are control flows in the system. (2) Interrupt requests can also be triggered from software. These properties, however, are provided by many modern μ-controller platforms. The Infineon TriCore, for instance, the SLOTH reference platform, provides 256 interrupt priority levels and more than 180 interrupt sources that can be triggered by software [71].

4.2. Implementation Approach: Generative Programming

The automatic tailoring of SLOTH towards the application and hardware is implemented using a *generative* approach, based on an custom, Perl-based DSL and generator framework: Application requirements and global policies are described in an (OIL-like [76])

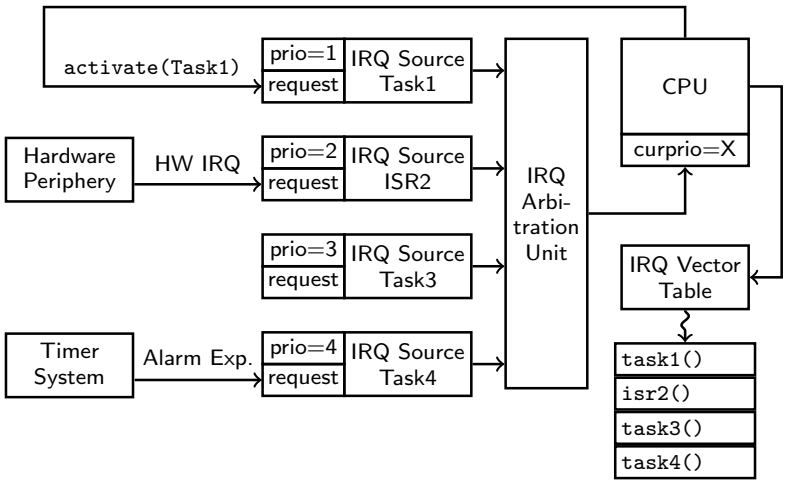


Figure 4.1.: Interrupt handlers are the unified control-flow abstraction in SLOTH
All control flows (tasks and ISRs) managed by the kernel are mapped to hardware interrupt sources. Activation takes place by either hardware events (`ISR2`, `Task4`) or software events (as in `activate(Task1)`). The IRQ arbitration unit *schedules* in hardware by reporting the highest-priority interrupt request to the CPU; if its priority is higher than `curprio`, the CPU *dispatches* in hardware by invoking the respective interrupt vector.

DSL, which is then used by the SLOTH generator framework to generate C code and linker scripts for the tailored kernel variant, which is highly optimized with respect to the mapping of the application's requirements to the specific target hardware. This "ideal" mapping of application requirements to hardware elements is achieved by a two-dimensional generation approach: The generation process is augmented by both architecture-specific and application-specific code-generation rules (Figure 4.2).

4.3. SLOTH Results

SLOTH kernels implement the conformance classes BCC1 [C19⋆] (all tasks are run-to-completion) and ECC1 [C10⋆] (tasks may block) of the OSEK OS standard [69] for event-triggered RTOS, as well as the OSEKtime standard [98] and AUTOSAR OS schedule tables [49] for time-triggered/mixed-mode RTOS. The generative approach facilitates portability to other platforms, even though the hardware-centric design requires a very deep interaction with hardware particularities. Besides the TriCore reference platform, SLOTH kernels are also available for the ARM (Cortex M3), PowerPC (Freescale MPC55xx), and IA32 (APIC) plattforms.

By its uniformed control-flow abstraction, SLOTH abolishes the artificial distinction between threads and interrupts: All control-flows managed by the kernel, may they be hardware-triggered, software-triggered, or time-triggered and bear blocking or nonblocking semantics, share the same mechanisms and priority space (which is the interrupt hardware and priority space).

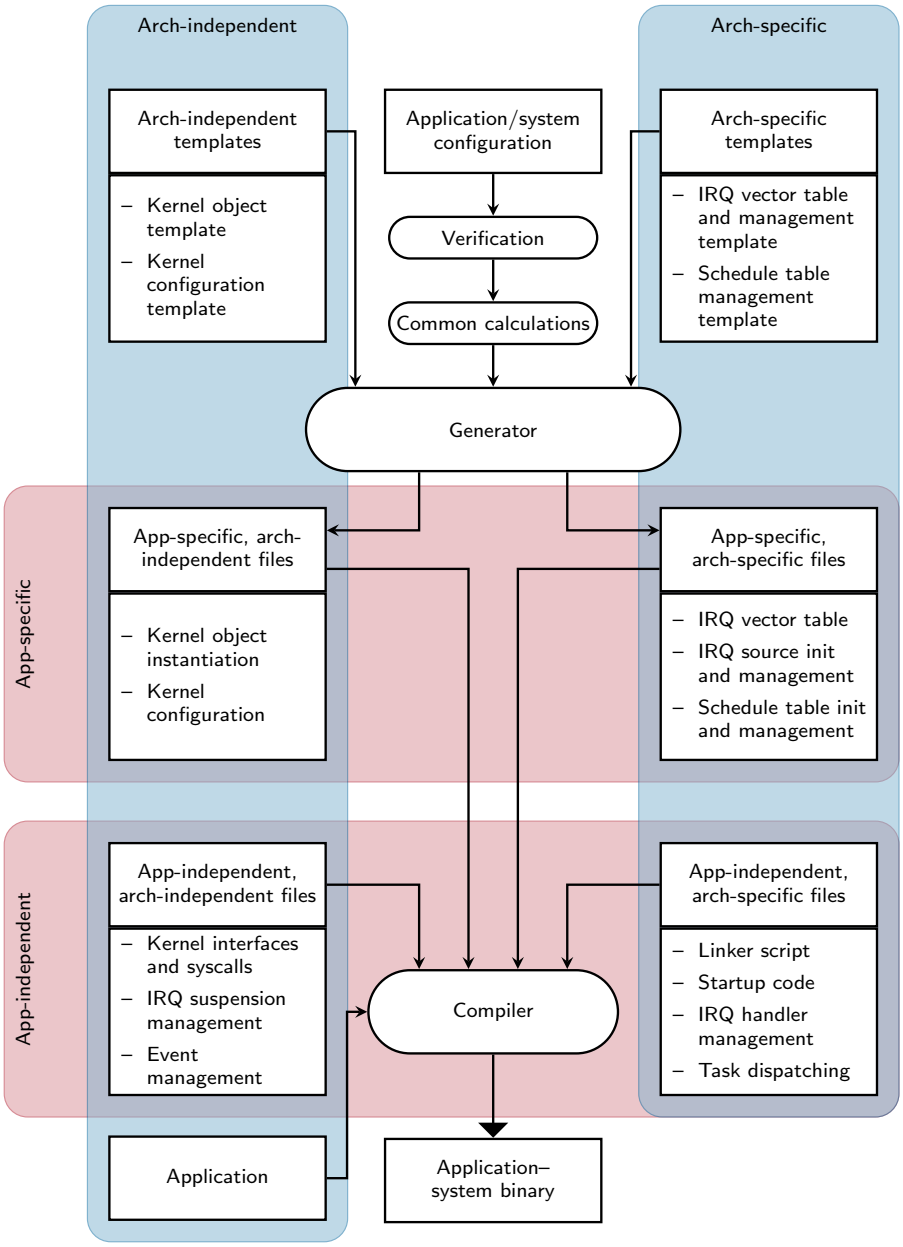


Figure 4.2.: Two-dimensional code generation with respect to *architecture* and *application*

Application properties and target architecture are given as an *application/system configuration* to the SLOTH generator, which combines *architecture-independent* and *architecture-dependent* generation rules to produce the *application-specific* tailored kernel code. The generated code is then passed together with *application-independent* parts of the kernel and the user code (*application*) to the compiler/linker to produce the resulting *application–system binary*.

This has a number of notable implications on important nonfunctional system properties: With its unified priority space for threads and interrupts, SLOTH avoids – by design – all issue of *rate-monotonic priority inversion* [53, 12][1] and achieves excellent priority obedience. Compared to leading commercial software-based implementations of OSEK OS, OSEKtime and AUTOSAR OS, SLOTH achieves kernel-time speedups of 1.3x–171x, event latencies as low as 14 clock cycles, and extremely low memory footprints [C19⋆, C10⋆, C7⋆].

4.4. SLOTH Key Papers

In the following, I briefly classify the role of the three related key papers, which are part of this cumulative habilitation treatise. Reprints of these papers are available in Appendix B.

RTSS '09 pp 101 ff — Hofer, Lohmann, Scheler, and Schröder-Preikschat. "Sloth: Threads as Interrupts" (Acceptance rate: 21%) [C19⋆]

The RTSS '09 paper introduces the SLOTH idea: To map thread executions to interrupts in order to have the interrupt controller of commodity hardware do all the scheduling and dispatching work and unify the priority space of interrupts and software tasks. The paper describes the implementation approach and compares SLOTH with CiAO with respect to event latencies and priority obedience. As interrupt handlers, software tasks in SLOTH must bear run-to-completion semantics, that is, they may not block. Nevertheless, SLOTH thereby already implements the BCC1 conformance class of the OSEK OS [69] standard.

RTSS '11 pp 111 ff — Hofer, Lohmann, and Schröder-Preikschat. "Sleepy Sloth: Threads as Interrupts as Threads" (Acceptance rate: 21%) [C10⋆]

The RTSS '11 paper complements the RTSS '09 paper with respect to blocking control flows. SLEEPY SLOTH provides a universal control-flow abstraction that, as in SLOTH, delegates all scheduling and dispatching work to the interrupt controller, but additionally provides blocking control flows, which may be either tasks or interrupts. The paper describes the implementation approach with the resulting SLEEPY SLOTH generation process and compares SLEEPY SLOTH to SLOTH and a commercial OSEK OS implementation. With its support for blocking as well as run-to-completion tasks, SLEEPY SLOTH implements the ECC1 conformance class of OSEK OS while still being fast.

[1] This kind of priority inversion occurs if a (logically) low-priority interrupt (such as a timer that activates a low-priority task) interrupts a high-priority software task. This effect can be observed in most RTOS, as interrupts are managed separately from tasks in the interrupt priority space, which has priority over all software tasks. The dual priority space has long been identified as a major obstacle in real-time software development: "Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements." [113]

RTSS '12 pp 123 ff — Hofer, Danner, Müller, Scheler, Schröder-Preikschat, and Lohmann. "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS" (Acceptance rate: 22%) [C7⋆]

Finally, the RTSS '12 paper presents SLOTH ON TIME, which generalizes the SLOTH idea "to let the hardware do the work" from event-triggered systems to time-triggered systems by mapping schedule tables, deadline monitoring, time synchronization, and execution budgeting to commodity hardware timer arrays. The paper describes the implementation and refined generation approach and compares SLOTH ON TIME to commercial OSEKtime and AUTOSAR OS implementations. SLOTH ON TIME extends SLEEPY SLOTH by the support for the OSEKtime standard [98] and AUTOSAR OS schedule tables [49] for time-triggered as well as mixed-mode RTOS.

The work on SLOTH, SLEEPY SLOTH, and SLOTH ON TIME was conducted with Wanja Hofer during his PhD [5], helped by Wolfgang Schröder-Preikschat. Daniel Danner developed the timer-cell mapping and respective generator for SLOTH ON TIME during his Masters.

5. The VAMOS Approach

Both the aspect-aware CiAO approach and the generative SLOTH approach lead to operating-system kernels and other pieces of system software that are highly configurable and provide excellent tailorability towards specific application requirements or hardware properties. However, they are *constructive* approaches that have to be applied from the very beginning of the development process. This limits their value for existing configurable system software, such as Linux.

The VAMOS approach is different in that it is an *analytical* approach. We aim to address issues of maintaining and managing variability in *existing* large-scale system software and *independently* of the underlying implementation techniques.

In the literature, configurability in system software is perceived as implemented mainly by means of the CPP [37, 23, 17, 136, 122]. In fact, real-world software employs a *multitude* of tools and languages for this purpose: The Linux v3.2 kernel, for instance, provides nearly twelve thousand configurable features (11,863 unique KCONFIG items) in its feature model. These features first control the *configuration-dependent* inclusion or exclusion of 28,000 source files by the build system (KBUILD rules, coarse-grained variability in Figure 5.1), which in turn contain 84,000 `#ifdef` blocks that are evaluated by the CPP (fine-grained variability in Figure 5.1) before the source code is finally passed to the compiler and (in the case of linker scripts, which may also be preprocessed by CPP) the linker [C6⋆]. Thus, in Linux the implementation of variation points and their constraints is spread over at least three interacting levels: feature model, build system, and preprocessor; each level constrains the configurable variability on the subsequent levels (Figure 5.1).

5.1. VAMOS Goals

This distribution of the variability implementation imposes big challenges for Linux developers with respect to ensuring (a) consistency of variability information and (b) coverage of configuration-conditional code. Furthermore, the high number of available features impose a challenge to Linux users with respect to (c) configuring an actual Linux kernel for a concrete use case.

The goal of the VAMOS project is to provide methods and tool support to deal with such issues in large-scale and highly distributed implementations of variability.

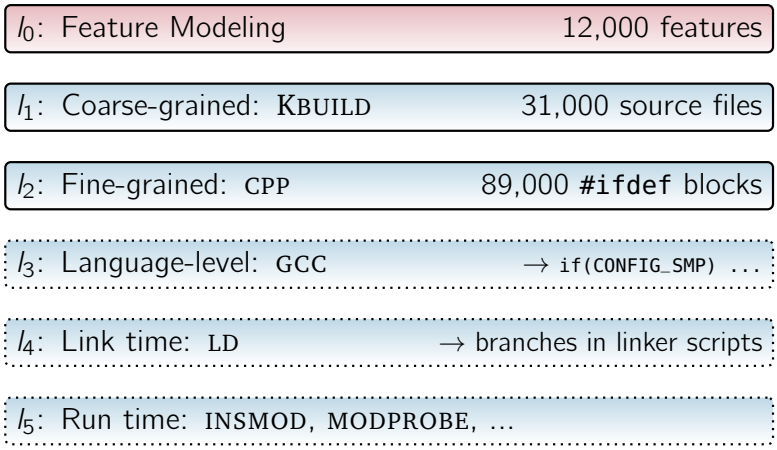


Figure 5.1.: Implementation levels of software variability in Linux

Each level effectively constrains the possible variability in the subsequent levels (for instance, the coarse-grained variability implemented in the build system (KBUILD) dominates fine-grained variability implemented by the preprocessor (CPP)). Further static variation points may be implemented on the language level or linker level. However, as of kernel version 3.2, this is not common in Linux.

5.2. Implementation Approach: Holistic Variability Model

The basic idea of VAMOS is to extract the variation points (features and constraints) from *all* sources of variability into a common *holistic variability model* based on propositional logic (Figure 5.2). In the case of Linux this is mostly variability that is implemented with *decompositional* approaches.

The holistic variability model is then be used to provide tools that address issues (a) to (c) in an automated manner [C14⋆, C6⋆, W5⋆, C17, J2, J3].

5.3. VAMOS Results

Our results show that variability has to be considered as a significant source of software bugs in its own respect – all of which could be found upfront by better tools support.

In [C14⋆], we have presented tool support to address issue (a) – that is, to automatically crosscheck the intended variability (expressed by the KCONFIG feature model) with the actually implemented variability (expressed by CPP statements in the code): We have found more than 1,700 *variability defects* – inconsistencies that manifest as dead #ifdef code or actual bugs. Our results have led to (accepted) fixes for more than 350 of them, including 20 new bugs and the removal of 5,000 superfluous lines of #ifdef code in Linux v2.6.36.

However, these numbers turned out to be just the tip of the iceberg: They did not take the KBUILD build system into account – which we found in Linux to have effect on more

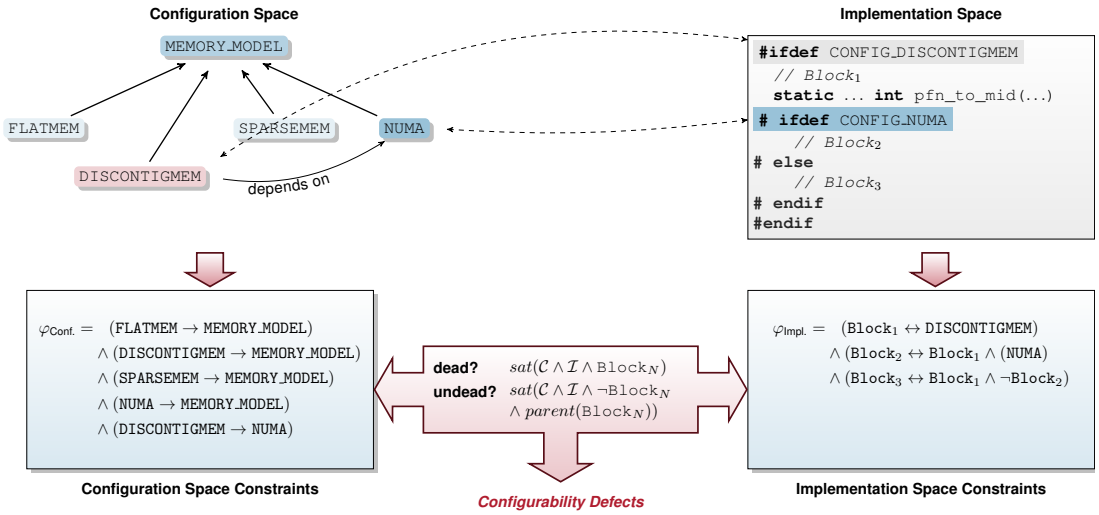


Figure 5.2.: The VAMOS approach at a glance

Variability information (features and their dependencies) is extracted from all sources of variability (here: KCONFIG and CPP level) into a holistic variability format based on propositional logic. A SAT solver can then be employed to check, for instance, for contradictions and tautologies regarding the reachability of each `#ifdef` block.

than two thirds (!) of all KCONFIG features [W3]. Extending the holistic variability model with the variability implemented in the build system [C6⋆] increases the number of found defects to more than 2,500 in Linux v3.2.

With respect to issue (b), we have extended our tool support to automatically derive configurations that maximize the coverage of configuration-conditional code (`#ifdef` blocks) by static analysis tools [W11]. For Linux/`arm` v3.2, we thereby have – just using the compiler as a static checker – found 91 new bugs (not yet published).

However, our holistic variability model does not only help developers to maintain feature-related code in highly configurable system software, but also users can benefit from this: With respect to issue (c), the 12,000 configuration options of Linux have rendered it nearly impossible for the average user to tailor a Linux kernel for this specific use case. In [W5⋆] we have presented an approach to automate this process. On the base of a kernel trace, our tools automatically derive a Linux configuration that includes only the necessary features. For a typical webserver setup (LAMP: Linux, Apache, MySQL, PhP), the resulting configuration enables only about ten percent of the features and executable code as enabled in a standard Linux Debian kernel, which here leads to a significant reduction of the attack surface exploitable by a potential attacker.

5.4. VAMOS Key Papers

In the following, I briefly classify the role of the three related key papers, which are part of this cumulative habilitation treatise. Reprints of these papers are available in Appendix B.

EuroSys '11 pp 135 ff — Tartler, Lohmann, Sincero, and Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem" (Acceptance rate: 15%) [C14⋆]

The EuroSys '11 paper introduces the problem of configurability-related software defects in large-scale system software on the example of Linux. It classifies sources and characteristics of such defects and presents the VAMOS approach of generating a holistic variability model to detect such defects early in the development process. Feasibility and scalability to the size of Linux is demonstrated by the implementation in the UNDERTAKER tool, which covers CPP- and KCONFIG-induced variability. The actual relevance of our findings is evaluated by analyzing the reaction of the Linux community to 123 submitted patches that fix 364 defects.

SPLC '12 pp 149 ff — Dietrich, Tartler, Schröder-Preikschat, and Lohmann. "A Robust Approach for Variability Extraction from the Linux Build System" (Acceptance rate: 33%) [C6⋆]

The SPLC '12 paper complements the VAMOS approach by an important building block: Variability extraction from the build system, which in Linux influences the code generation for two thirds of all features. However, extracting this variability information is difficult due to the declarative and Turing-complete MAKE language; previous approaches that rely on simple text processing of build scripts are inherently incomplete and have to be tailored to a specific Linux variant. The paper presents a robust probing approach that exploits the build system itself to extract the variability model. It describes the implementation in the VAMPYR tool and evaluates the probing approach by comparing it to existing text-based extraction approaches with respect to robustness, correctness, and impact on UNDERTAKER findings.

HotDep '12 pp 159 ff — Tartler, Kurmus, Heinloth, Rothberg, Ruprecht, Doreanu, Kapitza, Schröder-Preikschat, and Lohmann. "Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability" (Acceptance rate: 42%) [W5⋆]

Finally, the HotDep '12 paper generalizes the VAMOS approach towards a new use case: The automatic configuration of Linux with respect to a specific application. Based on the holistic variability model and a kernel trace with a distro kernel, UNDERTAKER automatically derives a tailored configuration that is optimized with respect to a nonfunctional property, which here is kernel attack surface. The resulting kernel is compared against the distribution kernel with respect to code size, known vulnerabilities, and throughput.

The work on VAMOS was conducted with Julio Sincero and Reinhard Tartler during their PhDs [6, 7]. Julio developed the `CPP` extractor and the initial version of the KCONFIG extractor; Reinhard refined the KCONFIG extractor and investigated, with help of bachelor student Christian Dietrich, the probing-based KBUILD extractor. Reinhard also conducted, together with Anil Kurmus (IBM Research), the automatic tailoring approach, helped by Bernhard Heinloth, Anderas Ruprecht, and Valentin Rothberg, who implemented the tools as part of their master's project.

6. Discussion, Future Work, and Conclusions

The CiAO, SLOTH, and VAMOS approaches briefly introduced in the preceding sections all address challenges in the design, implementation, and long-term controllability of variability in tailorable system software: Tailorable system software (Goal) is conceptually understood as a software product line; the functional and nonfunctional properties are modelled as optional and mandatory features and constraints (Method). The resulting variability is implemented (and maintained) in the code by decompositional, compositional, and generative approaches (Technique).

In the following, I discuss some broader aspects of the three approaches – together with my ideas for further research – and conclude my work.

6.1. Impact on Functional and Nonfunctional Properties

Tailoring basically means to leave out all unneeded *optional* features and to choose the "right" variant of all *alternative* features – "right" with respect to important nonfunctional properties, such as memory footprint, throughput, event latency, robustness, jitter, and so on. The "ideal" system software does not impair these properties by abstractions and policies that do not serve the application's needs. Both CiAO and SLOTH excel in this respect, but differ fundamentally in their design and implementation strategy:

- The CiAO approach leads to excellent *up-tailorability* of embedded system software (towards the requirements of a specific application) by its aspect-aware design and implementation approach. The resulting fine-grained separation of concerns and the strict decoupling of policies and mechanisms in the implementation make it easy to provide many optional and alternative features.
- The SLOTH approach leads to excellent *down-tailorability* of embedded system software (towards better exploitation of hardware) by its generative design and implementation approach. The generated SLOTH kernels map the application's control flows to the features offered by the hardware in an "ideal manner".

Both the CiAO and SLOTH kernels, implement (most of) the OSEK OS and AUTOSAR OS standards. So which approach leads to better system software?

- CiAO actually provides more than requested by OSEK OS and AUTOSAR OS. For the sake of *further* tailorability with respect to functional and nonfunctional properties, CiAO offers *alternatives* for many mechanisms and strategies (protection, synchronization, locking protocols, scheduling, ...). These can be chosen to fine-tune the

trade-off between relevant nonfunctional properties. An example is *priority obedience*: CiAO is, as most RTOSs that feature a software scheduler, subject to issues of rate-monotonic priority inversion. However, on the TriCore platform, CiAO provides an extension aspect [C20⋆] that maps interrupts to tasks by delegating all interrupt requests to the peripheral control processor (PCP) featured on this platform; the PCP runs truly parallel to the main CPU. Its program activates the corresponding task, but only interrupts the main CPU if the priority of the incoming interrupt is higher than the running task's priority [C21]. This completely prevents rate-monotonic priority inversion at the price of 3x–4x higher interrupt latencies (caused by the PCP detour) – the application developer is offered the alternative to trade latency for priority obedience.

- SLOTH, in contrast, provides only few options and alternatives. Functionally provided is what can be mapped efficiently to the hardware. The "unfiltrated" efficiency that results from the application–hardware specific generation of the kernel leads to an "almost-optimal" solution with respect to *many* nonfunctional properties: memory footprint, performance, latency, and priority obedience. Hence, for these properties, there is simply no need to trade one for another and thus no need for alternatives: The bare efficiency of SLOTH-based kernels often mitigates the need for alternative policies – as long as the mapping to hardware *is* actually possible: The control-flow model of OSEK is rewarding in this sense. Nevertheless, a higher number of tasks (exceeding the number of interrupt sources) or a different functionality (e.g., an earliest deadline first (EDF) scheduler) would be easy to integrate into CiAO, but would require to introduce software-based scheduling in SLOTH. This would reduce the "bare efficiency" effect and, thus, also increase the demand for alternatives of other features in order to keep the system tailorable.

The bottom line is that the SLOTH, in comparison to CiAO, trades efficiency (by implicit down-tailorability to the hardware) for generality (by explicit *up-tailorability* to the application). It should be possible to combine both approaches without generally trading one for another. This is a topic for further research.

6.2. Explicit, Implicit, and Automatic Tailoring

Another point is the influence of the implementation technique on the process of tailoring itself. CiAO and CiAO/IP are modelled as software product lines; each instance has to be tailored *explicitly* by providing an application description and an explicit configuration with respect to the 674 features offered by the CiAO feature model. SLOTH basically only requires the application description; most of the tailoring is then performed *implicitly* by the generator with respect to the application description and selected target hardware. In general, I consider this implicit tailoring preferable, as it removes the burden from the user to manually decide on the "right" configuration. However, implicit tailoring requires exploiting domain-specific knowledge in generators – it limits the generality of the system software.

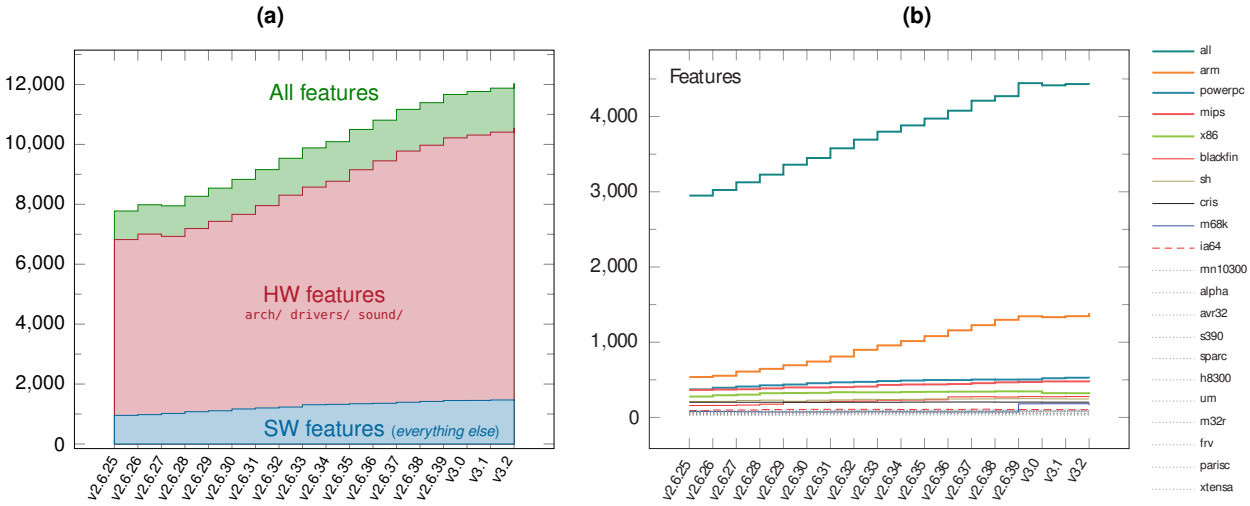


Figure 6.1.: Feature growth in Linux with respect to hardware/software-related functions

(a) Hardware support is the driving force of feature growth in Linux; since Linux V2.6.25, only 12–13 percent of all Linux features have been related to pure software functionality, 87–88 percent are dedicated to specific hardware support (subsystems `arch`, `drivers`, `sound`). **(b)** The growth of hardware-related features is dominated by the `arch` subsystem on Linux/`arm`.

Hence, for broad-scale system software, such as Linux with its twelve thousand features and application domains ranging from enterprise servers to mobile phones and appliances, *automatic tailoring* by the automatic derivation of a good initial configuration might be a reasonable alternative. This is an active field of research: With the VAMOS approach, we have shown that it is possible to automatically detect required functional features from a run-time trace and thereby indirectly optimize nonfunctional properties, such as code footprint and attack surface [W5⋆, C1]. Other researchers have investigated static source-code analysis for this purpose [44] or suggested systematic testing techniques for the automatic prediction [14] and incremental optimization [15] of nonfunctional properties. It would be interesting to combine these techniques in order to scale to the (increasing) configurability of system software, such as Linux. This is a topic for further research.

On the other hand: For specific domains, such as automotive control systems, I do see quite some potential for implicit tailoring – towards further nonfunctional properties (such as memory safety and fault resistance) – by generators and also compilers that are aware of the application domain, hardware properties, and operating-system semantics. Exploiting static knowledge across the *complete* stack [W7, W12, C13, W17, C4, W1, 26, 137, 117] (hardware, operating system, middleware, application) remains a promising field for further research.

But even Linux could profit from more implicit tailoring. I am convinced that explicit tailoring (even with automation support) is going to hit its limits in the case of Linux:

Since 2005, the number of configurable features has grown by 10–20 percent *every year* – mostly caused by advances in hardware: About 88 percent of all features directly deal with low-level hardware support (subsystems `arch`, `drivers`, `sound`, see Figure 6.1a). The rising ARM platform (Android smartphones) with its many derivatives and short innovation cycles has become the driving force in this process (Figure 6.1b). However, at the same time the "quality" of features (as conceptual abstractions) is going down: Many new features on Linux/`arm` address only hardware revisions, subtle device differences, or specific silicon bugs. Such features are difficult or even impossible to find by the automatic tailoring approaches described above. Hence, deriving a valid (and running) configuration for some particular ARM-based device requires a lot of arcane knowledge about this device – knowledge that, as SLOTH has shown, could better be hidden inside generators to generate device-specific parts of the code. Device-specific code generation is an active field of research [104, 75, 33], even though it remains unrealistic that drivers can completely be generated [11]. Nevertheless, the potential of integrating generative techniques into Linux in order to *reduce* the number of features is another topic of further research.

6.3. Multi-Level Separation of Concerns

Incorporating generative techniques into the Linux build process would result in even more implementation levels of software variability (Figure 5.1) – potentially increasing the complexity of software quality measures, evolution, and maintenance tasks even further. This holds especially for all features that are implemented *across* multiple levels or that interact with features implemented on other levels. So what are the broader implications regarding separation of concerns [147] and feature modularity [3, 35]?

As demonstrated by CiAO, language-based compositional approaches can be beneficial for the implementation of feature modularity, especially of interacting features. However, this always holds for just a single implementation level of variability – and there never is just one level: Even in CiAO we also have a feature model and a build system, both of which implement parts of the variability. The idea to have just one level and a single language to express all variability might be tempting, but I am convinced that this is neither conceptually nor practically achievable. The modularity problem of scattered feature implementations has to be solved at the tool level!

Lifting modularity as a concept from the language to the tool level has been suggested (for a single implementation level only) under the term *virtual separation of concerns* for decompositional, annotation-based variability. The general idea is to solve the common issues of "`#ifdef` hell" by providing feature-related integration views on the software system with variability-aware IDEs [34, 29, 46, 24] or by a variability-aware file system [W15].

What is needed – and what is a topic of further research – is an approach and tool support for virtual separation of concerns *across* all implementation levels of variability, independently of the actual implementation technique used on each level. It should become possible to (a) analyze, (b) visualize, and (c) control feature implementations

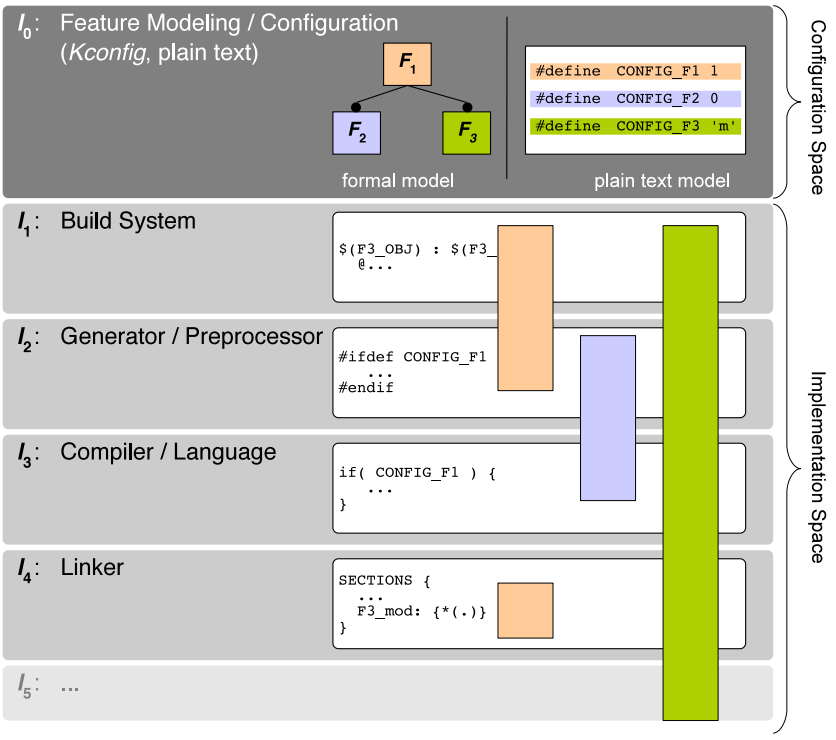


Figure 6.2.: The vision of multi-level separation of concerns for configurable features
Implementation fragments of a particular feature can be analyzed, visualized, and controlled as "cross-layer aspects" – virtual modules that describe variability over more than one level and implementation technique.

across all levels: The implementation of a feature may then combine compositional, decompositional, and generative techniques; its implementation parts, which technically are scattered over different levels and languages, are represented and maintained as a virtual feature module.

With respect to (a), the VAMOS approach with its holistic variability model might be a good starting point, also for (b) as a foundation for detecting and grouping related cross-level feature fragments with the technique suggested in [W15]. For (c), we need some sort of "cross-layer aspects" that can augment the implementation across multiple levels. Our work on aspect-oriented programming [J7], but also semantic patch languages, such as SmPL [55, 54, 36, 19] or C4 [60] might be a suitable approach for this purpose, ideally extended by some problem-specific cross-level/language constraint system, as we have suggested in the product-line component (PLiC) approach [C16, C15].

6.4. Conclusions

Providing no business value of its own, system software is expected to provide the "right" set of abstractions for a particular application use case: The functional and

nonfunctional requirements of the application have to be mapped *efficiently* to the functional and nonfunctional properties of the hardware. This is of particular importance in the hardware-cost–sensitive domain of deeply embedded systems, such as automotive control units.

Efficiency calls for tailored system software, reusability calls for generic system software. To overcome this dilemma, most system software provides built-in static variability; it can be configured at compile time to tailor it towards a specific use case. In my research, I have investigated methods and techniques towards highly tailorable system software with focus on the design, implementation, and maintenance of fine-grained static variability.

The resulting CiAO (compositional, aspect-oriented implementation of variability) and SLOTH (generative, two-dimensional implementation of variability) approaches have both been successfully applied to the construction of *new* system software (operating systems, network stacks). The CiAO and SLOTH families of RTOSs and the CiAO/IP family of TCP/IP network stacks offer – by their tailorability – unprecedented excellence with respect to many important nonfunctional properties, including memory footprint, event latency, priority obedience, jitter, throughput and energy consumption. The analytical VAMOS approach (decompositional, multi-level implementation of variability) has successfully been applied to mitigate maintenance, quality, and configuration challenges resulting from static variability in *existing* large-scale system software. In Linux we have, by our holistic variability model, found hundreds of bugs and thousands of lines of dead `#ifdef` code – and contributed accepted fixes for many of them. By automatic tailoring, we could reduce the attackable code size of special-purpose Linux installations by nearly ninety percent.

While writing these last lines of this habilitation treatise, Linux has reached version number v3.8 with nearly 13,000 (12,857) features. New hardware features that need to be abstracted *and* embraced by configurable system software appear every day. Tailorability as a system property – and, thus, the efficient and maintainable implementation thereof – is becoming more important every day.

In this respect, I envision the flexible and problem-oriented integration and combination of *all* implementation techniques for variability by methods and tools for *multi-level separation of concerns* – towards more implicit *down-tailoring*, rich functional *up-tailoring*, and the long-term *controllability* of variability for both developers and users.

A. Bibliography

A.1. General Bibliography

[1] *GNU M4 – GNU Project – Free Software Foundation (FSF)*. http://www.gnu.org/software/m4/, visited 2010-07-29. URL: http://www.gnu.org/software/m4/.

[2] *eCos homepage*. visited 2013-03-19. URL: http://www.ecoscentric.com/ecos/index.shtml.

[3] C. Kästner, S. Apel, and K. Ostermann. "The road to feature modularity?" In: *Proceedings of the 15th Software Product Line Conference (SPLC '11), Volume 2*. (FOSD '11 Proceedings). ACM Press. ISBN: 978-1-4503-0789-5. DOI: 10.1145/2019136.2019142.

[4] C. Borchert, H. Schirmeier, and O. Spinczyk. "Generative Software-based Memory Error Detection and Correction for Operating System Data Structures." In: *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN '13)*. IEEE Computer Society Press, 2013.

[5] W. Hofer. "Sloth: The Virtue and Vice of Latency Hiding in Hardware-Centric Operating Systems." PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[6] J. Sincero. "Variability Bugs in System Software." PhD thesis. Friedrich-Alexander University Erlangen-Nuremberg, 2013.

[7] R. Tartler. "Mastering Variability Challenges in Linux and Related Highly-Configurable System Software." PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[8] T. Berger, S. She, R. Lotufo, and A. W. und Krzysztof Czarnecki. *Variability Modeling in the Systems Software Domain*. Tech. rep. GSDLAB-TR 2012-07-06. Generative Software Development Laboratory, University of Waterloo, 2012. URL: http://gsd.uwaterloo.ca/sites/default/files/vm-2012-berger.pdf.

[9] J. Corbet, G. Kroah-Hartman, and A. McPherson. *Linux Kernel Development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. The Linux Foundation, 2012.

[10] P. Gazzillo and R. Grimm. "SuperC: parsing all of C by taming the preprocessor." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM Press, 2012, pp. 323–334. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254103.

[11] A. Kadav and M. M. Swift. "Understanding modern device drivers." In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*. ACM Press, 2012, pp. 87–98. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150987.

[12] L. E. Leyva-del-Foyo, P. Mejia-Alvarez, and D. de Niz. "Integrated Task and Interrupt Management for Real-Time Systems." In: *Transactions on Embedded Computing Systems* 11.2 (2012), 32:1–32:31. ISSN: 1539-9087. DOI: 10.1145/2220336.2220344. URL: http://doi.acm.org/10.1145/2220336.2220344.

[13] S. Nadi and R. C. Holt. "Mining Kbuild to Detect Variability Anomalies in Linux." In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. (Mar. 27–30, 2012). IEEE Computer Society Press, 2012. ISBN: 978-1-4673-0984-4. DOI: 10.1109/CSMR.2012.21.

[14] N. Siegmund, S. Kolesnikov, C. Kastner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake. "Predicting performance via automated feature-interaction detection." In: *Proceedings of the 34nd International Conference on Software Engineering (ICSE '12)*. IEEE Computer Society Press, 2012, pp. 167–177. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227196.

[15] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. "SPL Conqueror: Toward optimization of non-functional properties in software product lines." English. In: *Software Quality Journal* 20.3-4 (2012), pp. 487–517. ISSN: 0963-9314. DOI: 10.1007/s11219-011-9152-9. URL: http://dx.doi.org/10.1007/s11219-011-9152-9.

[16] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation." In: *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. ACM Press, 2011. DOI: 10.1145/2048066.2048128.

[17] J. Liebig, C. Kästner, and S. Apel. "Analyzing the discipline of preprocessor annotations in 30 million lines of C code." In: *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD '11)*. ACM Press, 2011, pp. 191–202. ISBN: 978-1-4503-0605-8. DOI: 10.1145/1960275.1960299.

[18] "Microsoft's Protocol Documentation Program: Interoperability Testing at Scale." In: *Queue* 9 (6 2011). A Discussion with Nico Kicillof, Wolfgang Grieskamp and Bob Binder, 20:20–20:27. ISSN: 1542-7730. DOI: 10.1145/1989748.1996412.

[19] N. Palix, G. Thomas, S. Saha, C. Calvès, J. L. Lawall, and G. Muller. "Faults in Linux: Ten years later." In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. ACM Press, 2011, pp. 305–318. DOI: 10.1145/1950365.1950401.

[20] A. Warg and A. Lackorzynski. "Rounding pointers: type safe capabilities with C++ meta programming." In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. ACM Press, 2011, 3:1–3:5. ISBN: 978-1-4503-0979-0. DOI: 10.1145/2039239.2039244. URL: http://doi.acm.org/10.1145/2039239.2039244.

[21] T. Berger and S. She. *Formal Semantics of the CDL Language*. Technical Note. University of Leipzig, 2010.

[22] T. Berger, S. She, R. Lotufo, and A. W. und Krzysztof Czarnecki. "Variability Modeling in the Real: A Perspective from the Operating Systems Domain." In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM Press, 2010, pp. 73–82. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859010.

[23] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines." In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*. ACM Press, 2010. DOI: 10.1145/1806799.1806819.

[24] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. "Emergent feature modularization." In: *Proceedings of the 25th ACM Conference Companion on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH '10)*. ACM Press, 2010, pp. 11–18. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869545.

[25] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. "Delta-oriented programming of software product lines." In: *Proceedings of the 14th Software Product Line Conference (SPLC '10)*. Vol. 6287. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 77–91. ISBN: 978-3-642-15578-9. DOI: 10.1007/978-3-642-15579-6_6.

[26] F. Scheler and W. Schröder-Preikschat. "The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems." In: *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '10)*. IEEE Computer Society Press, 2010, pp. 34–41. ISBN: 978-0-7695-4037-5. DOI: 10.1109/ISORC.2010.11.

[27] S. She and T. Berger. *Formal Semantics of the Kconfig Language*. Technical Note. University of Waterloo, 2010.

[28] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. "The Variability Model of the Linux Kernel." In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '10)*. 2010.

[29] S. Apel and C. Kästner. “Virtual Separation of Concerns - A Second Chance for Preprocessors.” In: *Journal of Object Technology* 8.6 (2009), pp. 59–78.

[30] A. Borisov. “Coreboot at your service!” In: *Linux Journal* 1 (186 2009).

[31] P.-E. Dagand, A. Baumann, and T. Roscoe. “Filet-o-Fish: practical and dependable domain-specific languages for OS development.” In: *Proceedings of the 5th Workshop on Programming Languages and Operating Systems (PLOS ’09)*. ACM Press, 2009, 5:1–5:5. ISBN: 978-1-60558-844-5. DOI: 10.1145/1745438.1745446.

[32] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. “On the Impact of the Optional Feature Problem: Analysis and Case Studies.” In: *Proceedings of the 13th Software Product Line Conference (SPLC ’09)*. Carnegie Mellon University, 2009. ISBN: 978-0-9786956-2-0.

[33] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. “Automatic device driver synthesis with termite.” In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. ACM Press, 2009, pp. 73–86. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629583.

[34] C. Kästner, S. Apel, and M. Kuhlemann. “Granularity in Software Product Lines.” In: *Proceedings of the 30th International Conference on Software Engineering (ICSE ’08)*. ACM Press, 2008, pp. 311–320. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368131.

[35] C. Kim, H. Peter, C. Kästner, and D. Batory. “On the Modularity of Feature Interactions.” In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE ’08)*. ACM Press, 2008, pp. 23–34. ISBN: 978-1-60558-267-2. DOI: 10.1145/1449913.1449919.

[36] Y. Padioleau, J. L. Lawall, G. Muller, and R. R. Hansen. “Documenting and Automating Collateral Evolutions in Linux Device Drivers.” In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys ’08)*. ACM Press, 2008.

[37] D. Spinellis. “A Tale of Four Kernels.” In: *Proceedings of the 30th International Conference on Software Engineering (ICSE ’08)*. ACM Press, 2008, pp. 381–390. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368140.

[38] W. Stallings. *Operating Systems. Internals and Design Principles*. Sixth. Prentice Hall PTR, 2008. ISBN: 978-0136006329.

[39] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter. “Design recovery and maintenance of build systems.” In: *Proceedings of the 23st IEEE International Conference on Software Maintainance (ICSM’07)*. IEEE Computer Society Press, 2007, pp. 114–123. ISBN: 978-1-4244-1256-3. DOI: 10.1109/ICSM.2007.4362624.

[40] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter. “The Evolution of the Linux Build System.” In: *Electronic Communications of the EASST* (2007). ISSN: 1863-2122.

[41] C. Kästner, S. Apel, and D. Batory. “A Case Study Implementing Features Using AspectJ.” In: *Proceedings of the 11th Software Product Line Conference (SPLC ’07)*. IEEE Computer Society Press, 2007, pp. 223–232. DOI: 10.1109/SPLC.2007.5.

[42] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. “Integrating Concurrency Control and Energy Management in Device Drivers.” In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP ’07)*. ACM Press, 2007, pp. 251–264. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294286.

[43] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. “Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis.” In: *Proceedings of the 15th IEEE Conference on Requirements Engineering (RE ’07)*. (Oct. 15–19, 2007). IEEE Computer Society, 2007, pp. 243–253. ISBN: 0-7695-2935-6. DOI: 10.1109/RE.2007.61.

[44] H. Schirmeier and O. Spinczyk. “Tailoring Infrastructure Software Product Lines by Static Application Analysis.” In: *Proceedings of the 11th Software Product Line Conference (SPLC ’07)*. IEEE Computer Society Press, 2007, pp. 255–260. ISBN: 0-7695-2888-0. DOI: 10.1109/SPLINE.2007.33.

[45] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. “Is The Linux Kernel a Software Product Line?” In: *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*. 2007.

[46] N. Singh, C. Gibbs, and Y. Coady. “C-CLR: A Tool for Navigating Highly Configurable System Software.” In: *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’07)*. ACM Press, 2007, pp. 1–6. ISBN: 1-59593-657-8. DOI: 10.1145/1233901.1233910.

[47] A. S. Tanenbaum. *Modern Operating Systems*. Third. Prentice Hall PTR, 2007. ISBN: 978-0136006633.

[48] AUTOSAR. *Requirements on Operating System (Version 2.0.1)*. Tech. rep. Automotive Open System Architecture GbR, 2006.

[49] AUTOSAR. *Specification of Operating System (Version 2.0.1)*. Tech. rep. Automotive Open System Architecture GbR, 2006.

[50] D. Beuche. *Variant Management with pure::variants*. Tech. rep. http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf, visited 2011-11-12. pure-systems GmbH, 2006.

[51] M. Engel and B. Freisleben. “TOSKANA: A Toolkit for Operating System Kernel Aspects.” In: *Transactions on AOSD II*. Lecture Notes in Computer Science 4242. Springer-Verlag, 2006, pp. 182–226.

[52] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. “K42: building a complete operating system.” In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys ’06)*. ACM Press, 2006, pp. 133–145. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217949.

[53] L. E. Leyva-del-Foyo, P. Mejia-Alvarez, and D. de Niz. “Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware.” In: *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS ’06)*. IEEE Computer Society Press, 2006, pp. 14–23. DOI: 10.1109/RTAS.2006.34.

[54] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller. “Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers.” In: *Proceedings of the Linguistic Support for Modern Operating Systems ASPLOS XII Workshop (PLOS ’06)*. ACM Press, 2006.

[55] Y. Padioleau, J. L. Lawall, and G. Muller. “SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers.” In: *International ERCIM Workshop on Software Evolution*. 2006.

[56] J. Siadat, R. J. Walker, and C. Kiddle. “Optimization Aspects in Network Simulation.” In: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD ’06)*. ACM Press, 2006, pp. 122–133.

[57] D. D. Silva, O. Krieger, R. W. Wisniewski, A. Waterland, D. Tam, and A. Baumann. “K42: an infrastructure for operating system research.” In: *ACM SIGOPS Operating Systems Review* 40.2 (2006), pp. 34–42. DOI: 10.1145/1131322.1131333.

[58] F. Steimann. “The Paradoxical Success of Aspect-Oriented Programming.” In: *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’06)*. ACM Press, 2006, pp. 481–497. DOI: 10.1145/1167515.1167514.

[59] A. Tešanović, M. Amirijoo, and J. Hansson. “Providing Configurable QoS Management in Real-Time Systems with QoS Aspect Packages.” In: *Transactions on AOSD II*. Lecture Notes in Computer Science 4242. Springer-Verlag, 2006, pp. 256–288.

[60] M. Yuen, M. E. Fiuczynski, R. Grimm, and Y. Coady. “Making Extensibility of System Software Practical with the C4 Toolkit.” In: *Proceedings of the 4th AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies (AOSD-SPLAT ’06)*. ACM Press, 2006.

[61] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. “FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming.” In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE ’05)*. 2005.

[62] Delta Software Technology GmbH. *Angie – An Introduction*. 2005. URL: http://www.d-s-t-g.com/angie.

[63] M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. “patch(1) Considered Harmful.” In: *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS ’05)*. USENIX Association, 2005.

[64] D. Gay, P. Levis, and D. Culler. “Software Design Patterns for TinyOS.” In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’05)*. ACM Press, 2005, pp. 40–49. ISBN: 1-59593-018-3. DOI: 10.1145/1070891.1065917.

[65] J. Lawall, H. Duchesne, G. Muller, and A.-F. L. Meur. “Bossa Nova: Introducing Modularity into the Bossa Domain-Specific Language.” In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE ’05)*. 2005, pp. 78–93.

[66] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. “TinyOS: An Operating System for Wireless Sensor Networks.” In: Ambient Intelligence. Springer-Verlag, 2005.

[67] M. Mernik, J. Heering, and A. M. Sloane. “When and how to develop domain-specific languages.” In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344.

[68] G. Muller, J. L. Lawall, and H. Duchesne. “A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation.” In: *Proceedings of the 9th IEEE International Symposium on High-Assurance Systems Engineering (HASE ’05)*. IEEE Computer Society Press, 2005, pp. 56–65. ISBN: 0-7695-2377-3. DOI: 10.1109/HASE.2005.1.

[69] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17. OSEK/VDX Group, 2005.

[70] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. “Information Hiding Interfaces for Aspect-Oriented Design.” In: *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 2005, pp. 166–175. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081734.

[71] *TriCore 1 User’s Manual (V1.3.5), Volume 1: Core Architecture*. Infineon Technologies AG. 2005.

[72] D. Batory. “Feature-Oriented Programming and the AHEAD Tool Suite.” In: *Proceedings of the 26th International Conference on Software Engineering (ICSE ’04)*. IEEE Computer Society Press, 2004, pp. 702–703.

[73] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. “Variability management with feature models.” In: *Science of Computer Programming* 53.3 (2004), pp. 333–352. ISSN: 0167-6423. DOI: 10.1016/j.scico.2003.04.005.

[74] C. Constantinides, T. Skotiniotis, and M. Störzer. “AOP Considered Harmful.” In: *1st European Interactive Workshop on Aspects in Software (EIWAS ’04)*. 2004.

[75] C. L. Conway and S. A. Edwards. “NDL: A Domain-Specific Language for Device Drivers.” In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’04)*. ACM Press, 2004, pp. 30–36. ISBN: 1-58113-806-7. DOI: 10.1145/997163.997169.

[76] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2009-09-09. OSEK/VDX Group, 2004.

[77] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. “Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software.” In: *Journal of Embedded Computing* (2004).

[78] A. Tešanović, K. Sheng, and J. Hansson. “Application-Tailored Database Systems: A Case of Aspects in an Embedded Database.” In: *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS ’04)*. IEEE Computer Society Press, 2004.

[79] E. Visser. “Program Transformation with Stratego/XT.” In: *Domain-Specific Program Generation*. Vol. 3016/2004. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 216–238. ISBN: 978-3-540-22119-7. DOI: 10.1007/b98156.

[80] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. “On the Automatic Evolution of an OS Kernel Using Temporal Logic and AOP.” In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE ’03)*. IEEE Computer Society Press, 2003, pp. 196–204.

[81] C. Clifton and G. T. Leavens. *Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy*. Tech. rep. TR 03-01. http://archives.cs.iastate.edu/documents/disk0/00/00/02/96/00000296-00/paper.pdf. Department of Computer Science, Iowa State University, 2003.

[82] Y. Coady and G. Kiczales. “Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code.” In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD ’03)*. ACM Press, 2003, pp. 50–59.

[83] A. Dunkels. “Full TCP/IP for 8-bit architectures.” In: *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys ’03)*. ACM Press. 2003, pp. 85–98.

[84] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. “The nesC language: A holistic approach to networked embedded systems.” In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’03)*. ACM Press, 2003, pp. 1–11. ISBN: 1-58113-662-5.

[85] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. “XVCL: XML-based variant configuration language.” In: *Proceedings of the 25th International Conference on Software Engineering (ICSE ’03)*. IEEE Computer Society Press, 2003, pp. 810–811. ISBN: 0-7695-1877-X.

[86] M. Barbeau and F. Bordeleau. “A Protocol Stack Development Tool Using Generative Programming.” In: *Proceedings of the 1st International Conference on Generative Programming and Component Engineering (GPCE ’02)*. Lecture Notes in Computer Science. Springer-Verlag, 2002, pp. 93–109. ISBN: 978-3-540-44284-4. DOI: 10.1007/3-540-45821-2_6. URL: http://dx.doi.org/10.1007/3-540-45821-2_6.

[87] L. P. Barreto and G. Muller. “Bossa: a Language-based Approach to the Design of Real-time Schedulers.” In: *10th International Conference on Real-Time Systems (RTS ’02)*. 2002, pp. 19–31.

[88] M. D. Ernst, G. J. Badros, and D. Notkin. “An Empirical Analysis of C Preprocessor Use.” In: *IEEE Transactions on Software Engineering* 28.12 (2002), pp. 1146–1170. ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1158288.

[89] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. “THINK: A Software Framework for Component-based Operating System Kernels.” In: *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX Association, 2002, pp. 73–86.

[90] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. “The JX Operating System.” In: *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX Association, 2002, pp. 45–58. ISBN: 1-880446-00-6.

[91] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. “An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family.” In: *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems (ECOOP-OOOS ’02)*. 2002, pp. 49–54. ISBN: 84-699-8733-X.

[92] A. Massa. *Embedded Software Development with eCos*. New Riders, 2002. ISBN: 978-0130354730.

[93] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001. ISBN: 0-20-17043-15.

[94] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. “Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code.” In: *Proceedings of the 3rd Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE ’01)*. 2001.

[95] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. “Discussing aspects of AOP.” In: *Communications of the ACM* 44.10 (2001), pp. 33–38.

[96] A. Fröhlich. *Application-Oriented Operating Systems*. GMD Research Series 17. GMD - Forschungszentrum Informationstechnik, 2001.

[97] L. Northrop and P. Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN: 978-0-201-70332-0.

[98] OSEK/VDX Group. *Time-Triggered Operating System Specification 1.0*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`. OSEK/VDX Group, 2001.

[99] H. Ossher and P. Tarr. “Using Multidimensional Separation of Concerns to (Re)shape Evolving Software.” In: *Communications of the ACM* (2001), pp. 43–50.

[100] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, 2000. ISBN: 0-20-13097-77.

[101] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. “Generative Programming and Active Libraries.” In: *Generic Programming*. Vol. 1766. Lecture Notes in Computer Science. Springer-Verlag, 2000, pp. 25–39. ISBN: 978-3-540-41090-4. DOI: `10.1007/3-540-39953-4_3`. URL: `http://dx.doi.org/10.1007/3-540-39953-4_3`.

[102] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. “Stub-code performance is becoming important.” In: *Proceedings of the 1st conference on Industrial Experiences with Systems Software - Volume 1*. USENIX Association, 2000, 4:1–4:8.

[103] Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe. “C/C++ Conditional Compilation Analysis Using Symbolic Execution.” In: *Proceedings of the 16th IEEE International Conference on Software Maintainance (ICSM’00)*. IEEE Computer Society Press, 2000, p. 196. ISBN: 0-7695-0753-0.

[104] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. “Devil: an IDL for hardware programming.” In: *4th Symposium on Operating System Design and Implementation (OSDI ’00)*. USENIX Association, 2000, pp. 17–30.

[105] G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Mérillon, and L. Réveillère. “Towards robust OSes for appliances: a new approach based on domain-specific languages.” In: *Proceedings of the 9th ACM SIGOPS European Workshop “Beyond the PC: New Challenges for the Operating System”*. ACM Press, 2000, pp. 19–24. DOI: `10.1145/566726.566732`.

[106] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. “The Koala Component Model for Consumer Electronics Software.” In: *Computer* 33.3 (2000), pp. 78–85. ISSN: 0018-9162. DOI: `10.1109/2.825699`.

[107] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. “Knit: Component Composition for Systems Software.” In: *4th Symposium on Operating System Design and Implementation (OSDI ’00)*. USENIX Association, 2000, pp. 347–360.

[108] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. “A DSL Approach to Improve Productivity and Safety in Device Drivers Development.” In: *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE ’00)*. IEEE Computer Society Press, 2000, pp. 101–110. ISBN: 0-7695-0710-7.

[109] N. Wells. “BusyBox: A Swiss Army Knife for Linux.” In: *Linux Journal* 10 (78es 2000).

[110] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. “The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems.” In: *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC ’99)*. IEEE Computer Society Press, 1999, pp. 45–53.

[111] S. Chandra, B. Richards, and J. R. Larus. “Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols.” In: *IEEE Transactions on Software Engineering* 25.3 (1999), pp. 317–333. ISSN: 0098-5589. DOI: 10.1109/32.798322.

[112] A. Fröhlich and W. Schröder-Preikschat. “High Performance Application-oriented Operating Systems – the EPOS Aproach.” In: *Proceedings of the 11th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD ’99)*. IEEE Computer Society Press, 1999, pp. 3–9.

[113] D. B. Stewart. “Twenty-Five Most Common Mistakes with Real-Time Software Development.” In: *Proceedings of the 1999 Embedded Systems Conference (ESC ’99)*. 1999.

[114] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. “N Degrees of Separation: Multi-Dimensional Separation of Concerns.” In: *Proceedings of the 21st International Conference on Software Engineering (ICSE ’99)*. IEEE Computer Society Press, 1999, pp. 107–119.

[115] S. Thibault, R. Marlet, and C. Consel. “Domain-specific languages: from design to implementation application to video device drivers generation.” In: *IEEE Transactions on Software Engineering* 25.3 (1999), pp. 363–377. ISSN: 0098-5589. DOI: 10.1109/32.798325.

[116] D. Box. *Essential COM*. Addison-Wesley, 1998. ISBN: 0-201-63446-5.

[117] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-.-.-. N. Volanschi. “Tempo: Specializing Systems Applications and Beyond.” In: *ACM Computing Surveys* 30.3es (1998).

[118] M. Hohmuth. *The Fiasco kernel: System architecture*. Technical report. TU Dresden, 1998.

[119] S. Thibault, C. Consel, and G. Muller. “Safe and Efficient Active Network Programming.” In: *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS ’98)*. IEEE Computer Society Press, 1998, pp. 135–143. ISBN: 0-8186-9218-9. DOI: 10.1109/RELDIS.1998.740484.

[120] T. L. Veldhuizen and D. Gannon. “Active Libraries: Rethinking the Roles of Compilers and Libraries.” In: *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*. SIAM Press, 1998.

[121] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. “Flick: a flexible, optimizing IDL compiler.” In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’97)*. ACM Press, 1997, pp. 44–56. ISBN: 0-89791-907-6. DOI: 10.1145/258915.258921.

[122] J.-M. Favre. “A Rigorous Approach to Support the Maintenance of Large Portable Software.” In: *1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR ’97)*. IEEE Computer Society Press, 1997, p. 44. ISBN: 0-8186-7892-5. DOI: 10.1109/CSMR.1997.583003.

[123] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. “The Flux OSKit: A Substrate for Kernel and Language Research.” In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP ’97)*. ACM SIGOPS Operating Systems Review. ACM Press, 1997, pp. 38–51.

[124] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. “The Performance of μ-Kernel-Based Systems.” In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP ’97)*. ACM Press, 1997. DOI: 10.1145/269005.266660.

[125] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. “Aspect-Oriented Programming.” In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ’97)*. Vol. 1241. Lecture Notes in Computer Science. Springer-Verlag, 1997, pp. 220–242.

[126] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[127] K. Driesen and U. Hölzle. “The Direct Cost of Virtual Function Calls in C++.” In: *Proceedings of the 11th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’96)*. 1996.

[128] 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: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95)*. ACM SIGOPS Operating Systems Review. ACM Press, 1995, pp. 267–283. DOI: 10.1145/224056.224077.

[129] D. R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management." In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. ACM Press, 1995, pp. 251–266. DOI: 10.1145/224057.224076.

[130] J. Liedtke. "On μ-Kernel Construction." In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. ACM Press, 1995. DOI: 10.1145/224057.224075.

[131] M. A. Simos. "Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle." In: *Proceedings of the 1995 Symposium on Software Reusability (SSR '95)*. ACM Press, 1995, pp. 196–205. ISBN: 0-89791-739-1. DOI: 10.1145/211782.211845.

[132] T. Veldhuizen. "Template Metaprograms." In: *C++ Report* (1995).

[133] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. "SPIN — An Extensible Microkernel for Application-specific Operating System Services." In: *ACM SIGOPS European Workshop*. 1994, pp. 68–71. URL: http://citeseer.ist.psu.edu/article/chambers94spin.html.

[134] R. Campbell, N. Islam, P. Madany, and D. Raila. "Designing and Implementing Choices: An Object-Oriented System in C++." In: *Communications of the ACM* 36.9 (1993), pp. 117–126. DOI: 10.1145/162685.162717.

[135] J. Liedtke. "Improving IPC by Kernel Design." In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM Press, 1993. ISBN: 0-89791-632-8. DOI: 10.1145/168619.168633.

[136] H. Spencer and G. Collyer. "#ifdef Considered Harmful, or Portability Experience With C News." In: *Proceedings of the 1992 USENIX Annual Technical Conference*. USENIX Association, 1992.

[137] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. ACM Press/Addison-Wesley Publishing Co., 1992. ISBN: 0-201-54428-8.

[138] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Carnegie Mellon University, Software Engineering Institute, 1990.

[139] P. Freeman. "A conceptual analysis of the Draco approach to constructing software systems." In: *IEEE Transactions on Software Engineering* 13.7 (1987), pp. 830–844. ISSN: 0098-5589.

[140] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. "MACH: A New Kernel Foundation for UNIX Development." In: *Proceedings of the USENIX Summer Conference*. USENIX Association, 1986, pp. 93–113.

[141] B. W. Lampson. "Hints for Computer System Design." In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP '83)*. ACM Press, 1983, pp. 33–48. ISBN: 0-89791-115-6. DOI: 10.1145/800217.806614.

[142] D. L. Parnas. "Designing Software for Ease of Extension and Contraction." In: *IEEE Transactions on Software Engineering* SE-5.2 (1979), pp. 128–138.

[143] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall PTR, 1978.

[144] A. N. Habermann, L. Flon, and L. W. Cooprider. "Modularization and Hierarchy in a Family of Operating Systems." In: *Communications of the ACM* 19.5 (1976), pp. 266–272.

[145] D. L. Parnas. "On the Design and Development of Program Families." In: *IEEE Transactions on Software Engineering* SE-2.1 (1976), pp. 1–9.

[146] D. L. Parnas. *Some Hypothesis About the "Uses" Hierarchy for Operating Systems*. Tech. rep. TH Darmstadt, Fachbereich Informatik, 1976.

[147] D. L. Parnas. "On the Criteria to be used in Decomposing Systems into Modules." In: *Communications of the ACM* (1972), pp. 1053–1058.

[148] E. W. Dijkstra. “The Structure of the THE-Multiprogramming System.” In: *Communications of the ACM* 11.5 (1968), pp. 341–346.

A.2. Personal Bibliography (90)

All journal, conference, and workshop papers listed in the following were selected for publication by international program committees in a formal review process. They have been published in printed journals, proceedings, or widely recognized digital libraries (ACM, IEEE, USENIX).

Journal Papers (9)

[J1] D. Lohmann, O. Spinczyk, W. Hofer, and W. Schröder-Preikschat. "The Aspect-Aware Design and Implementation of the CiAO Operating-System Family." In: *Transactions on AOSD IX*. Lecture Notes in Computer Science 7271. Springer-Verlag, 2012, pp. 168–215. DOI: 10.1007/978-3-642-35551-6_5.

[J2] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. "Configuration Coverage in the Analysis of Large-Scale System Software." In: *ACM SIGOPS Operating Systems Review* 45.3 (2012), pp. 10–14. ISSN: 0163-5980. DOI: 10.1145/2094091.2094095.

[J3] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann. "Revealing and Repairing Configuration Inconsistencies in Large-Scale System Software." In: *International Journal on Software Tools for Technology Transfer (STTT)* 14.5 (2012), pp. 531–551. DOI: 10.1007/s10009-012-0225-2.

[J4] M. Urban, D. Lohmann, and O. Spinczyk. "PUMA: An Aspect-Oriented Code Analysis and Manipulation Framework for C and C++." In: *Transactions on AOSD VIII*. Lecture Notes in Computer Science 6580. Springer-Verlag, 2011, pp. 141–162. DOI: 10.1007/978-3-642-22031-9_5.

[J5] R. Tartler, D. Lohmann, F. Scheler, and O. Spinczyk. "AspectC++: An integrated approach for static and dynamic adaptation of system software." In: *Knowledge-Based Systems* 2010.23 (2010), pp. 704–720. DOI: 10.1016/j.knosys.2010.03.002.

[J6] W. Schröder-Preikschat, D. Lohmann, F. Scheler, and O. Spinczyk. "Dimensions of Variability in Embedded Operating Systems." In: *Informatik - Forschung und Entwicklung* 22.1 (2007), pp. 5–22. DOI: 10.1007/s00450-007-0037-x.

[J7] O. Spinczyk and D. Lohmann. "The Design and Implementation of AspectC++." In: *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20.7 (2007), pp. 636–651. DOI: 10.1016/j.knosys.2007.05.004.

[J8] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. "Lean and Efficient System Software Product Lines: Where Aspects Beat Objects." In: *Transactions on AOSD II*. Lecture Notes in Computer Science 4242. Springer-Verlag, 2006, pp. 227–255. DOI: 10.1007/11922827_8.

[J9] O. Spinczyk, D. Lohmann, and M. Urban. "AspectC++: An AOP Extension for C++." In: *Software Developers Journal* 5 (2005), pp. 68–76. URL: http://www.aspectc.org/fileadmin/publications/sdj-2005-en.pdf.

Conference Papers (29)

[C1] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring." In: *Proceedings of the 20th Network and Distributed Systems Security Symposium*. (Feb. 24–27, 2013). The Internet Society, 2013. URL: http://www.internetsociety.org/sites/default/files/03_2_0.pdf.

[C2] S. Nadi, C. Dietrich, R. Tartler, R. Holt, and D. Lohmann. "Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?" In: *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. (To appear). IEEE Computer Society Press, 2013.

[C3] J. Paul, W. Stechele, M. Kröhnert, T. Asfour, B. Oechslein, C. Erhardt, J. Schedel, D. Lohmann, and W. Schröder-Preikschat. “A Resource-Aware Nearest Neighbor Search Algorithm for K-Dimensional Trees.” In: *Proceedings of the 2013 Conference on Design & Architectures for Signal and Image Processing (DASIP ’13)*. IEEE Computer Society Press, 2013, pp. 80–87. ISBN: 979-10-92279-01-6.

[C4] I. Stilkerich, M. Strotz, C. Erhardt, M. Hoffmann, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. “A JVM for Soft-Error-Prone Embedded Systems.” In: *Proceedings of the 2013 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’13)*. ACM Press, 2013, pp. 21–32. ISBN: 978-1-4503-2085-6. DOI: 10.1145/2465554.2465571.

[C5⋆] C. Borchert, D. Lohmann, and O. Spinczyk. “CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack.” In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys ’12)*. ACM Press, 2012, pp. 435–448. ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307676.

[C6⋆] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. “A Robust Approach for Variability Extraction from the Linux Build System.” In: *Proceedings of the 16th Software Product Line Conference (SPLC ’12)*. (Sept. 2–7, 2012). ACM Press, 2012, pp. 21–30. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362544.

[C7⋆] W. Hofer, D. Danner, R. Müller, F. Scheler, W. Schröder-Preikschat, and D. Lohmann. “Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS.” In: *Proceedings of the 33rd IEEE International Symposium on Real-Time Systems (RTSS ’12)*. (Dec. 4–7, 2012). IEEE Computer Society Press, 2012, pp. 237–247. ISBN: 978-0-7695-4869-2. DOI: 10.1109/RTSS.2012.75.

[C8] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schröder-Preikschat, and R. Schmid. “Eliminating Single Points of Failure in Software-Based Redundancy.” In: *Proceedings of the 9th European Dependable Computing Conference (EDCC ’12)*. IEEE Computer Society Press, 2012, pp. 49–60. ISBN: 978-1-4673-0938-7. DOI: 10.1109/EDCC.2012.21.

[C9] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Härtig, L. Hedrich, A. Herkersdorf, R. Kapitza, D. Lohmann, P. Marwedel, M. Platzner, W. Rosenstiel, U. Schlichtmann, O. Spinczyk, M. Tahoori, J. Teich, N. Wehn, and H.-J. Wunderlich. “Design and Architectures for Dependable Embedded Systems.” In: *Proceedings of the 9th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS ’11)*. ACM Press, 2011, pp. 69–78. ISBN: 978-1-4503-0715-4. DOI: 10.1145/2039370.2039384.

[C10⋆] W. Hofer, D. Lohmann, and W. Schröder-Preikschat. “Sleepy Sloth: Threads as Interrupts as Threads.” In: *Proceedings of the 32nd IEEE International Symposium on Real-Time Systems (RTSS ’11)*. (Nov. 29–Dec. 2, 2011). IEEE Computer Society Press, 2011, pp. 67–77. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.14.

[C11] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel. “DistRM: Distributed Resource Management for On-chip Many-Core Systems.” In: *Proceedings of the 9th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS ’11)*. ACM Press, 2011, pp. 119–128. ISBN: 978-1-4503-0715-4. DOI: 10.1145/2039370.2039392.

[C12⋆] D. Lohmann, W. Hofer, W. Schröder-Preikschat, and O. Spinczyk. “Aspect-Aware Operating-System Development.” In: *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD ’11)*. ACM Press, 2011, pp. 69–80. ISBN: 978-1-4503-0605-8. DOI: 10.1145/1960275.1960285.

[C13] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schröder-Preikschat, and D. Lohmann. “Escaping the Bonds of the Legacy: Step-Wise Migration to a Type-Safe Language in Safety-Critical Embedded Systems.” In: *Proceedings of the 14th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC ’11)*. IEEE Computer Society Press, 2011, pp. 163–170. ISBN: 978-0-7695-4368-0. DOI: 10.1109/ISORC.2011.29.

[C14⋆] ■ R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem.” In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys ’11)*. ACM Press, 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966451.

[C15] C. Elsner, C. Schwanninger, W. Schröder-Preikschat, and D. Lohmann. “Multi-Level Product Line Customization.” In: *Proceedings of the 2010 Conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT ’10)*. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 37–58. ISBN: 978-1-60750-628-7. DOI: 10.3233/978-1-60750-629-4-37.

[C16] C. Elsner, P. Ulbrich, D. Lohmann, and W. Schröder-Preikschat. “Consistent Product Line Configuration Across File Type and Product Line Boundaries.” In: *Proceedings of the 14th Software Product Line Conference (SPLC ’10)*. Vol. 6287. Lecture Notes in Computer Science. Best paper award (out of 90 submissions). Springer-Verlag, 2010, pp. 181–195. ISBN: 978-3-642-15578-9. DOI: 10.1007/978-3-642-15579-6_13.

[C17] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. “Efficient Extraction and Analysis of Preprocessor-Based Variability.” In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE ’10)*. ACM Press, 2010, pp. 33–42. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868300.

[C18] M. Urban, D. Lohmann, and O. Spinczyk. “The aspect-oriented design of the PUMA C/C++ parser framework.” In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD ’10)*. ACM Press, 2010, pp. 217–221. ISBN: 978-1-60558-958-9. DOI: 10.1145/1739230.1739256.

[C19⋆] ■ W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. “Sloth: Threads as Interrupts.” In: *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS ’09)*. (Dec. 1–4, 2009). IEEE Computer Society Press, 2009, pp. 204–213. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.18.

[C20⋆] ■ D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. “CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems.” In: *Proceedings of the 2009 USENIX Annual Technical Conference*. USENIX Association, 2009, pp. 215–228. ISBN: 978-1-931971-68-3. URL: http://www.usenix.org/event/usenix09/tech/full_papers/lohmann/lohmann.pdf.

[C21] F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann. “Parallel, Hardware-Supported Interrupt Handling in an Event-Triggered Real-Time Operating System.” In: *Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES ’09)*. ACM Press, 2009, pp. 59–67. ISBN: 0-7695-2400-1. DOI: 10.1145/1629395.1629419.

[C22] R. Tartler, D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. “Dynamic AspectC++: Generic Advice at Any Time.” In: *Proceedings of the 2009 Conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT ’09)*. Frontiers in Artificial Intelligence and Applications 199. IOS Press, 2009, pp. 165–186. ISBN: 978-1-60750-049-0. DOI: 10.3233/978-1-60750-049-0-165.

[C23] C. Gibbs, D. Lohmann, R. Liu, and Y. Coady. “Modular Integration through Aspects: Making Cents of Legacy Systems.” In: *Proceedings of the 40th Hawaii International Conference on System Sciences (HICSS ’07) - Clinical Process and Data Integration and Evolution*. IEEE Computer Society Press, 2007. DOI: 10.1109/HICSS.2007.390.

[C24] W. Gilani, F. Scheler, D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. “Unification of Static and Dynamic AOP for Evolution in Embedded Software Systems.” In: *Proceedings of the Sixth International Symposium on Software Composition*. Lecture Notes in Computer Science 4829. Springer-Verlag, 2007, pp. 216–234. ISBN: 978-3-540-77350-4. DOI: 10.1007/978-3-540-77351-1.

[C25] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. “A Quantitative Analysis of Aspects in the eCos Kernel.” In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys ’06)*. ACM Press, 2006, pp. 191–204. ISBN: 1-59593-322-0. DOI: 10.1145/1218063.1217954.

[C26] W. Schröder-Preikschat, D. Lohmann, W. Gilani, F. Scheler, and O. Spinczyk. “Static and Dynamic Weaving in System Software with AspectC++.” In: *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS ’06) - Track 9*. IEEE Computer Society Press, 2006. DOI: 10.1109/HICSS.2006.437.

[C27] D. Lohmann and O. Spinczyk. “On Typesafe Aspect Implementations in C++.” In: *Proceedings of Software Composition 2005 (SC ’05)*. Vol. 3628. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 135–149. DOI: 10.1007/11550679.

[C28] O. Spinczyk, D. Lohmann, and M. Urban. “Advances in AOP with AspectC++.” In: *New Trends in Software Methodologies, Tools and Techniques (SoMeT ’05)*. Frontiers in Artificial Intelligence and Applications 129. IOS Press, 2005, pp. 33–53. ISBN: 1-58603-556-8.

[C29] D. Lohmann, G. Blaschke, and O. Spinczyk. “Generic Advice: On the Combination of AOP with Generative Programming in AspectC++.” In: *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE ’04)*. Vol. 3286. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 55–74. ISBN: 978-3-540-23580-4. DOI: 10.1007/978-3-540-30175-2_4.

Workshop Papers (35)

[W1] M. Hoffmann, C. Dietrich, and D. Lohmann. “Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience.” In: *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES ’13)*. Lecture Notes in Informatics. German Society of Informatics, 2013.

[W2] K.-B. Schultis, C. Elsner, and D. Lohmann. “Moving Towards Industrial Software Ecosystems: Are Our Software Architectures Fit for the Future?” In: *Proceedings of the 4th International Workshop on Product LinE Approaches in Software Engineering (ICSE-PLEASE ’13)*. 2013, pp. 9–12. DOI: 10.1109/PLEASE.2013.6608655.

[W3] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. “Understanding Linux Feature Distribution.” In: *Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software (AOSD-MISS ’12)*. (Mar. 27, 2012). ACM Press, 2012. ISBN: 978-1-4503-1217-2. DOI: 10.1145/2162024.2162030.

[W4] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk. “FAIL*: Towards a Versatile Fault-Injection Experiment Framework.” In: *25th International Conference on Architecture of Computing Systems (ARCS ’12), Workshop Proceedings*. Vol. 200. Lecture Notes in Informatics. Gesellschaft für Informatik, 2012, pp. 201–210. ISBN: 978-3-88579-294-9.

[W5⋆] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Doreanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann. “Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability.” In: *Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep ’12)*. USENIX Association, 2012, pp. 1–6.

[W6] C. Elsner, D. Lohmann, and W. Schröder-Preikschat. “An Infrastructure for Composing Build Systems of Software Product Lines.” In: *Proceedings of the 15th Software Product Line Conference (SPLC ’11), Volume 2*. (MAPLE/SCALE ’11 Proceedings). ACM Press, 2011, 18:1–18:8. ISBN: 978-1-4503-0789-5. DOI: 10.1145/2019136.2019157.

[W7] C. Erhardt, M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. “Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study.” In: *JTRES ’11: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM Press, 2011, pp. 96–105. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043927.

[W8] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. "OctoPOS: A Parallel Operating System for Invasive Computing." In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA'11)*. Vol. USB Proceedings. 2011, pp. 9–14. URL: http://research.microsoft.com/en-us/um/people/tharris/sfma/papers/sfma-final7.pdf.

[W9] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk. "Revisiting Fault-Injection Experiment-Platform Architectures." In: *Proceedings of the 17th International Symposium on Dependable Computing (PRDC '11)*. Fast abstract. IEEE Computer Society Press, 2011, pp. 284–285. ISBN: 978-1-4577-2005-5. DOI: 10.1109/PRDC.2011.46.

[W10] H. Schirmeier, R. Kapitza, D. Lohmann, and O. Spinczyk. "DanceOS: Towards Dependability Aspects in Configurable Embedded Operating Systems." In: *Proceedings of the 3rd HiPEAC Workshop on Design for Reliability (DFR '11)*. 2011, pp. 21–26.

[W11] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. "Configuration Coverage in the Analysis of Large-Scale System Software." In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. ACM Press, 2011, 2:1–2:5. ISBN: 978-1-4503-0979-0. DOI: 10.1145/2039239.2039242.

[W12] I. Thomm, M. Stilkerich, R. Kapitza, D. Lohmann, and W. Schröder-Preikschat. "Automated Application of Fault Tolerance Mechanisms in a Component-Based System." In: *JTRES '11: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM Press, 2011, pp. 87–95. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043925.

[W13] C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat. "Variability in Time – Product Line Variability and Evolution Revisited." In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '10)*. ICB Research Reports 37. 2010, pp. 131–138.

[W14] M. Gernoth, D. Lohmann, W. Schröder-Preikschat, J. Sincero, R. Tartler, and D. Wischermann. "Challenges in Operating-Systems Reengineering for Many Cores." In: *Proceedings of the 3rd International Workshop on Multicore Software Engineering (IWMSE '10)*. ACM Press, 2010, pp. 52–53. ISBN: 978-1-60558-964-0. DOI: 10.1145/1808954.1808968.

[W15] W. Hofer, C. Elsner, F. Blendinger, W. Schröder-Preikschat, and D. Lohmann. "Toolchain-Independent Variant Management with the Leviathan Filesystem." In: *Proceedings of the 2nd Workshop on Feature-Oriented Software Development (FOSD '10)*. ACM Press, 2010, pp. 18–24. ISBN: 978-1-4503-0208-1. DOI: 10.1145/1868688.1868692.

[W16] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. "Gradual Software-Based Memory Protection." In: *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS '10)*. ACM Press, 2010. ISBN: 978-1-4503-0120-6.

[W17] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. "Memory Protection at Option." In: *Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety*. ACM Press, 2010, pp. 17–20. ISBN: 978-1-60558-915-2. DOI: 10.1145/1772643.1772649.

[W18] C. Elsner, D. Lohmann, and W. Schröder-Preikschat. "Product Derivation for Solution-Driven Product Line Engineering." In: *Proceedings of the 1st Workshop on Feature-Oriented Software Development (FOSD '09)*. ACM Press, 2009, pp. 35–41. DOI: 10.1145/1629716.1629724.

[W19] C. Elsner, D. Lohmann, and C. Schwanninger. "Eine Infrastruktur für modellgetriebene hierarchische Produktlinien." In: *Software Engineering 2009 - Workshopband*. Vol. 150. Lecture Notes in Informatics. Gesellschaft für Informatik, 2009, pp. 107–113. ISBN: 978-3-88579-244-4.

[W20] P. Stellwag, D. Lohmann, and W. Schröder-Preikschat. "An Asynchronous Nonblocking Coordination and Synchronization Protocol for a Parallel Robotic Control Kernel." In: *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems (IIES '09)*. ACM Press, 2009, pp. 7–12. ISBN: 978-1-60558-464-5. DOI: 10.1145/1519130.1519132.

[W21] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. “Dead or Alive: Finding zombie features in the Linux kernel.” In: *Proceedings of the 1st Workshop on Feature-Oriented Software Development (FOSD ’09)*. ACM Press, 2009, pp. 81–86. ISBN: 978-1-60558-567-3. DOI: 10.1145/1629716.1629732.

[W22] C. Elsner, D. Lohmann, and W. Schröder-Preikschat. “Towards Separation of Concerns in Model Transformation Workflows.” In: *Proceedings of the 12th Software Product Line Conference (SPLC ’08), Second Volume*. Lero International Science Centre, 2008, pp. 81–88. ISBN: 978-1-905952-06-9.

[W23] W. Hofer, D. Lohmann, and W. Schröder-Preikschat. “Concern Impact Analysis in Configurable System Software—The AUTOSAR OS Case.” In: *Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’08)*. ACM Press, 2008, pp. 1–6. ISBN: 978-1-60558-142-2. DOI: 10.1145/1404891.1404897.

[W24] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat. “Configurable Memory Protection by Aspects.” In: *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS ’07)*. ACM Press, 2007, pp. 1–5. ISBN: 978-1-59593-922-7. DOI: 10.1145/1376789.1376794.

[W25] D. Lohmann, J. Streicher, O. Spinczyk, and W. Schröder-Preikschat. “Interrupt Synchronization in the CiAO Operating System.” In: *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’07)*. ACM Press, 2007. ISBN: 1-59593-657-8. DOI: 10.1145/1233901.1233907.

[W26] D. Lohmann, F. Scheler, W. Schröder-Preikschat, and O. Spinczyk. “PURE Embedded Operating Systems – CiAO.” In: *Proceedings of the ECRTS Workshop on Operating Systems Platforms for Embedded Real-Time applications (ECRTS-OSPERT ’06)*. 2006.

[W27] O. Spinczyk, D. Lohmann, and W. Schröder-Preikschat. “Concern Hierarchies.” In: *Proceedings for First Workshop on Aspect-oriented Product Line Engineering (AOPLE-1)*. COMP-004-2007. 2006.

[W28] O. Spinczyk, D. Lohmann, and W. Schröder-Preikschat. “Concern Hierarchies.” In: *1st GPCE Workshop on Aspect-Oriented Product Line Engineering (GPCE-AOPLE ’06)*. (published as Lancaster University TR: COMP-004-2007). 2006, pp. 13–19.

[W29] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. “Functional and Non-Functional Properties in a Family of Embedded Operating Systems.” In: *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS ’05)*. 2005, pp. 413–420. DOI: 10.1109/WORDS.2005.37.

[W30] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. “The Design of Application-Tailorable Operating System Product Lines.” In: *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS ’05)*. Vol. 3956. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 99–117. ISBN: 3-540-33689-3. DOI: 10.1007/11741060_6.

[W31] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. “On the Configuration of Non-Functional Properties in Operating System Product Lines.” In: *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’05)*. Northeastern University, Boston (NU-CCIS-05-03), 2005, pp. 19–25.

[W32] D. Lohmann, W. Gilani, and O. Spinczyk. “On Adapable Aspect-Oriented Operating Systems.” In: *Proceedings of the 2004 ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS ’04)*. 2004, pp. 47–52.

[W33] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. “On the Design and Development of a Customizable Embedded Operating System.” In: *Proceedings of the SRDS Workshop on Dependable Embedded Systems (SRDS-DES ’04)*. IEEE Computer Society Press, 2004, pp. 1–6.

[W34] O. Spinczyk and D. Lohmann. “Using AOP to Develop Architecture-Neutral Operating System Components.” In: *Proceedings of the 11th ACM SIGOPS European Workshop*. ACM Press, 2004, pp. 188–192. DOI: 10.1145/1133572.1133582.

[W35] D. Lohmann and J. Ebert. “A Generalization of the Hyperspace Approach using Meta-Models.” In: *Proceedings of the 2003 AOSD Early Aspects Workshop (AOSD-EA ’03)*. 2003.

Book Chapters (1)

[B1] W. Hofer, J. Sincero, D. Lohmann, and W. Schröder-Preikschat. “Configuration of Non-Functional Properties in Embedded Operating Systems: The CiAO Approach.” In: Methodologies for Non-Functional Requirements in Service Oriented Architecture. IGI Global, 2011. Chap. 5, pp. 84–103. DOI: 10.4018/978-1-60960-493-6.

Other Peer-Reviewed Contributions (8)

[O1] W. Hofer, C. Elsner, F. Blendinger, W. Schröder-Preikschat, and D. Lohmann. “Leviathan: SPL Support on Filesystem Level.” In: *Proceedings of the 14th Software Product Line Conference (SPLC ’10)*. Vol. 6287. Lecture Notes in Computer Science. Poster. Springer-Verlag, 2010, pp. 491–491. ISBN: 978-3-642-15578-9. DOI: 10.1007/978-3-642-15579-6_43.

[O2] W. Hofer, C. Elsner, F. Blendinger, W. Schröder-Preikschat, and D. Lohmann. *Leviathan: Taming the #ifdef Beast in Linux et al.* 9th USENIX Symposium on Operating System Design and Implementation (OSDI ’10). Poster. 2010.

[O3] J. Sincero, R. Tartler, C. Egger, W. Schröder-Preikschat, and D. Lohmann. *Facing the Linux 8000 Feature Nightmare*. ACM SIGOPS/EuroSys European Conference on Computer Systems 2009 (EuroSys ’09). Talk & Poster. 2010.

[O4] R. Tartler, J. Sincero, C. Egger, W. Schröder-Preikschat, and D. Lohmann. *Configurability Bugs in Linux: The 10000 Feature Challenge*. 9th USENIX Symposium on Operating System Design and Implementation (OSDI ’10). Poster. 2010.

[O5] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. *Sloth: Let the Hardware Do the Work*. 22nd ACM Symposium on Operating Systems Principles (SOSP ’09). WiP Talk. 2009.

[O6] C. Elsner and D. Lohmann. *Inter Product-Line-Reuse by Product-Line Families*. 7th International Conference on Aspect-Oriented Software Development (AOSD ’08). Poster. 2008.

[O7] F. Scheler, D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. “Aspect-Oriented Real-Time Architecture—AORTA.” In: *Proceedings of the 27th IEEE International Symposium on Real-Time Systems (RTSS ’06)*. Work-in-Progress Session. IEEE Computer Society Press, 2006, pp. 5–8.

[O8] D. Lohmann and O. Spinczyk. *Architecture-Neutral Operating System Components*. 19th ACM Symposium on Operating Systems Principles (SOSP ’03). WiP Talk. 2003.

Theses (2)

[T1] D. Lohmann. “Aspect Awareness in the Development of Configurable System Software.” PhD thesis. Friedrich-Alexander University Erlangen-Nuremberg, 2009. URL: http://www.opus.ub.uni-erlangen.de/opus/volltexte/2009/1328/pdf/LohmannDissertation.pdf.

[T2] D. Lohmann. “Multidimensionales Trennen der Belange im Softwareentwurf.” Diplomarbeit. Universität Koblenz-Landau, 2002. URL: http://www4.informatik.uni-erlangen.de/~lohmann/download/Daniel-Lohmann_Diplomarbeit.pdf.

Technical Reports (2)

[R1] J. Sincero, R. Tartler, and D. Lohmann. *An Algorithm for Quantifying the Program Variability Induced by Conditional Compilation.* Tech. rep. CS-2010-02. University of Erlangen, Department of Computer Science, 2010.

[R2] O. Spinczyk and D. Lohmann. *AspectC++ Quick Reference (Version 1.11)*. pure::systems GmbH. 2006. URL: http://www.aspectc.org/fileadmin/documentation/ac-quickref.pdf.

Workshop Readers and Proceedings (4)

[P1] *Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software (AOSD-MISS '12)*. (Mar. 27, 2012). ACM Press, 2012. ISBN: 978-1-4503-1217-2.

[P2] *Proceedings of the 9th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '10)*. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam 33. Universitätsverlag Potsdam, 2010. ISBN: 978-3-86956-043-4. URL: http://pub.ub.uni-potsdam.de/volltexte/2010/4122/.

[P3] *Proceedings of the 8th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '09)*. ACM Press, 2009. ISBN: 978-1-60558-450-8.

[P4] *Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)*. ACM Press, 2008. ISBN: 978-1-60558-142-2.

B. Paper Reprints

Out of my 82 peer-reviewed publications (see Appendix A.2 on pp. 51ff), the following 9 key contributions of my research are the main part of this cumulative habilitation treatise. They are provided as personal reprints here, all copyrights remain with the respective publishers (ACM, IEEE, USENIX).

CiAO

USENIX '09 pp 61 ff	Lohmann, Hofer, Schröder-Preikschat, Streicher, and Spinczyk. "CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems" (Acceptance rate: 16%)	[C20⋆]
AOSD '11 pp 75 ff	Lohmann, Hofer, Schröder-Preikschat, and Spinczyk. "Aspect-Aware Operating-System Development" (Acceptance rate: 23%)	[C12⋆]
MobiSys '12 pp 87 ff	Borchert, Lohmann, and Spinczyk. "CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack" (Acceptance rate: 18%)	[C5⋆]

SLOTH

RTSS '09 pp 101 ff	Hofer, Lohmann, Scheler, and Schröder-Preikschat. "Sloth: Threads as Interrupts" (Acceptance rate: 21%)	[C19⋆]
RTSS '11 pp 111 ff	Hofer, Lohmann, and Schröder-Preikschat. "Sleepy Sloth: Threads as Interrupts as Threads" (Acceptance rate: 21%)	[C10⋆]
RTSS '12 pp 123 ff	Hofer, Danner, Müller, Scheler, Schröder-Preikschat, and Lohmann. "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS" (Acceptance rate: 22%)	[C7⋆]

VAMOS

EuroSys '11 pp 135 ff	Tartler, Lohmann, Sincero, and Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem" (Acceptance rate: 15%)	[C14⋆]
SPLC '12 pp 149 ff	Dietrich, Tartler, Schröder-Preikschat, and Lohmann. "A Robust Approach for Variability Extraction from the Linux Build System" (Acceptance rate: 33%)	[C6⋆]
HotDep '12 pp 159 ff	Tartler, Kurmus, Heinloth, Rothberg, Ruprecht, Doreanu, Kapitza, Schröder-Preikschat, and Lohmann. "Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability" (Acceptance rate: 42%)	[W5⋆]

CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems*

Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat
{lohmann, hofer, wosch}@cs.fau.de
FAU Erlangen–Nuremberg

Jochen Streicher, Olaf Spinczyk
{jochen.streicher, olaf.spinczyk}@tu-dortmund.de
TU Dortmund

Abstract

This paper evaluates aspect-oriented programming (AOP) as a first-class concept for implementing configurability in system software for resource-constrained embedded systems. To compete against proprietary special-purpose solutions, system software for this domain has to be highly configurable. Such fine-grained configurability is usually implemented "in-line" by means of the C preprocessor. However, this approach does not scale – it quickly leads to "#ifdef hell" and a bad separation of concerns. At the same time, the challenges of configurability are still increasing. AUTOSAR OS, the state-of-the-art operating-system standard from the domain of automotive embedded systems, requires configurability of even fundamental architectural system policies.

On the example of our CiAO operating-system family and the AUTOSAR-OS standard, we demonstrate that AOP – if applied from the very beginning – is a profound answer to these challenges. Our results show that a well-directed, pragmatic application of AOP leads to a much better separation of concerns than does #ifdef-based configuration – without compromising on resource consumption. The suggested approach of aspect-aware operating-system development facilitates providing even fundamental system policies as configurable features.

1 Introduction

The design and implementation of operating systems has always been challenging. Besides the sheer size and the inherent asynchronous and concurrent nature of operating-system code, developers have to deal with lots of crucial nonfunctional requirements such as performance, reliability, and maintainability. Therefore, researchers have always tried to exploit the latest advances in programming languages and software engineering (such as object orientation [6], meta-object protocols [26], or virtual execution environments [14]) in order to reduce the complexity of operating system development and to improve the systems' nonfunctional properties.

1.1 Operating Systems for Small Embedded Systems

This paper focuses on small ("deeply") embedded systems. More than 98 percent of the worldwide annual production of microprocessors ends up in small embedded systems [24] – typically employed in products such as cars, appliances, or toys. Such embedded systems are subject to an enormous hardware-cost pressure. System software for this domain has to cope not only with strict resource constraints, but especially with a broad *variety* of application requirements and platforms. So to allow for reuse, an operating system for the embedded-systems domain has to be developed as a system-software product line that is highly configurable and tailorable. Furthermore, resource-saving static configuration mechanisms are strongly favored over dynamic (re-)configuration.

A good example for this class of highly configurable systems with small footprint is the new embedded operating-system standard specified by AUTOSAR, a consortium founded by all major players in the automotive industry [3]. The goal of AUTOSAR is to continue the success story of the OSEK-OS specification [19]. OSEK-compliant operating systems have been used in almost all European cars over the past ten years, which led to an enormous productivity gain in automotive software development. AUTOSAR extends the OSEK-OS specification in order to cover the whole system-software stack including communication services and a middleware layer.

Even in this restricted domain, there is already a huge variety of application requirements on operating systems. For instance, power-train applications are typically safety-critical and have to deal with real-time requirements,

*This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4, SCHR 603/7-1, and SP 968/2-1.

```
1  Cyg_Mutex::Cyg_Mutex() {
2    CYG_REPORT_FUNCTION();
3    locked      = false;
4    owner       = NULL;
5  #if defined(CYGSEM_PRI_INVERSION_PROTO_DEFAULT) && \
6      defined(CYGSEM_PRI_INVERSION_PROTO_DYNAMIC)
7  #ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_INHERIT
8    protocol    = INHERIT;
9  #endif
10 #ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_CEILING
11   protocol    = CEILING;
12   ceiling     = CYGSEM_PRI_INVERSION_PROTO_DEFAULT_PRI;
13 #endif
14 #ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_NONE
15   protocol    = NONE;
16 #endif
17 #else // not (DYNAMIC and DEFAULT defined)
18 #ifdef CYGSEM_PRI_INVERSION_PROTO_CEILING
19 #ifdef CYGSEM_DEFAULT_PRIORITY
20   ceiling = CYGSEM_DEFAULT_PRIORITY;
21 #else
22   ceiling = 0;   // Otherwise set it to zero.
23 #endif
24 #endif
25 #endif // DYNAMIC and DEFAULT defined
26   CYG_REPORT_RETURN();
27 }
```

Figure 1: "#ifdef hell" example from eCos [18]

while car body systems are far less critical. Hardware platforms range from 8-bit to 32-bit systems. Some applications require a task model with synchronization and communication primitives, whereas others are much simpler control loops. In order to reduce the number of electronic control units (up to 100 in modern cars [5]), some manufacturers have the requirement to run multiple applications on the same unit, which is only possible with guaranteed isolation; others do not have this requirement. To fulfill all these varying requirements, the AUTOSAR-OS specification [2] describes a family of systems defined by so-called *scalability classes*. It not only requires configurability of simple functional features, but also of all *policies* regarding temporal and spatial isolation. To achieve this within a single kernel implementation is challenging. The decision about fundamental operating-system policies (like the question if and how address-space protection boundaries should be enforced) is traditionally made in the early phases of operating-system development and is deeply reflected in its architecture, which in turn has an impact on many other parts of the kernel implementation. In AUTOSAR-OS systems, these decisions have to be postponed until the application developer configures the operating system.

1.2 The Price of Configurability

In a previous paper [17], we analyzed the implementation of static configurability in the popular eCos operating system [18], which also targets small embedded systems. The system implements configurability in the familiar way with #ifdef-based conditional compilation (in C++). Even though eCos does not support configurability of architectural concerns as required by AUTOSAR (such as the memory or timing protection model), we have found an "#ifdef hell", which illustrates that these techniques do not scale well enough. Maintainability and evolvability of the implementation suffer significantly. As an example, Figure 1 shows the "#ifdef hell" in the constructor of the eCos mutex class, caused by just four variants of the optional protcol for the prevention of priority inversion. However, the configurability of this protocol does not only affect the constructor code – a total of 34 #ifdef-blocks is spread over 17 functions and data structures in four implementation files.

As a solution, we proposed aspect-oriented programming (AOP) [15] and analyzed the code size and performance impact of applying AOP to factor out the scattered implementation of configurable eCos features into distinct modules called aspects.

1.3 Aspect-Oriented Programming

AOP describes a programming paradigm especially designed to tackle the implementation of *crosscutting concerns* – concerns that, even though conceptually distinct, overlap with the implementation of other concerns in the code by sharing the same functions or classes, such as the mutex configuration options in eCos.

In AOP, *aspects* encapsulate pieces of code called *advice* that implement a crosscutting concern as a distinct module. A piece of advice targets a number of *join points* (points in the static program structure or in the dynamic execution flow) described by a predicate called *pointcut expression*. Pointcut expressions are evaluated by the *aspect weaver*, which weaves the code from the advice bodies to the join points that are matched by the respective predicates.

As pointcuts are described declaratively, the target code itself does not have to be prepared or instrumented to be affected by aspects. Furthermore, the same aspect can affect various and even unforeseen parts of the target code. In the AOP literature [10], this is frequently referred to as the *obliviousness* and *quantification* properties of AOP.

The AOP language and weaver used in the eCos study and in the development of CiAO is AspectC++ [22], a source-to-source weaver that transforms AspectC++ sources to ISO C++ code, which can then be compiled by any standard-compliant C++ compiler.

Figure 2 illustrates the syntax of aspects written in AspectC++. The (excerpted) aspect `Priority_Ceiling` implements the priority ceiling variant of the eCos mutex class. For this purpose, it *introduces* a *slice* of additional elements (the member variable `ceiling`) into the

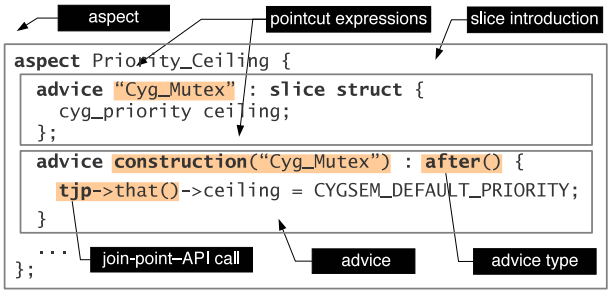


Figure 2: Syntactical elements of an aspect

class `Cyg_Mutex` and gives a piece of *advice* to initialize `ceiling` *after* each *construction* of a `Cyg_Mutex` instance. The targets of the introduction and the piece of construction advice are given by *pointcut expressions*.

In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. The match expression `"Cyg_Mutex"`, for instance, returns a pointcut containing just the class `Cyg_Mutex`. Match expressions can also be fed into pointcut functions to yield pointcuts that represent events in the control flow of the running program, such as the event where some function is about to be *called* (`call()` advice) or an object instance is about to be *constructed* (see `construction("Cyg_Mutex")` in Figure 2). In most cases, the join points for a given pointcut can be derived statically by the aspect weaver so that the respective advice is also inserted statically at compile time without any run-time overhead.

The construction pointcut in the example is used to specify some *after* advice – that is, additional behavior to be triggered after the event occurrence. Other types of advice include *before* advice (speaks for itself) and *around* advice (replaces the original behavior associated with the event occurrence).

Inside the advice body, the type and pointer `JoinPoint *tjp` provide an interface to the event context. The aspect developer can use this *join-point API* to retrieve (and partly modify) contextual information associated with the event, such as the arguments or return value of the intercepted function call (`tjp->arg(`i`)`, `tjp->result()`). The `tjp->that()` in Figure 2 returns the `this` pointer of the affected object instance, which is used here to initialize the `ceiling` member variable (which in this case was introduced by the aspect itself).

1.4 Contribution and Outline

The results of applying AOP to eCos were very promising [17]. The refactored eCos system was much better structured than the original; the number of configuration points per feature could be drastically reduced. At the same time, we found that there is no negative impact on the system's performance or code size.

However, we also found that not all configurable features could be refactored into a modular aspect-oriented implementation. The main reason was that eCos did not expose enough unambiguous join points. We took this as a motivation to work on "aspect-aware operating system design". This led to the development of fundamental design principles and the implementation of the CiAO[1] OS family for evaluation purposes. The idea was to build an operating system in an aspect-oriented way from scratch, considering AOP and its mechanisms *from the very beginning* of the development process. The resulting CiAO system is *aspect-aware* in the sense that it was analyzed, designed, and implemented with AOP principles in mind. In order to avoid evaluation results biased by the eCos implementation, CiAO was newly designed after the AUTOSAR-OS standard introduced above [2].

Our main goal is to evaluate the suitability of aspect-oriented software development as a first-class concept for the design and implementation of highly configurable embedded system software product-lines. The research contributions of this work are the following:

- Deeper insights on reasons for the #ifdef hell and the value of AOP in this context (Section 2).
- Design principles for aspect-aware operating system development (Section 4).
- CiAO: The first complete implementation of an operating system kernel developed with AOP concepts[2] (Section 5).
- A discussion of our results from CiAO (Section 6) and general experiences with the approach (Section 7).

For each of the topics, there is a dedicated section in the remaining part of this paper. In addition to that, Section 3 discusses relevant related work. The paper ends with our conclusions in Section 8.

2 Problem Analysis

Why exactly do state-of-the-art configurable systems like eCos exhibit badly modularized code termed as "#ifdef hell"? Is this an inherent property of highly configurable operating systems or just a matter of implementation means? In order to examine these questions, we took a detailed look at an abstract system specification, namely the AUTOSAR-OS standard introduced in Section 1.

[1] CiAO is Aspect-Oriented

[2] The CiAO-OS family is freely available for research purposes from the authors.

	System abstractions (functional)							Callbacks		Protection facilities (architectural)						Internal	
	OS control	Tasks	ISRs category 1	ISRs category 2	Resources	Events	Alarms	Hooks	...	Timing protection	Invalid parameters	Wrong context	Interrupts disabled	Foreign OS objects	...	Preemption	...
... <3 OS services>	⊕							◑	...		◐	◐	◐	◐	...		...
`ActivateTask()`		⊕						◑	...		◐	◐	◐	◐	...	◑	...
`TerminateTask()`		⊕						◑	...			◐	◐		...	◑	...
`Schedule()`		⊕						◑	...			◐	◐		...	◑	...
... <3 more task services>		⊕						◑	...		◐	◐	◐	◐	...	◑	...
`ResumeAllInterrupts()`			⊕						...	◐		◐			...		...
`SuspendAllInterrupts()`			⊕						...	◑		◐			...		...
... <7 more ISR services>			⊕	⊕				◑	...	●	◐	◐	◐	◐	...		...
`GetResource()`					⊕			◑	...	◑	◐	◐	◐	◐	...		...
❶ `ReleaseResource()`					⊕			◑	...	◐	◐	◐	◐	◐	...	◑	...
... <4 event services>						⊕		◑	...		◐	◐	◐	◐	...	◑	...
... <6 alarm services>							⊕	◑	...		◐	◐	◐	◐	...	◑	...
... <7 schedule table services>							⊕	◑	...		◐	◐	◐	◐	...		...
... <7 OS application services>								◑	...		◐	◐	◐	◐	...		...
❷ `TaskType`		⊕			⊛	⊛			...	⊛				⊛	...	⊛	...
`ResourceType`					⊕				...					⊛	...		...
... <4 more structures>	⊕	⊛		⊕		⊛	⊕		...	⊛				⊛	...		...
System startup		◑					◑	◑	...						...		...
Task switch								●	...	●					...		...
Protection violation								◑	...						...		...
... <4 more internal points>		◑				◑		◐	...	●				◑	...	◑	...

❸

Table 1: Influence of configurable concerns (columns) on system services, system types, and internal events (rows) in AUTOSAR OS [2, 19]; kind of influence: ⊕ = extension of the API by a service or type, ⊛ = extension of an existing type, ◑ = modification after service or event, ◐ = modification before, ● = modification before *and* after

2.1 Why #ifdef Hell Appears to Be Unavoidable

The AUTOSAR-OS standard proposes a set of *scalability classes* for the purpose of system tailoring. These classes are, however, relatively coarse-grained (there are only four of them) and do not clearly separate between conceptually distinct concerns. CiAO provides a much better granularity; each AUTOSAR-OS concern is represented as an individual feature in CiAO, subject to application-dependent configuration.

In order to be able to grasp all concerns and their interactions, we have developed a specialized analysis method termed *concern impact analysis* (CIA) [13]. The idea behind CIA is to consider requirement documents together with domain-expert knowledge to develop a matrix of concerns and their influences in an iterative way. In the analysis of the AUTOSAR-OS standard, CIA yielded a comprehensive matrix, which is excerpted in Table 1.

The rows show the AUTOSAR OS system services (API functions) and system abstractions (types) in groups that represent distinct features. AUTOSAR OS is a statically configured operating system with static task priorities; hence, at run time, only services that alter the status of a *task* (e.g., setting it ready or suspended) are available. *Interrupt service routines* (ISRs), in contrast, are triggered asynchronously; the corresponding system functionality allows the application to prohibit their occurrence collectively or on a per-source basis. AUTOSAR OS distinguishes between two categories of ISRs that are somewhat comparable to top halves and bottom halves in Linux: Category-1 ISRs are scheduled by the hardware only and must not interact with the kernel. Category-2 ISRs, in contrast, run under the control of the kernel and may invoke other AUTOSAR-OS services. The third type of control flows supported by the AUTOSAR-OS kernel are *hooks*. Hooks define a callback interface for applications to be notified about kernel-internal events, such as task switch events or error conditions (see Column 8 in Table 1).

Resources are the means for AUTOSAR applications to ensure mutual exclusion to synchronize concurrent access to data structures or hardware periphery. They are comparable to mutex objects in other operating systems. In order to avoid priority inversion and deadlocks, AUTOSAR prescribes a stack-based priority ceiling protocol, which adapts task priorities at run time. Hence, a task never blocks on `GetResource()`. The only way for application tasks to become blocked is by waiting for an AUTOSAR-OS *event*; another task or ISR that sets that event can unblock that task.

Alarms allow applications to take action after a specified period of time; a *schedule table* is an abstraction that encapsulates a series of alarms. Finally, tasks, ISRs, and data can be partitioned into *OS applications*, which define a spatial and temporal protection boundary to be enforced

by the operating system.

The table lists selected identified concerns of AUTOSAR OS (column headings) and how we can expect them to interact with the named entities of the specification (row headings); that is, the 44 system services (e.g., `ActivateTask()`) and the relevant system abstractions (e.g., `TaskType`) as specified in [19, 2]. Furthermore, the lower third lists how we can expect concerns to impact *system-internal* transitions, which are not visible in the system API that is specified by the standard. Table 1 thereby provides an overview of how we can expect AUTOSAR-OS concerns to crosscut with each other in the structural space (abstractions, services) and behavioral space (control flow events) of the implementation.

The comprehensive table shows that a system that is built according to that specification will *inherently* exhibit extensive crosscutting between its concern implementations, leading to code tangling (many different concerns implemented in a single implementation module) and scattering (distribution of a single concern implementation across multiple implementation artifacts). This is because services like `ReleaseResource()` (see Table 1, row ❶) and types like `TaskType` (see Table 1, row ❷), for instance, are affected by as many as nine different concerns! That means that these implementations will exhibit at least nine #ifdef blocks – in the ideal case that each concern can be encapsulated in a single block, completely independent of the other concerns (which is unrealistic, of course). In fact, there is not a single AUTOSAR-OS service that is influenced by only one concern, which means that a straight-forward implementation using the C preprocessor will have numerous #ifdefs in every implementation entity. Thus, "#ifdef hell" seems unavoidable for the class of special-purpose, tailorable operating systems.

2.2 Why AOP Is a Promising Solution

There are several properties inherent in AOP that are promising with respect to overcoming the drawbacks in #ifdef-based configuration techniques that were detailed above.

First, AOP introduces a new kind of binding between modules. In traditional programming paradigms, the caller module P (event producer) knows and *has to know* the callee module C (event consumer); that is, its name and interface (see Figure 3.a):

```
void C::callee() {
  <additional feature>
}
void P::caller() {
  ...
  C::callee(); // has to know C to bind feature
}
```

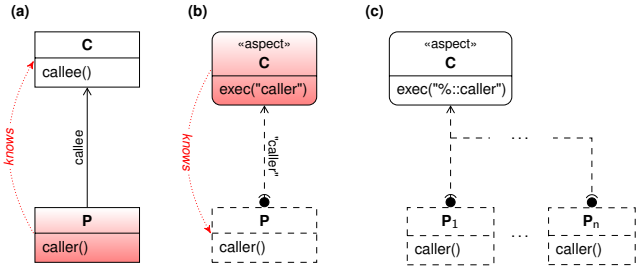


Figure 3: The mechanisms offered by AOP: advice-based binding and implicit per–join-point instantiation of advice

The advice-based binding mechanism offered by AOP can effectively invert that relationship: The callee (i.e., the aspect module C) can *integrate itself* into the caller (i.e., the base code P) without the caller having to know about the callee (see Figure 3.b):

```
advice execution("void P::caller()") : after() {
  <additional feature>
}
void P::caller() {
  ...
  // feature binds "itself"
}
```

If module C is optional and configurable, this loose coupling is an ideal mechanism for integration, because the call is implicit in the callee module. Using the traditional mechanisms, the call has to be included in the base module P and therefore has to be explicitly omitted if the feature implemented by module C is not in the current configuration. This configurable omission is realized by #ifdefs in state-of-the-art systems, bearing the significant disadvantages described above. A similar advantage of advice-based binding applies to configurable static program entities like classes or structures; aspects can integrate the state and operations needed to implement the corresponding feature into those entities *themselves* through slice introductions.

Second, by offering the mechanism of quantification through pointcut-expression matching, AOP allows for a modularized implementation of crosscutting concerns, which is also one of its main proclaimed purposes. This mechanism provides a flexible and implicit instantiation of additional implementation elements at compile-time (see Figure 3.c), ideally suited for the integration of concern implementations into configurable base code where the number of junction points (i.e., AOP join points) is flexible, ranging from zero to *n*:

```
advice execution("void %::caller()") : after() {
  <additional feature> // binds to any "caller()"
}
```

As we have seen in Table 1, most concerns in an AUTOSAR-OS implementation have a crosscutting im-

pact on many different points in the system in a similar way. An example is the policy that system services must not be called while interrupts are disabled (see Table 1, column ❸). In the requirements specification of AUTOSAR OS, this policy is defined by requirement OS093:

> *If interrupts are disabled and any OS services, excluding the interrupt services, are called outside of hook routines, then the operating system shall return* `E_OS_DISABLEDINT`. [2, p. 40]

This requirement can be translated almost "literally" to a single, modularized AspectC++ aspect:

```
aspect DisabledIntCheck {
  advice call(pcOSServices() && !pcISRServices())
  && !within(pcHooks()) : around() {
    if(interruptsDisabled())
      *tjp->result() = E_OS_DISABLEDINT;
    else
      tjp->proceed();
} };
```

For convenience and the sake of separation of concerns, the aspect uses predefined *named pointcuts*, which are defined separately from the aspects in a global header file and specify which AUTOSAR-OS service belongs to which group:

```
pointcut pcOSServices() = "% ActivateTask()" || ...
pointcut pcISRServices() = ...
...
```

Using these named pointcuts, the aspect gives advice to all points in the system where any OS service but not the interrupt services are called:

```
call(pcOSServices() && !pcISRServices()) ...
```

The resulting set of join points is further filtered to exclude all events from within a hook routine:

```
... && !within(pcHooks())
```

Thus, we eventually get all calls outside of hook routines that are made to any service that is not an ISR service. The piece of around advice given to these join points performs a test whether the interrupts are currently disabled: If positive, the return code is set to the prescribed error code and the call is aborted; if negative, the call is performed as normal. (Around advice *replaces* the original processing of the intercepted event; however, it is possible to invoke the original processing explicitly with `tjp->proceed()`.)
The complete concern is encapsulated in this single aspect. The result is an enhanced separation of concerns in the system implementation. Layered, configurable systems can especially benefit from AOP mechanisms by being able to flexibly omit parts of the system without breaking caller–callee relationships.

3 Related Work

There are several other research projects that investigate the applicability of aspects in the context of operating systems. Among the first was the α-kernel project [7], in which the evolution of four scattered OS concern implementations (namely: prefetching, disk quotas, blocking, and page daemon activation) between versions 2 and 4 of the FreeBSD kernel is analyzed retroactively. The results show that an aspect-oriented implementation would have led to significantly *better evolvability* of these concerns. Around the same time, our own group experimented with AspectC++ in the PURE OS product line and later with aspect-refactoring eCos [17]. Our results from analyzing the AspectC++ implementation of various previously hard-wired crosscutting concerns show that this new paradigm leads to *no overhead* in terms of resource consumption per se.

Not a general-purpose AOP language but an AOP-inspired language of temporal logic is used in the Bossa project to integrate the Bossa scheduler framework into the Linux kernel [1]. Another example for a special-purpose AOP-inspired language is C4 [12, 21], which is intended for the application of kernel patches in Linux. The same goal of smarter patches (with a focus on "collateral evolutions" – changes to the kernel API that have to be caught up in dozens or hundreds of device drivers) is followed by Coccinelle [20]. Although the input language for the Coccinelle engine "SmPL" is not called an AOP language, it supports the modular implementation of crosscutting kernel modifications (i.e., quantification). Other related work concentrates on dynamic aspect weaving as a means for run-time adaptation of operating-system kernels: TOSKANA provides an infrastructure for the dynamic extension of the FreeBSD kernel by aspects [9]; KLASY is used for aspect-based dynamic instrumentation in Linux [25].

All these studies demonstrate that there are good cases for aspects in system software. However, both Bossa and our own work on eCos show that a useful application of AOP to existing operating systems requires additional AOP expressivity that results in run-time overheads (e.g., temporal logic or dynamic instrumentation). So far no study exists that analyzes the effects of using AOP for the development of an operating-system kernel from the very beginning. This paper explores just that.

4 Aspect-Aware Operating-System Development

The basic idea behind aspect-aware operating-system development is the strict separation of concerns in the *implementation*. Each implementation unit provides exactly one feature; its mere presence or absence in the config-

ured source tree decides on the inclusion of the particular feature into the resulting system variant.

Technically, this comes down to a strict decoupling of policies and mechanisms by using aspects as the primary composition technique: Kernel mechanisms are glued together and extended by aspects; they support aspects by ensuring that all relevant internal control-flow transitions are available as unambigious and statically evaluable join points.

However, this availability cannot be taken for granted. Improving the configurability of eCos even further did not work as good as expected because of join-point ambiguity [17]. For instance, eCos does not expose a dedicated user API to invoke system services. This means that, on the join-point level, *user*$\rightleftharpoons$*kernel* transitions are not statically distinguishable from the kernel-internal activation and termination of system services. The consequence is that policy aspects that need to hook into these events become more expensive than necessary – for instance, an aspect that implements a new *kernel-stack* policy by switching stacks when entering/leaving the kernel. The ideal implementation of the kernel-stack feature had a performance overhead of 5% for the actual stack switches, whereas the aspect implementation induced a total overhead of 124% only because of unambiguous join points. The aspect had to use dynamic pointcut functions to disambiguate at run time: It used `cflow()`, a dynamic pointcut function that induces an extra internal control-flow counter that has to be incremented, decremented, and tested at run time to yield the join points.However, in other cases it was not possible at all to disambiguate, rendering an aspect-based implementation of new configuration options impossible.

We learned from this that the exposure of all relevant gluing and extension points as statically evaluable and unambigious join points has to be understood as a primary design goal from the very beginning. The key premise for such *aspect awareness* is a component structure that makes it possible to influence the composition and shape of components as well as all run-time control flows that run through them by aspects [16].

4.1 Design Principles

The eCos experience led us to the three fundamental principles of aspect-aware operating-system development:

The principle of loose coupling. Make sure that aspects can hook into all facets of the static and dynamic integration of system components. The *binding* of components, but also their *instantiation* (e.g, placement in a certain memory region) and the time and order of their *initialization* should all be established (or at least be influenceable) by aspects.

The principle of visible transitions. Make sure that aspects can hook into all control flows that run through the system. All control-flow transitions into, out of, and within the system should be influenceable by aspects. For this they have to be represented on the join-point level as statically evaluable, unambiguous join points.

The principle of minimal extensions. Make sure that aspects can extend all features provided by the system on a fine granularity. System components and system abstractions should be fine-grained, sparse, and extensible by aspects.

Aspect awareness, as described by these principles, means that we moderate the AOP ideal of obliviousness, which is generally considered by the AOP community as a defining characteristic of AOP [11]. CiAO's system components and abstractions are *not* totally oblivious to aspects – they are supposed to provide explicit support for aspects and even depend on them for their integration.

4.2 Role and Types of Classes and Aspects

The relationship between aspects and classes is asymmetrical in most AOP languages: Aspects augment classes, but not vice versa. This gives rise to the question which features are best to be implemented as classes and which as aspects and how both should be applied to meet the above design principles.

The general rule we came up with in the development of CiAO is to provide some feature as a class if – and only if – it represents a *distinguishable instantiable concept* of the operating system. Provided as classes are:

1. **System components**, which are instantiated on behalf of the kernel and manage its run-time state (such as the `Scheduler` or the various hardware devices).

2. **System abstractions**, which are instantiated on behalf of the application and represent a system object (such as `Task`, `Resource`, or `Event`).

However, the classes for system components and system abstractions are sparse and to be further "filled" by *extension slices*. The main purpose of these classes is to provide a distinct scope with unambiguous join points for the aspects (that is, *visible transitions*).

All other features are implemented as aspects. During the development of CiAO we came up with three idiomatic roles of aspects:

1. **Extension aspects** add additional features to a system abstraction or component (*minimal extensions*), such as extending the scheduler by means for task synchronization (e.g., AUTOSAR-OS resources).

2. **Policy aspects** "glue" otherwise unrelated system abstractions or components together to implement some kernel policy (*loose coupling*), such as activating the scheduler from a periodic timer to implement time-triggered preemptive scheduling.

3. **Upcall aspects** bind behavior defined by higher layers to events produced in lower layers of the system, such as binding a driver function to interrupt events.

The effect of *extension aspects* typically becomes visible in the API of the affected system component or abstraction. *Policy aspects*, in contrast, lead to a different system behavior. We will see examples for extension and policy aspects in the following section. *Upcall aspects* do not contribute directly to a design principle, but have a more technical purpose: they exploit advice-based binding and the fact that AspectC++ inlines advice code at the respective join point for flexible, yet very efficient upcalls.

5 Case Study: CiAO-AS

CiAO is designed and implemented as a family of operating systems and has been developed from scratch using the principles of aspect-aware operating-system development. Note, however, that the application developer does not have to have any AOP expertise to use the OS. A concrete CiAO variant is configured statically by selecting features from a feature model in an Eclipse-based graphical configuration tool [4].

The *CiAO-AS* family member implements an operating-system kernel according to the AUTOSAR-OS standard[3] [2], including configurable protection policies (memory protection, timing protection, service protection). The primary target platform for CiAO is the Infineon TriCore, an architecture of 32-bit microcontrollers that also serves as a reference platform for AUTOSAR and is widely used in the automotive industry.

5.1 Overview

Figure 4 shows the basic structure of the CiAO-AS kernel. Like most operating systems, CiAO is designed with a *layered architecture*, in which each layer is implemented using the functionality of the layers below. The only exceptions to this are the aspects implementing architectural policies, which may affect multiple layers.

On the coarse level, we have three layers. From bottom up these are: the *hardware access layer,* the *system layer* (the operating system itself), and the *API layer.*

[3]Because of legal issues, we do not claim full conformance; we have not performed any formal conformance testing.

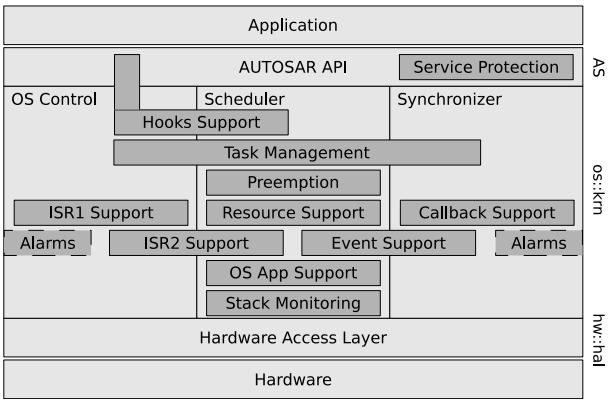


Figure 4: Structure of the CiAO-AS kernel

In CiAO, however, layers do not just serve conceptual purposes, but also are a means of aspect-aware development. With regard to the principle of *visible transitions*, each layer is represented as a separate C++ namespace in the implementation (`hw::hal`, `os::krn`, `AS`). Thereby, cross-layer control-flow transitions (especially into and out of `os::krn`) can be grasped by statically evaluable pointcut expressions. The following expression, for instance, yields all join points where a system-layer component accesses the hardware:

```
pointcut pcOStoHW() = call("% hw::hal::%(...)")
        && within("% os::krn::%(...)");
```

5.2 The Kernel

In its full configuration, the system layer bears three logical *system components* (displayed as columns in Figure 4):

1. The *scheduler* (`Scheduler`) takes care of the dispatching of tasks and the scheduling strategy.

2. The *synchronization facility* (`Synchronizer`) takes care of the management of events, alarms, and the underlying (hardware / software) counters.

3. The *OS control facility* (`OSControl`) provides services for the controlled startup and shutdown of the system and the management of OSEK/AUTOSAR application modes.

However, as pointed out in Section 4.2, these classes are sparse or even empty. If at all, they implement only a minimal base of their respective concern. All further concerns and variants (depicted in dark grey in Figure 4) are brought into the system by aspects, most of which touch multiple system components and system abstractions.

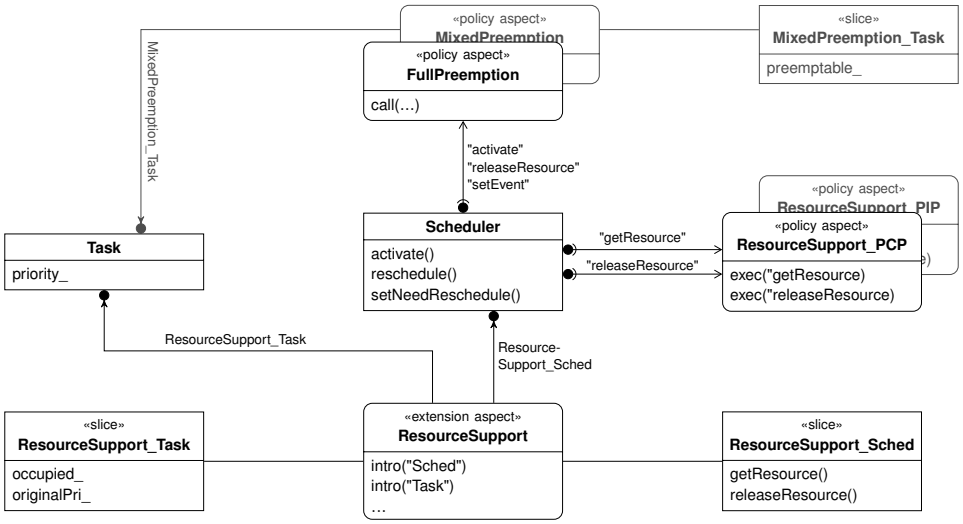


Figure 5: Interactions between optional policies and extensions of the CiAO scheduler

5.3 Aspect-Aware Development Applied

Figure 5 demonstrates how components, abstractions, and aspects engage with each other on a concrete example. The central element is the system component `Scheduler`. However, `Scheduler` provides only the minimal base of the scheduling facility, which is nonpreemptive scheduling:

```
class Sched {
  Tasklist    ready_;
  Task::Id    running_;
 public:
  void activate(Task::Id whom);
  void reschedule();
  void setNeedReschedule();
  ...
};
```

Support for preemption and further abstractions is provided by additional *extension aspects* and *policy aspects*.

`ResourceSupport` is an example for an *extension aspect*. It extends the `Task` system abstraction `Scheduler` system component with support for resources. For this purpose, it introduces some state variables (`occupied_`, `originalPri_`) and operations (`getResource()`, `releaseResource()`).[4] The elements to introduce are given by respective *extension slices*:

```
slice struct ResourceSupport_Task {
  ResourceMask occupied_;
  Pri originalPri_;
};
```

[4] `ResourceSupport` furthermore extends the API on the interface layer (it introduces the respective AUTOSAR-OS services `Get-/ReleaseResource()` and the `ResourceType` abstraction) so that applications can use the new functionality. For the sake of simplicity, this cross-layer extension is omitted here.

```
slice struct ResourceSupport_Sched {
  void getResource(Resource::Id resid) {...}
  void releaseResource(Resource::Id resid) {...}
};

aspect ResourceSupport {
  advice "Task" : slice ResourceSupport_Task;
  advice "Scheduler" : slice ResourceSupport_Sched;
};
```

`FullPreemption` is an example for a *policy aspect*. It implements the full-preemption policy as specified in [19], according to which every point where a higher-priority task may become ready is a potential point of preemption:

```
pointcut pcPreemptionPoints() =
  "% Scheduler::activate(...)" ||
  "% Scheduler::setEvent(...)" ||
  "% Scheduler::releaseResource(...)";

aspect FullPreemption {
  advice execution(pcPreemptionPoints()) : after() {
    tjp->that()->reschedule();
} };
```

The named pointcut `pcPreemptionPoints()` (defined in a global header file) specifies the potential preemption points. To these points, *if present*, the aspect `FullPreemption` binds the invocation of `reschedule()`. This demonstrates the benefits of *loose coupling* by the AOP mechanisms, which makes it easy to cope with conceptually different, but technically interacting features: In a fully-preemptive system without resource support, `Scheduler::releaseResource()` is just not present, thus does not constitute a join point for `FullPreemption`. However, if the `ResourceSupport` extension aspect is part of the current configuration, `Scheduler::release-`

concern	extension	policy	upcall	advice	join points	extension of | advice-based binding to
ISR cat. 1 support	1		m	$2+m$	$2+m$	API, OS control | m ISR bindings
ISR cat. 2 support	1		n	$5+n$	$5+n$	API, OS control, scheduler | n ISR bindings
Resource support	1	1		3	5	scheduler, API, task | PCP policy implementation
Resource tracking		1		3	4	task, ISR | monitoring of Get/ReleaseResource
Event support	1			5	5	scheduler, API, task, alarm | trigger action JP
Full preemption		1		2	6	| 3 points of rescheduling
Mixed preemption		1		3	7	task | 3 points of rescheduling for task / ISR
Wrong context check		1		1	s	| s service calls
Interrupts disabled check		1		1	30	| all services except interrupt services
Invalid parameters check		1		1	25	| services with an OS object parameter
Error hook			1	2	30	scheduler | 29 services
Protection hook	1	1		2	2	API | default policy implementation
Startup / shutdown hook			1	2	2	| explicit hooks
Pre-task / post-task hook			1	2	2	| explicit hooks

Table 2: Selected CiAO-AS kernel concerns implemented as aspects with number of affected join points. Listed are selected kernel concerns that are implemented as *extension*, *policy*, or *upcall aspects*, together with the related pieces of *advice* (not including order advice), the affected number of *join points*, and a short explanation for the purpose of each join point (separated by "|" into *introductions of extension slices* | *advice-based binding*).

`Resource()` implicitly triggers the advice. The separation of policy invocation from mechanism implementation makes it easy to integrate additional features, such as the `ResourceSupport_PCP` aspect, which implements a stack-based priority ceiling protocol for resources. As AspectC++ inlines advice code at the matching join point, this flexibility does not cause overhead at run time.

6 Discussion of Results

By following the principles of aspect-aware operating system development, policies and mechanisms are cleanly separated in the CiAO implementation. This separation is a golden rule of system-software development, but in practice difficult to achieve. While on the design level it is usually possible to describe a policy in a well-separated manner from underlying mechanisms, the implementation often tends to be crosscutting. The reason is that many system policies, such as the preemption policy, not only depend on decisions but also on the specific points in the control flow where these decisions are made. Here, the modularization into aspects shows some clear advantages.

6.1 Modularization of the System

Table 2 displays an excerpt of the list of AUTOSAR-OS concerns that are implemented as aspects in CiAO-AS. The first three columns list for each concern the number of *extension*, *policy*, and *upcall aspects* that implement the concern. (The resource-support aspect and the protection-hook aspect have both an extension and a policy facet.)

An interesting point is the realization of synergies by means of AOP *quantification.* If for some concern the number of pieces of advice is lower than the number of affected join points, we have actually profited from the AOP concept of quantification by being able to reuse advice code over several join points. For 8 out of the 14 concerns listed in Table 2, this is the case.

The net amount of this profit depends on the type of concern and aspect. *Extension aspects* typically crosscut *heterogeneously* with the implementation of other concerns, which means that they have specific pieces of advice for specific join points. These kinds of advice do not leave much potential for synergies by quantification. *Policy aspects* on the other hand – especially those for architectural policies – tend to crosscut *homogeneously* with the implementation of other concerns, which means that a specific piece of advice targets many different join points at once. In these cases, quantification creates significant synergies.

For all concerns, however, the implementation is realized as a distinct set of aspect modules, thereby reaching complete encapsulation and separation of concerns. Thus, any given feature configuration demanded by the application developer can be fulfilled by only including the implementation entities belonging to that configuration in the configured source tree to be compiled.

6.2 Scalability of the System

Execution Time. The effects of the achieved configurability also become visible in the CPU overhead. Table 3 displays the execution times of the micro-benchmark sce-

narios[5] **(a)** to **(j)** and the comprehensive application **(k)** on CiAO and a commercial OSEK implementation[6]. For each scenario, we first configured both systems to support the smallest possible set of features (*min* colums in Table 3). The differences between CiAO and OSEK are considerable: CiAO is noticeably faster in all test scenarios.

One reason for this is that CiAO provides a much better configurability (and thereby granularity) than OSEK. As the micro-benchmark scenarios utilize only subsets of the OSEK/AUTOSAR features, this has a significant effect on the resulting execution times. The smallest possible configurations of the commercial OSEK still contained a lot of unwanted functionality. The scheduler is synchronized with ISRs, for instance; however, most of the application scenarios do not include any ISRs that could possibly interrupt the kernel.

To judge these effects, we performed additional measurements with an "artifically enriched" version of CiAO that provides the same amount of unwanted functionality as OSEK (column *full* in Table 3). This reduces the performance differences; however, CiAO is still faster in six out of eleven test cases. This is most notable in test case **(k)**, which is a comprehensive application that actually *uses* the full feature set.

Another reason for the relative advantage of CiAO is that OSEK's internal thread-abstraction implementation is less efficient. This is mainly due to particularities of the TriCore platform, which renders standard context-switch implementations ported to that platform very inefficient. CiAO, however, has a highly configurable and adaptable thread abstraction, therefore not only providing for an upward tailorability (i.e., to the needs of the application), but also downward toward the deployment platform.

Memory Requirements. In embedded systems, tailorability is crucial – especially with respect to memory consumption, because RAM and ROM are typically limited to sizes of a few kilobytes. Since system software does not directly contribute to the business value of an embedded system, scalability is of particular importance here. Thus, we also investigated how the memory requirements of the CiAO-AS kernel scale up with the number of selected configurable features; the condensed results

test scenario		CiAO		OSEK
		min	full	min
(a)	voluntary task switch	*160*	*178*	*218*
(b)	forced task switch	*108*	*127*	*280*
(c)	preemptive task switch	*192*	*219*	*274*
(d)	system startup	*194*	*194*	*399*
(e)	resource acquisition	*19*	*56*	*54*
(f)	resource release	*14*	*52*	*41*
(g)	resource release with preemption	*240*	*326*	*294*
(h)	category 2 ISR latency	*47*	*47*	*47*
(i)	event blocking with task switch	*141*	*172*	*224*
(j)	event setting with preemption	*194*	*232*	*201*
(k)	comprehensive application	*748*	*748*	*1216*

Table 3: Performance measurement results [clock ticks]

are depicted in Table 4. Listed are the deltas in code, data, and BSS section size per feature that is added to the CiAO base system.

Each `Task` object, for instance, takes 20 bytes of *data* for the kernel task context (priority, state, function, stack, interrupted flag) and 16 bytes (*bss*) for the underlying CiAO thread abstraction structure. Aspects from the implementation of other features, however, may extend the size of the kernel task context. Resource support, for instance, crosscuts with task management in the implementation of the `Task` structure, which it extends by 8 bytes to accommodate the occupied resources mask and the original priority.

The cost of several features does not simply induce a constant cost, but depends on the number of affected join points, which in turn can depend on the presence of *other* features, as explained in Section 5.3 with the example of full preemption and resource support. This effect underlines again the flexibility of loose coupling by advice-based binding.

7 Experiences with the Approach

The CiAO results show that the approach of aspect-aware operating-system development is both feasible and beneficial for the class of configurable embedded operating systems. The challenge was to implement a system in which almost everything is configurable. In the following, we describe our experience with the approach.

7.1 Extensibility

We are convinced that the three design principles of aspect-aware operating-system development (*loose coupling*, *visible transitions*, *minimal extensions*) also lead to an easy extensibility of the system for new, unanticipated features. While it is generally difficult to prove the soundness of an approach for unanticipated change, we have at least some evidence that our approach has clear benefits

[5] All variants were woven and compiled for the Infineon TriCore platform with AC++-1.0PRE3 and TRICORE-G++-3.4.3 using -O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections optimization flags. Memory numbers are retrieved byte-exact from the linker-map files. Run-time numbers are measured with a high-resolution hardware trace unit (Lauterbach PowerTrace TC1796).

[6] ProOSEK is the leading commercial implementation of the OSEK standard and part of the BMW and Audi/VW standard cores. We compare CiAO against ProOSEK since (1) AUTOSAR is a true superset of OSEK and (2) we do not yet have access to a complete AUTOSAR implementation.

feature	with feature or instance	text	data	bss
Base system (OS control and tasks)				
	per task	*+ func*	*+ 20*	*+ 16 + stack*
	per application mode	*0*	*+ 4*	*0*
ISR cat. 1 support		*0*	*0*	*0*
	per ISR	*+func*	*0*	*0*
	per disable–enable	*+ 4*	*0*	*0*
Resource support		*+ 128*	*0*	*0*
	per resource	*0*	*+ 4*	*0*
	per task	*0*	*+ 8*	*0*
Event support		*+ 280*	*0*	*0*
	per task	*0*	*+ 8*	*0*
	per alarm	*0*	*+ 12*	*0*
Full preemption		*0*	*0*	*0*
	per join point	*+ 12*	*0*	*0*
Mixed preemption		*0*	*0*	*0*
	per join point	*+ 44*	*0*	*0*
	per task	*0*	*+ 4*	*0*
Wrong context check		*0*	*0*	*0*
	per void join point	*0*	*0*	*0*
	per StatusType join point	*+ 8*	*0*	*0*
Interrupts disabled check		*0*	*0*	*0*
	per join point	*+ 64*	*0*	*0*
Invalid parameters check		*0*	*0*	*0*
	per join point	*+ 36*	*0*	*0*
Error hook		*0*	*0*	*+ 4*
	per join point	*+ 54*	*0*	*0*
Startup hook or shutdown hook		*0*	*0*	*0*
Pre-task hook or post-task hook		*0*	*0*	*0*

Table 4: Scalability of CiAO's memory footprint. Listed are the increases in static memory demands [bytes] of selected configurable CiAO features.

here:

In a specific real-time application project that we implemented using CiAO, minimal and deterministic event-processing latencies were crucial. The underlying hardware platform was the Infineon TriCore, which actually is a heterogeneous multi-processor-system-on-chip that comes with an integrated peripheral control processor (PCP). This freely-programmable co-processor is able to handle interrupts independently of the main processor. We decided to extend CiAO in a way that the PCP pre-handles all hardware events (interrupts) in order to map them to activations of respective software tasks, thereby preventing the real-time problem of rate-monotonic priority inversion [8]. This way, the CPU is only interrupted when there actually is a control flow of a higher priority than the currently executing one ready to be dispatched.

This relatively complex and unanticipated extension could nevertheless be integrated into CiAO by a single *extension aspect,* which is shown in Figure 6. The `PCP_Extension` aspect is itself a *minimal extension*; its implementation profited especially from the fact that all other CiAO components are designed according to the principle of *visible transitions.* This ensures here that all relevant transitions of the CPU, such as when the kernel is entered or left (lines 9 and 14, respectively) or when the running CPU task is about to be preempted (line 17), are available as statically evaluable and unambigious join points to which the aspect can bind.

Note, that the aspect in Figure 6 is basically the complete code for that extension, except for some initialization code (10 lines of code) and the PCP code, which is written in assembly language due to the lack of a C/C++ compiler for the PCP instruction set.

7.2 The Role of Language

We think that the expressiveness of the base language (in our case C++) plays an important role for the effectiveness of the approach. Thanks to modularization through namespaces and classes, C++ has some clear advantages over C with respect to *visible transitions*: the more of the base program's purpose and semantics is expressed in its syntactic structure, the more unambigious and "semantically rich" join points are available to which the aspects can bind.

Note, however, that even though CiAO is using C++, it is not developed in an object-oriented manner. We used C++ as a purely static language and stayed away from any language feature that induces a run-time overhead, such as virtual functions, exceptions, run-time type information, and automatic construction of global variables.

7.3 Technical Issues

Aspects for Low-Level Code. A recurring challenge in the development of CiAO was that the implementation of fundamental low-level OS abstractions, such as interrupt handlers or the thread dispatcher, requires more control over the resulting machine code than is guaranteed by the semantics of ISO C++. Such functions are typically (1) written entirely in external assembly files or (2) use a mixture of inline assembly and nonstandard language extensions (such as `__attribute__((interrupt))` in `gcc`). For the sake of *visible transitions*, we generally opted for (2). However, the resulting join points often have to be considered as fragile – if advice is given to, for instance, the context switch function, the transformations performed by the aspect weaver might break the programmer's implicit assumptions about register usage or the stack layout. The workaround we came up with for these cases is to provide *explicit join points* to which the aspects can bind instead. Technically, an explicit join point is represented by an empty inline function that is invoked from the fragile code when the execution context is safe. CiAO's context switch functionality, for instance, exposes four explicit join points to which aspects can bind: `before_CPURelease()`, `before_LastCPURelease()`, `after_CPUReceive()`, and `after_FirstCPUReceive()`. Because of function inlin-

```
aspect PCP_Extension {
  advice execution("void hw::init()") : after() {
    PCP::init();
  }
  advice execution("% Scheduler::setRunning(...)") :
  before() {
    PCP::setPrio(os::krn::Task::getPri(tjp->args<0>()));
  }
  advice execution("% enterKernel(...)") : after() {
    // wait until PCP has left kernel (Peterson)
    PCP_FLAG0 = 1; PCP_TURN = 1;
    while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}
  }
  advice execution("% leaveKernel(...)") : before() {
    PCP_FLAG0 = 0;
  }
  advice execution("% AST0::ast(...)") : around() {
    // AST0::ast() is the AST handler that activates
    // the scheduler (bound by an upcall aspect)

    // wait until PCP has left kernel (Peterson)
    PCP_FLAG0 = 1; PCP_TURN = 1;
    while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}

    // proceed to aspect that activates scheduler
    tjp->proceed();
    PCP_FLAG0 = 0;
  }
  advice execution("% Scheduler::schedule(...)") : after() {
   // write priority of running task to PCP memory
   PCP::setPrio(Task::getPri(
      Scheduler::Inst().getRunning()));
  }
};
```

Figure 6: PCP co-processor extension aspect

ing, this does not induce an overhead and the aspect code is still embedded directly into the context switch functionality.

Aspect–Aspect Interdependencies. In several cases we had to deal with subtle interdependencies between aspects that affect the same join points. For instance, the following aspect implements the *ErrorHook* feature, which exempts the application developer from manually testing the result code of OS services:

```
aspect ErrorHook {
  advice execution(pcOSServices() ... ) : after() {
    if(*tjp->result() != E_OK)
      invokeErrorHook(*tjp->result());
} };
```

Later we figured that, depending on the configuration, there are also other *aspects* that modify the result code. To fulfill its specification, `ErrorHook` has to be invoked after these other aspects. Whereas detecting such interdependencies was sometimes tricky (especially those that emerge only in certain configurations), they were generally easy to resolve by *order* advice:

```
advice execution(pcOSServices() ... ) : order(
  "ErrorHook", !"ErrorHook");
```

This type of advice allows the developer to define a (partial) order of aspect invocation for a pointcut. The precedence of aspects is specified as a sequence of match expressions, which are evaluated against all aspect identifiers. In the above example, the aspect yielded by the expression `"ErrorHook"` has precedence (is invoked *last* of all aspects that give *after* advice to the pointcut) over all other aspects (the result of `!"ErrorHook"`). Very helpful was that order advice does not necessarily have to be given by one of the affected aspects, instead it can be given by any aspect. This made it relatively easy to encapsulate and deal with configuration-dependent ordering constraints.

Join-Point Traceability. An important factor for the development were effective tools for join-point traceability. From the viewpoint of an aspect developer, the set of join points offered by some class implementation constitutes an interface. However, these interfaces are "implicit at best" [23]; a simple refactoring, such as renaming a method, might silently change the set of join points and thereby break some aspect. To prevent such situations, we used the Eclipse-based AspectC++ Development Toolkit (ACDT[7]), which provides a join-point–set delta analysis (very helpful after updating from the repository) and visualizes code that is affected by aspects. Thereby, unwanted side effects of code changes could be detected relatively easy.

8 Summary and Conclusions

Operating systems for the domain of resource-constrained embedded systems have to be highly configurable. Typically, such configurability is implemented "in line" by means of the C preprocessor. However, due to feature interdependencies and the fact that system policies and system mechanisms tend to crosscut with each other in the implementation, this approach leads to "#ifdef hell" and a bad separation of concerns. Our analysis of the AUTOSAR-OS specification revealed that these effects can already be found in the requirements; they are an *inherent phenomenon* of complex systems. If fundamental architectural policies have to be provided as configurable features, "#ifdef hell" appears to be unavoidable.

We showed that a pragmatic application of aspect-oriented programming (AOP) provides means for solving these issues: The advice mechanism of AOP effectively reverses the direction of feature integration; an (optional) feature that is implemented as an aspect *integrates itself* into the base code. Thanks to AOP's pointcut expressions, the integration of features through join points is declarative – it scales implicitly with the presense or absence of

[7] `http://acdt.aspectc.org/`

other features. A key prerequisite is, however, that the system's implementation exhibits enough unambigious and statically evaluable join points. This is achieved by the three design principles of *aspect-aware operating-system development*.

By following this design approach in the development of CiAO, we did not only achieve the complete separation of concerns in the code, but also excellent configurability and scalability in the resulting system. We hope that our results encourage developers who start from scratch with a piece of configurable system software to follow the guidelines described in this paper.

Acknowledgments

We wish to thank the anonymous reviewers for EuroSys and USENIX for their helpful comments. Special thanks go to Robert Grimm, whose demanding and encouraging shepherding helped us tremendously to improve content and clarity of this paper.

References

[1] Åberg, R. A., Lawall, J. L., Südholt, M., Muller, G., and Meur, A.-F. L. On the automatic evolution of an OS kernel using temporal logic and AOP. In *18th IEEE Int. Conf. on Automated Software Engineering (ASE '03)* (Montreal, Canada, Mar. 2003), IEEE, pp. 196–204.

[2] AUTOSAR. Specification of operating system (version 2.0.1). Tech. rep., Automotive Open System Architecture GbR, June 2006.

[3] AUTOSAR homepage. `http://www.autosar.org/`, visited 2009-03-26.

[4] Beuche, D. Variant management with pure::variants. Tech. rep., pure-systems GmbH, 2006. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, visited 2009-03-26.

[5] Broy, M. Challenges in automotive software engineering. In *28th Int. Conf. on Software Engineering (ICSE '06)* (New York, NY, USA, 2006), ACM, pp. 33–42.

[6] Campbell, R., Islam, N., Madany, P., and Raila, D. Designing and implementing Choices: An object-oriented system in C++. *CACM 36*, 9 (1993).

[7] Coady, Y., and Kiczales, G. Back to the future: A retroactive study of aspect evolution in operating system code. In *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)* (Boston, MA, USA, Mar. 2003), M. Akşit, Ed., ACM, pp. 50–59.

[8] del Foyo, L. E. L., Mejia-Alvarez, P., and de Niz, D. Predictable interrupt management for real time kernels over conventional PC hardware. In *12th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '06)* (Los Alamitos, CA, USA, 2006), IEEE, pp. 14–23.

[9] Engel, M., and Freisleben, B. TOSKANA: a toolkit for operating system kernel aspects. In *Transactions on AOSD II* (2006), A. Rashid and M. Aksit, Eds., no. 4242 in LNCS, Springer, pp. 182–226.

[10] Filman, R. E., Elrad, T., Clarke, S., and Aksit, M. *Aspect-Oriented Software Development*. AW, 2005.

[11] Filman, R. E., and Friedman, D. P. Aspect-oriented programming is quantification and obliviousness. In *W'shop on Advanced SoC (OOPSLA '00)* (Oct. 2000).

[12] Fiuczynski, M., Grimm, R., Coady, Y., and Walker, D. patch(1) considered harmful. In *10th W'shop on Hot Topics in Operating Systems (HotOS '05)* (2005), USENIX.

[13] Hofer, W., Lohmann, D., and Schröder-Preikschat, W. Concern impact analysis in configurable system software—the AUTOSAR OS case. In *7th AOSD W'shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)* (New York, NY, USA, Mar. 2008), ACM, pp. 1–6.

[14] Hunt, G. C., and Larus, J. R. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[15] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *11th Eur. Conf. on OOP (ECOOP '97)* (June 1997), M. Aksit and S. Matsuoka, Eds., vol. 1241 of *LNCS*, Springer, pp. 220–242.

[16] Lohmann, D. *Aspect Awareness in the Development of Configurable System Software*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2009.

[17] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., and Schröder-Preikschat, W. A quantitative analysis of aspects in the eCos kernel. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2006 (EuroSys '06)* (New York, NY, USA, Apr. 2006), ACM, pp. 191–204.

[18] Massa, A. *Embedded Software Development with eCos*. New Riders, 2002.

[19] OSEK/VDX Group. Operating system specification 2.2.3. Tech. rep., OSEK/VDX Group, Feb. 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2009-03-26.

[20] Padioleau, Y., Lawall, J. L., Muller, G., and Hansen, R. R. Documenting and automating collateral evolutions in Linux device drivers. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2008 (EuroSys '08)* (Glasgow, Scotland, Mar. 2008).

[21] Reynolds, A., Fiuczynski, M. E., and Grimm, R. On the feasibility of an AOSD approach to Linux kernel extensions. In *7th AOSD W'shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)* (New York, NY, USA, Mar. 2008), ACM, pp. 1–7.

[22] Spinczyk, O., and Lohmann, D. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software 20*, 7 (2007), 636–651.

[23] Steimann, F. The paradoxical success of aspect-oriented programming. In *21st ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '06)* (New York, NY, USA, 2006), ACM, pp. 481–497.

[24] Turley, J. The two percent solution. *embedded.com* (Dec. 2002). `http://www.embedded.com/story/OEG20021217S0039`, visited 2009-03-26.

[25] Yanagisawa, Y., Kourai, K., Chiba, S., and Ishikawa, R. A dynamic aspect-oriented system for OS kernels. In *6th Int. Conf. on Generative Programming and Component Engineering (GPCE '06)* (New York, NY, USA, Oct. 2006), ACM, pp. 69–78.

[26] Yokote, Y. The Apertos reflective operating system: the concept and its implementation. In *7th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '92)* (New York, NY, USA, 1992), ACM, pp. 414–434.

Aspect-Aware Operating System Development*

Daniel Lohmann[1] Wanja Hofer[1] Wolfgang Schröder-Preikschat[1] Olaf Spinczyk[2]

[1]Friedrich–Alexander University Erlangen–Nuremberg
[2]Technische Universität Dortmund
{lohmann,hofer,wosch}@cs.fau.de
olaf.spinczyk@tu-dortmund.de

ABSTRACT

The domain of operating systems has often been mentioned as an "ideal candidate" for the application of AOP; fundamental policies we find in these systems, such as *synchronization* or *preemption*, seem to be inherently cross-cutting in their implementation. Their clear separation into dedicated aspect modules should facilitate better evolvability and – the focus of this paper – *configurability.* Our experience with applying AOP to the domain of highly configurable embedded operating systems has shown, however, that these advantages can by no means be taken for granted. To reveal maximum configurability of central system policies, aspects and their potential interactions with the system have to be taken into account much earlier, that is, "from the very beginning". We propose the analysis and design process of *aspect-aware development*, which leads to such an "aspect-friendly" system structure and demonstrate its feasibility on the example of CiAO, an AUTOSAR-OS-compliant operating system that provides configurability of all fundamental system policies by means of AOP.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and Interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Measurement, Experimentation, Design

Keywords

Aspect-Aware Design, Aspect-Oriented Programming (AOP), AspectC++, CiAO

*This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4, SCHR 603/7-1, SP 968/2-1, SP 968/4-1, and LO 1719/1-1.

AOSD'11, March 21–25, 2011, Pernambuco, Brazil.

1. INTRODUCTION

> *Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details (mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later.* (Silberschatz et al., "Operating System Concepts", p. 72, 2005)

When, more than a decade ago, the advent of aspect-oriented programming (AOP) promised a new dimension of separation of concerns in software systems, operating systems were among the targets that were first mentioned for the new approach [16]. AOP is appealing for this domain, as fundamental operating-system concerns, such as *synchronization, preemption, prefetching,* or *monitoring* seem to be inherently cross-cutting. Their clear separation into dedicated aspect modules would facilitate better *evolvability* and *configurability* of operating-system policies [5, 8, 1]. As operating-system engineers in the domain of embedded systems – a domain for which configurability is of utmost importance – we immediately became excited when we first heard about AOP at ECOOP '97. This triggered the design and development of the AspectC++ language and tool suite [26] and extensive studies with aspects in the PURE and eCos operating system families [25, 21].

Now, ten years later, our research activities on applying AOP to the domain of configurable operating systems have culminated in the development of CiAO (CiAO is Aspect-Oriented) – the first operating system family that has been designed and developed with AOP concepts from scratch. By the application of AOP, CiAO reaches excellent configurability, a good separation of concerns, and very low resource consumption in the resulting systems, which outperforms leading commercial implementations [20]. On the path to CiAO, however, we had to learn a lot. The separation and configuration of fundamental operating system policies by aspects turned out to be surprisingly challenging. To reveal maximum benefits, the incorporation of AOP (as a programming paradigm) had to be reflected in the system's design much deeper than we had initially expected; the no-overhead integration and configuration of even low-level operating system concerns by aspects required a decent level of pragmatism.

About This Paper

In this paper, we describe our *experiences* with applying AOP to the domain of configurable operating systems for resource-constrained embedded devices – and how they led to the analysis and design method of *aspect-aware operating system development* that we came up with for CiAO. In particular, we make the following contributions:

- We describe, on the example of two case studies, *typical particularities* of system software that in practice hinder better configurability by AOP and *analyze the fundamental issues* behind them (Section 2).

- We provide a new *analysis and design method* and a set of *fundamental design principles* to improve on the situation (Section 3). We have evaluated our method with AUTOSAR OS [4], the dominant operating system standard for automotive applications.

Our contribution is rounded up by a detailed discussion of the particularities of and requirements on AOP in general for the development of configurable system software (Section 4). In Section 5 we describe related work and finally conclude with the pros and cons that AOP offers for *this particular domain* (Section 6).

2. ANALYSIS AND BACKGROUND

System software provides no business value of its own. Its sole purpose is to ease the development and integration of applications – that is, to serve application developers and integrators with the “right” set of abstractions for *their* particular problems.

2.1 Embedded Operating Systems

This is a challenge especially in the domain of small (“deeply”) embedded systems, which are subject to an enormous hardware-cost pressure. System software for this domain has to cope not only with strict resource constraints, but also with a broad variety of application requirements and platforms. For instance, power-train applications are typically safety-critical and have to deal with real-time requirements, while car body systems are far less critical. Hardware platforms range from 8-bit to 32-bit systems. Some applications require a task model with synchronization and communication primitives, whereas others are much simpler control loops. Thus, to allow for reuse, an operating system for the embedded systems domain has to be designed and developed as a software family – that is, for configurability (provide alternatives) and tailorability (leave out as much as possible). Furthermore, resource-saving static configuration mechanisms are strongly favored over dynamic (re-)configuration.

This necessity for best-possible configurability and tailorability was the reason we considered AOP to be so promising: It facilitates separation of many more concerns than the traditional implementation techniques. We especially sought for a technique to implement even fundamental internal “architectural” policies of an operating system kernel in a configurable and tailorable manner. Some examples for such fundamental policies are: Synchronization of kernel components (explicit vs. implicit, fine-grained vs. coarse-grained, hardware-supported), Interaction between kernel components (message-based vs. procedure-based), Preemption of control flows inside the kernel (fully-preemptive, at dedicated preemption points, no preemption until the kernel is left), and Protection of kernel components against invalid access and behavior (coarse-grained vs. fine-grained vs. no memory protection, deadline monitoring, parameter validation).

Such fundamental internal policies define what is commonly referred to as the *architecture* of an operating system, like *micro-kernel* (message-based interaction, implicit synchronization, fine-grained protection) or *monolith* (procedure-based interaction, explicit synchronization, coarse-grained protection). Their implementation, however, is notoriously cross-cutting and, hence, often hard-wired into the system – our call for AOP.

2.2 Early AOP Experiences

Our initial experiences with employing aspect to improve on the situation in the PURE and eCos operating system families, were, despite many success stories [25, 21], double edged: When it comes to the actual implementation, apparently orthogonal concerns (such as Interaction, Synchronization, and Preemption) often turned out to induce hidden functional dependencies and unexpected ambiguities on the join-point level. The following two examples from PURE and eCos illustrate some of the more problematic cases.

2.2.1 Device-Driver Invocation in PURE

In the late nineties, our research group developed the PURE family of operating systems for deeply embedded devices [6]. With more than 250 configurable features and a kernel memory footprint between 434 B and >100 KiB, PURE offers excellent scalability. We achieved this scalability without a single `#ifdef` in the C++ code by a design approach that put old ideas from HABERMANN and PARNAS (*functional dependencies* and *functional hierarchies* [13, 23]) to an extreme and that mapped functional layers to C++ classes.

PURE, however, did not offer configurability of fundamental system policies. Later, we applied AspectC++ to improve on the configurability of its architectural system policies, among them Interrupt Synchronization and Interaction between application code and device drivers [25]. This worked successfully in the first case; however, we ran into unexpected difficulties in the second case.

The scenario was as follows: In the default configuration, the invocation of device-driver services (such as `FloppyDriver::readBlock()`) is implemented by plain method calls, explicitly synchronized on a per-driver basis by mutex objects. A (more micro-kernel–like) architectural alternative for this implementation of Interaction and Synchronization is to employ message passing and active servers: Each device driver is a mini server that runs a message loop in its own thread. In [25], the `ServerSync` aspect implements this alternative (well encapsulated and transparently for application and device-driver developers) by the introduction of a `Thread` object into each driver class plus a piece of around-advice that intercepts all noninternal calls to device-driver services (`call(“% FloppyDriver::%(...)”) && !within(“FloppyDriver”)`) to transform them into messages that are sent to and dispatched by the introduced thread.

A significant side effect of the `ServerSync` aspect is that (by employing threads) it induces a new *functional dependency* `FloppyDriver` → `Scheduler`, which had not been reflected in the original PURE design. Such *ex post* changes to the functional hierarchy of an operating system are risky; they may induce dependency cycles or otherwise invalidate correctness assumptions of the system’s design [13, 23].

In our initial tests and analyses for [25], this new dependency was apparently compatible to the existing design; `FloppyDriver` and `Scheduler` were completely unrelated before. Later, however, we realized that both concerns *indirectly* interact with each other – via *another* policy we had *not* explicitly considered before: Initialization. If device drivers employ threads, the `Scheduler` has to be initialized *before* the driver objects – whereas in the default configuration the order of initialization does not matter. The latter is the correctness assumption that is invalidated by the `ServerSync` aspect. The problem: PURE implements the Initialization of system components (such as `Scheduler` and `FloppyDriver`) by means of C++ global instance construction, for which the order is undefined across different compilation units; it cannot be influenced by aspects, such as `ServerSync`. In the end it turned out to be *technically* impossible to turn Interaction (and several other architectural poli-

```
void Cyg_Alarm::enable() {
  // Prevent DSR execution
  Cyg_Scheduler::lock();
  if( !enabled ){
    // ensure the alarm time is in our future:
    synchronize();
    enabled = true;
    counter->add_alarm(this);
  }
  // Unlock the scheduler and propagate
  // DSRs. (No thread was set ready, so
  // this is no point of preemption.)
  Cyg_Scheduler::unlock();
}
```

```
void Cyg_Mutex::unlock() {
  // Prevent preemption and DSR execution
  Cyg_Scheduler::lock();
  if( !queue.empty() ) {
    Cyg_Thread *thread = queue.dequeue();
    thread->set_wake_reason(Cyg_Thread::DONE);
    thread->wake();
  }
  locked      = false;
  owner       = NULL;
  // Unlock the scheduler, propagate DSRs
  // and maybe switch threads
  Cyg_Scheduler::unlock();
}
```

Figure 1: Join point ambiguity in the eCos kernel. Because `Cyg_Scheduler::unlock()` is not only used to enforce Synchronization, but also Preemption, the related execution join points are ambiguous.

cies) into fully configurable features by means of aspects due to (1) *hidden dependencies* to *other* policies that (2) were designed in an *"AOP-unfriendly"* way.

2.2.2 Synchronization and Preemption in eCos

eCos, the *embedded Configurable operating system* [9, 22] is an industry-strength and broadly accepted open-source operating system family for the embedded systems domain. Including all optional packages, eCos offers more than 750 configuration options; the kernel itself consists of 5,000 lines of C++ code and offers nearly 100 configuration options, which are technically implemented by means of the C preprocessor – an "`#ifdef` hell".

As part of a larger case study about the run-time and memory effects of AOP, we refactored 16 eCos configuration options and kernel policies from conditional compilation into aspects – and thereby achieved a much better separation of concerns *without* extra run-time and memory costs [21]. One of these polices was Synchronization, which in eCos enforces mutual exclusion between threads and in-kernel interrupt handlers (called *deferred service routines, DSRs*) in order to ensure consistency of kernel state. Although only required if both threads *and* DSRs are actually employed by the application (many embedded applications use only either one), Synchronization is a mandatory feature in eCos that always causes run-time and memory costs. This is probably due to its implementation, which homogeneously cross-cuts large parts of the kernel source base: Each kernel function (as shown in Figure 1) is wrapped by calls to `Cyg_Scheduler::lock()` and `Cyg_Scheduler::unlock()` (187 invocations in total).

Extracting and separating Synchronization into an aspect was straightforward and clearly improved the clarity of the code [21]. However, our further attempts to thereby also *improve* the tailorability of eCos (turning Synchronization into a truly optional feature should be simple if implemented by an aspect) have failed. The eCos developers exploited the fact that `Cyg_Scheduler::unlock()` is called by all kernel functions immediately before the kernel is left to piggyback the enforcement of another central kernel policy on it. `Cyg_Scheduler::unlock()` does not only re-enable DSR propagation (Synchronization), it also activates the scheduler to possibly preempt the running thread (Preemption). The result of this "clever optimization" is ambiguity: Apparently, all 101 invocations of `Cyg_Scheduler::unlock()` that can be found in the kernel sources represent a point of Synchronization (like in `Cyg_Alarm::enable()` in Figure 1), but only 51 of them also represent a point of Preemption, for which the scheduler activation is actually necessary (like in `Cyg_Mutex::unlock()`). However, both concerns are *not distinguishable* on the join-point level; leaving out the enforcement of Synchronization would partly remove Preemption as well – even though both policies are conceptually independent concerns and all Synchronization code has been well encapsulated into an aspect.

2.3 Lessons Learned – A Summary

The respective "show stoppers" we encountered in the described PURE and eCos problem cases may appear to be very specific. Nevertheless, they exemplify some *general issues* we have found over the years in our attempts to achieve the configurability of even fundamental system policies in operating systems:

Hidden concerns caused by correctness assumptions that manifest only *implicitly* in the specification, in the design and – especially – in the implementation of the operating system.

Missing join points caused by optimizations, low-level code, and a generally "aspect-unfriendly" design and implementation.

The Initialization and Preemption policies in PURE and eCos, respectively, are examples for (partly) *hidden concerns*. Conceptually orthogonal to the rest of the system, their *concrete realization* causes *hidden functional dependencies* with respect to other concerns. However, even though eventually revealed and *theoretically resolvable*, it turned out as technically impossible to actually resolve these issues by the aspects, because of *missing join points* in the system's *design* and *implementation*.

2.3.1 Hidden Concerns

Hidden concerns can probably be found in any type of software, but operating systems are particularly prone to them. In our experience, they are often caused by the (comparatively complex) internal control-flow interaction schemes: With *interrupts*, *DSRs*, and *threads*, even small operating systems, like PURE and eCos, support at least three different *types* of control flows, all of which bear specific (and often subtle) interaction constraints. Interrupts and DSRs, for instance, must never block, whereas threads have to be aware of preemption and interruption at any time. Features that have been designed and implemented with *implicit assumptions* in this respect (like "this code is {never | always} invoked on {interrupt | thread | ...} level") can be found in every operating system. These assumptions, however, are often invalidated if we implement fundamental system policies as configurable features: A developer implementing the Threading concern, for instance, typically does this under the assumption that the context-switching code is never invoked from the interrupt level; hence, it does not have to be interrupt-safe. Now consider a `IRQThreads` policy aspect that reduces interrupt latencies by mapping interrupt requests to thread activations (a strategy implemented, for instance, by Solaris [17]). If `IRQThreads` is applied, the original assumption is no longer valid. This is not *per se* an issue if (1) the aspect developer is aware of this fact, *and* (2) it is possible to resolve the new dependency by

augmenting the context-switching code by an aspect in order to make it interrupt-safe.

2.3.2 *Missing Join Points*

The latter could be achieved, for instance, by a piece of advice that disables (re-enables) interrupts before (after) each context switch – but only if *all* respective thread transitions are exposed as unambiguous join-points to which aspects safely can bind. In theory, missing join-points should never be an issue: “Just program like you always do, and we’ll be able to add the aspects later” [11]. In practice, the *obliviousness principle* in the form as postulated by FILMAN and FRIEDMAN has probably not passed the reality check for any type of software. However, in our experience, the problem of missing join-points is *particularly* challenging in operating systems: The context-switching code, for instance, may be scattered over the scheduler implementation for optimization purposes (the case in eCos). Parts of it are typically written in assembly language, which does not expose join-points for aspects written in a high-level language, such as AspectC++. In other cases, join-points are exposed by fragile *near-hardware code* – to which aspects better do not bind, because the transformations performed by the aspect weaver will probably break this code. A further issue are *join-point ambiguities* like we found in eCos.

Join-point ambiguities seem to be a general problem of optimized systems code: ÅBERG and colleagues found a similar situation in the Linux scheduling code and had to enhance their pointcut language by temporal logic [1] to disambiguate scheduling events at run time with stateful aspects (very similar to the later *tracematches* [2]). Besides the *additional* run-time overhead such approach induces it also effectively results in some sort of late binding of the respective features (such as Preemption and Synchronization). This spoils dead-code elimination and, thereby, the original goal: To remove unneeded functionality from the resulting binary. Embedded systems engineers consider such overheads as unacceptable – especially if a static solution *would* be possible.

We expected all these problems to become even more severe if we want to configure *many* fundamental policies.

3. THE CIAO APPROACH

Based on our experiences with using AOP in the PURE and eCos operating systems, we have developed an integrated analysis and development approach for operating systems. Our approach provides for (1) the *early identification* of hidden concerns and (2) their *aspect-aware* design and implementation.

The overall goal is to reach *configurability* even *of fundamental system policies*, whose implementation is highly cross-cutting and interacting in traditional operating systems. Furthermore, by using aspects for the implementation of system configurability, we want to reach a *more fine-grained level of configuration* possibilities – without trading off efficiency in terms of resource usage. Since hardware resource usage is crucial in most embedded systems, our precondition here is to aim at overhead-free configurability, as it would be possible with traditional conditional compilation.

In order to keep the investigation of the suitability of AOP for operating system engineering as independent as possible, we decided to start with a publicly available standard in the domain – AUTOSAR OS [4, 3]. This way, the choice of OS abstractions and system services and their functionality is not biased by the intended AOP implementation. In fact, AUTOSAR OS is very C-focused, and most implementations are therefore also in C, configured by some kind of code preprocessor. We briefly introduce AUTOSAR in Section 3.1, followed by a description of how we analyzed the specification for crosscutting with a method called Concern Impact Analysis (see Section 3.2). After that, we present our concept of aspect-aware operating system development in Section 3.3.

3.1 AUTOSAR OS

AUTOSAR is an initiative formed by all major automotive manufacturers and suppliers like BMW, Ford, Toyota, and Bosch. Their goal is to standardize the interfaces and functionality of the operating system and drivers in automotive microcontrollers in order to facilitate application development in the domain. The operating system standard, AUTOSAR OS [4, 3] describes a kernel that is completely statically configured; the overall system configuration is known at compile time.

AUTOSAR OS offers different kinds of abstractions to the application programmer. Among the control flows, there are tasks (named threads in other operating systems) and hooks, which are callback functions invoked when the corresponding internal point in the system is reached (e.g., upon a task switch, or upon a protection violation). Interrupt services routines (ISRs) are invoked asynchronously by the hardware; ISRs of category 1 must not use OS services, whereas ISRs of category 2 are allowed to invoke the kernel and must therefore be synchronized with the kernel in order not to corrupt kernel state. Tasks and ISRs themselves can synchronize by acquiring and releasing AUTOSAR resources; AUTOSAR events can be used for task and ISR notification. AUTOSAR alarms allow the application to take action after a specified amount of time has elapsed.

The main point that distinguishes AUTOSAR OS from other operating systems in the domain is its configurable support for properties of architectural kinds. These include the decision to make the system fully-, mixed-, or non-preemptable, and different levels of protection between AUTOSAR applications. Protection entails memory protection to prevent memory corruption, timing protection to ensure that applications will not miss their deadlines because of another, misbehaving application, and service protection, which checks for correct usage and context of system-service invocation.

3.2 Concern Impact Analysis

In order to be able to design an aspect-aware system, the developer not only has to scope the system’s functionality in the analysis phase, but he also needs to assess the effect of configurable concerns on the system. This raised awareness of the different concerns in the system and their relationships to each other are the main goal of our specialized analysis process named Concern Impact Analysis (CIA; see Figure 2). CIA eventually aims to provide the information necessary to map the initially abstract concerns (c and d in Figure 2) to design and implementation artifacts such as classes or aspects (g–l in Figure 2).

In a first step, the developer scrutinizes the given requirements in the form of a specification or abstract requirements list, mining it for distinguishable concerns in the target domain. This includes concerns that will be kept configurable (and therefore omissible) in the final program family, but also concerns that are fundamental to all system variants. To some extent, this subprocess requires the knowledge of a domain expert, who will also be able to identify concerns that are internal to a system. Such internal concerns are rarely mentioned explicitly in a requirements or specification document; nevertheless, many of them are vital to a working software system. The overall outcome of the first step is a list of identified *explicit* concerns plus a preliminary list of identified *internal* concerns, both of which serve as a basis for the following impact analysis. Obviously, the concern list is a super set of all features that *could* be present in a system; it is subject to tailoring by the system configurator by excluding features from the configuration.

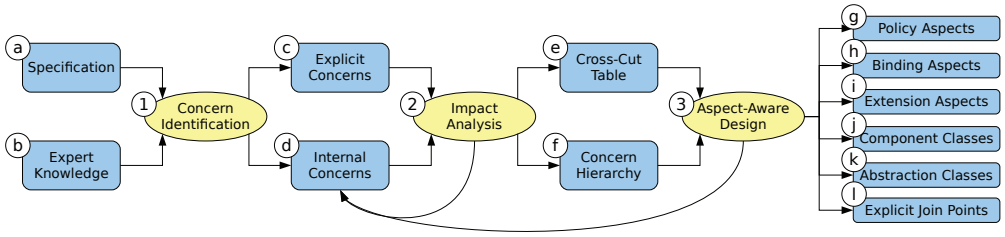


Figure 2: The process of Concern Impact Analysis (CIA) to aid aspect awareness in the system design

	System abstractions (functional)							Callbacks		Protection facilities (architectural)										Internal	
	OS control	Tasks	ISRs category 1	ISRs category 2	Resources	Events	Alarms	Alarm callbacks	Hooks	OS applications	Memory protection	Stack monitoring	Timing protection	Invalid parameters	Out of range	Wrong context	Missing task end	Enable w/o disable	Interrupts disabled	Nontrusted shutdown	Foreign OS objects	Preemption	Kernel sync
GetActiveApplicationMode()	⊕								◑							◐			◐				
StartOS()	⊕													◐		◐			◐		◐		
ShutdownOS()	⊕								◑							◐			◐	◐			
ActivateTask()		⊕							◑					◐		◐			◐		◐	◑	
TerminateTask()		⊕							◑							◐			◐			◑	
ChainTask()		⊕							◑					◐		◐			◐		◐	◑	
Schedule()		⊕							◑							◐			◐			◑	
GetTaskID()		⊕							◑							◐			◐				
GetTaskState()		⊕							◑					◐		◐			◐		◐		
EnableAllInterrupts()			⊕										◐			◐		◐					
DisableAllInterrupts()			⊕										◑			◐							
ResumeAllInterrupts()			⊕										◐			◐		◐					
SuspendAllInterrupts()			⊕										◑			◐							
ResumeOSInterrupts()				⊕									◐			◐		◐					
SuspendOSInterrupts()				⊕									◑			◐							
GetISRID()				⊕												◐			◐				
DisableInterruptSource()				⊕					◑					◐		◐			◐		◐		
EnableInterruptSource()				⊕					◑					◐		◐			◐		◐		
GetResource()					⊕				◑				◑	◐		◐			◐		◐		
ReleaseResource()					⊕				◑				◐	◐		◐			◐		◐	◑	
SetEvent()						⊕			◑					◐		◐			◐		◐	◑	
ClearEvent()						⊕			◑							◐			◐				
GetEvent()						⊕			◑					◐		◐			◐		◐		
WaitEvent()						⊕			◑							◐			◐			◑	
IncrementCounter()							⊕		◑					◐		◐			◐		◐	◑	
GetAlarmBase()							⊕		◑					◐		◐			◐		◐		
GetAlarm()							⊕		◑					◐		◐			◐		◐		
SetRelAlarm()							⊕		◑					◐	◐	◐			◐		◐		
SetAbsAlarm()							⊕		◑					◐	◐	◐			◐		◐		
CancelAlarm()							⊕		◑					◐		◐			◐		◐		
StartScheduleTableRel()							⊕		◑					◐	◐	◐			◐		◐		
StartScheduleTableAbs()							⊕		◑					◐	◐	◐			◐		◐		
StopScheduleTable()							⊕		◑					◐		◐			◐		◐		
NextScheduleTable()							⊕		◑					◐		◐			◐		◐		
SetScheduleTableAsync()							⊕		◑					◐		◐			◐		◐		
SyncScheduleTable()							⊕		◑					◐		◐			◐		◐		
GetScheduleTableStatus()							⊕		◑					◐		◐			◐		◐		
GetApplicationID()										⊕						◐			◐				
TerminateApplication()									◑	⊕						◐			◐				
CallTrustedFunction()									◑	⊕						◐			◐				
CheckObjectAccess()										⊕				◐		◐			◐		◐		
CheckObjectOwnership()										⊕						◐			◐				
CheckISRMemoryAccess()											⊕			◐		◐			◐		◐		
CheckTaskMemoryAccess()											⊕			◐		◐			◐		◐		
AppModeType	⊕	⊛					⊛																
TaskType		⊕			⊛	⊛				⊛	⊛	⊛	⊛								⊛	⊛	
ISR category 2				⊕						⊛			⊛								⊛		
ResourceType					⊕					⊛											⊛		
AlarmType/ScheduleTableType		⊛				⊛	⊕	⊛		⊛					⊛						⊛		
ApplicationType										⊕	⊛												
alarm expiry		◑				◑		◑														◑	
category 2 ISR execution											●		●									◑	
system startup		◑					◑		◑														
system shutdown									◐														
protection violation									◑														
task switch									●		◑	◐	●										
application switch											◑										◑		
uncontrolled task end																	◑						
user ↦ kernel transition											◑												◐
kernel ↦ user transition											◑												◑

Table 1: Influence of configurable concerns (columns) on system services, system types, and internal events (rows) in AUTOSAR OS. Kind of influence: ⊕ = introduction of a service or type, ⊛ = impact on a type, ◐ / ◑ / ● = impact before / after / around a service or internal event.

3.2.1 Cross-Cut Table

The main CIA step is the actual analysis of the impact each concern has on the system to be built (see step 2 in Figure 2). The impact is classified and visualized in a cross-cut table matrix (e in Figure 2) as exemplified in Table 1 with the analysis of the AUTOSAR OS specification and the design of CiAO.

As a starting point for the cross-cut table (which is, to some degree, comparable to a design structure matrix [7]), the columns list the concerns as identified in the first step, whereas the rows above the double line list the API of the system to be built, in this case AUTOSAR OS. The API includes both system services (listed in semantic groups in Table 1) as well as instantiable system types. The events listed below the double line include system-internal transitions that are of importance to one concern or the other. This list of transitions does not form until the actual analysis of the concerns takes place, which is when the need to list these transitions emerges.

In order to populate the cross-cut table, in an iterative process, each concern is analyzed for its impact on the system to be. This mainly entails three types of impact:

1. Extension of the system API by a service or a type. Some concerns are reflected in a system's interface to the using components, since without them, certain services cannot be offered by the system. Thus, those concerns *introduce* API services and types, denoted by a ⊕ sign in the cross-cut table.

2. Modification of a system service or its functionality. Several concerns will not affect the system's external interface, but they will alter or adapt its functionality. Usually, this adaptation only affects the execution or invocation of well-selected services, which is denoted by a ◐, ◑, or ● sign in the corresponding row and column in the table. If besides that, a concern needs to be notified of additional events that are internal to the system, that event is listed in an additional row below the double line and marked to be influenced by the given concern.

3. Extension of a system type. In some cases, a concern will not introduce a new type to be able to fulfill its duty, but it will instead need to extend an existing API type as introduced by another concern. This type-internal extension is denoted by a ⊛ sign in the cross-cut table.

We are aware of the fact that producing the cross-cut table means thinking about the concerns' *implementation* to some degree already in the analysis phase. However, we think that this is crucial to be able to design a complex configurable software system in an aspect-aware manner, because it is the cross-cut table that enables the developer to make informed decisions about the system architecture. Furthermore, forcing the developer to think about the impact of each concern will reveal additional internal concerns that were previously hidden in the requirements (see feedback loop from step 2 in Figure 2). A typical example from the operating systems domain is kernel synchronization, which is rarely mentioned but vital to keeping kernel state consistent if interrupt service routines are allowed to call system services. Revealing and analyzing such concerns in the early analysis stage makes the subsequent design process respect them explicitly.

Consider, for instance, the different sets of concerns as depicted in Table 1 (vertical analysis view). The system abstraction concerns each canonically extend the system API by the corresponding system services and types of the given abstraction, since they extend the system *functionally*. Note that each system service is introduced by exactly one concern (i.e., exactly one ⊕ sign per service/type row). The architectural concerns, however, which all implement some form of protection mechanism, mostly influence and enhance *existing* system services. Some of them are highly cross-cutting (e.g., consider the column corresponding to the concern "wrong context", which checks for the correct invocation context of a system service call), whereas others have a very selective influence on the system (e.g., consider the concern "nontrusted shutdown", which prevents nontrusted application from shutting down the whole system). By making the type of influence and its locality and dimension explicit in the cross-cut table diagram, the concerns can be directly considered to be modeled as a class or an aspect in the aspect-aware design step (see Section 3.3). Highly cross-cutting concerns, for instance, will probably benefit the most from AOP quantification mechanisms and are likely candidates for an aspect module implementation.

On the other hand, consider the system services and types as depicted in Table 1 (horizontal analysis view). Note that not a single service is influenced by only one concern; in fact, system services such as ReleaseResource() are influenced by as many as eight concerns! Hence, such "hot spot" services require special attention in the design, and potential aspect implementations of influencing concerns need to be carefully ordered not to break each other's functionality.

Ultimately, a comprehensive analysis with a comprehensive impact table like the one in Table 1 will provide an ideal basis for the aspect-aware design of the system concerns, making the implementation a straight-forward step.

3.2.2 Concern Hierarchy

The second artifact to be output by the impact analysis step besides the cross-cut table is a concern hierarchy (f in Figure 2). Due to space constraints, we have omitted the resulting concern hierarchy of CiAO from this paper. However, a concern hierarchy is basically a functional hierarchy [13] enriched by *influence* relationships between the different (sub) concerns. It describes, on the one hand, which concern *uses* which other concerns: This means that the respective concerns are tightly coupled – the functional correctness of the using concern depends on the one of the used concern. On the other hand, an extension to functional hierarchies, a concern might only *influence* other concerns: This indicates a *loose* coupling; if one of the target concerns is not included in a given system configuration, the source concern will still be able to fulfill its specification semantically.

We found that a concern that extends one or more system types (denoted by a ⊛ sign in the cross-cut table) typically *uses* the concerns that introduce the respective types. In AUTOSAR OS, for instance, events are generally task bound, hence, the Events concern *uses* the Tasks concern. This is already indicated in Table 1 by the fact that Events extends `TaskType`, which is introduced by Tasks.

A concern that modifies system services only, on the other hand, usually *influences* the respective target concerns. The Interrupts disabled concern, for instance, ensures for a number of system services (among them `GetResource()` and `ReleaseResource()`) that they are not invoked while running on interrupt level (with interrupts disabled). If some *influenced* concern (e.g., Resources) is not present in the current configuration, this property is implicitly true for its services.

3.3 Aspect-Aware Design

Eventually, the system's concerns and their interactions have been identified and described as far as possible. The goal of the following step 3 (Figure 2) is then to compose the gained knowledge into a model of *classes*, *aspects*, and, where necessary, *explicit join points*

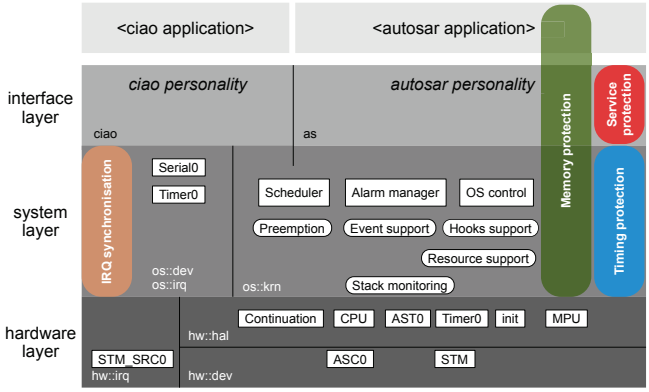


Figure 3: Layered structure of CiAO. Depicted are the three fundamental layers of the CiAO architecture with a selection of their sublayers, components, abstractions, and aspects (depicted with rounded corners).

(documents g–l). Based on our experience with PURE and eCos, this process is guided by three fundamental principles of aspect-aware software development:

The principle of loose coupling. Make sure that aspects can hook into all facets of the static and dynamic integration of system components. The *binding* of components, but also their *instantiation* (e.g, placement in a certain memory region) and the time and order of their *initialization* should all be established (or at least be influenceable) by aspects.

The principle of visible transitions. Make sure that aspects can hook into all control flows that run through the system. All control-flow transitions into, out of, and within the system should be influenceable by aspects. For this they have to be represented on the join-point level as statically evaluable, unambiguous join-point shadows.

The principle of minimal extensions. Make sure that aspects can extend all features provided by the system on a fine granularity. System components and system abstractions should be fine-grained, sparse, and extensible by aspects.

All subsequent design and implementation decisions are evaluated with respect to these three principles.

3.3.1 CiAO Architecture

The first steps towards *aspect-awareness* are already made in the architecture of the system: Like most operating systems, CiAO is based on a layered architecture, in which each layer is implemented using the functionality of the layers below (Figure 3). The only exceptions from this are the aspects implementing architectural policies, which may take effect across multiple layers.

On the coarse level, we have three layers. From bottom up these are: the *hardware layer* (`hw`, hardware programming interface), the *system layer* (`os`, the operating system itself), and the *interface layer* (`ciao` or `as`, the (configurable) application programming interface).

This architecture is aspect-aware in the sense that layers do not only serve as conceptual levels of abstraction, but also as a means to provide cross-layer control-flow transitions on the join-point level (*visible transitions*). Each layer is modeled as a top-level C++ namespace or class, which makes it easy to grasp such transitions by pointcuts, like the following AspectC++ pointcut yields all join points where a system-layer component (namespace `os`) accesses the hardware (namespace `hw`):

```
pointcut OStoHW() = call("% hw::...::%(...)")
                 && within("% os::...::%(...)");
```

Control-flow transitions *down* the layer hierarchy (such as the invocation of some system service) are established by method calls; aspects can interfere with these transitions by giving advice to a pointcut like `OStoHW`. Transitions *up* the hierarchy (*upcalls,* such as a thread start or a signal delivery) are modeled as explicit join-point shadows and *only* established by aspects (*loose coupling*). In the case of CiAO, aspects thereby can hook into all transitions into and out of the *system layer* that are visible on the static join-point level (*visible transitions*).

3.3.2 Classes and Aspects

With respect to the three design principles: Which concerns are best to be implemented as classes and which as aspects? With respect to *loose coupling*, we came up with the following general rule: Some concern is implemented as a class if – and only if – it represents a *distinguishable run-time–instantiable concept* of the system, otherwise it is realized as an aspect.

In the case of CiAO, this holds in particular for the **system abstractions** taken from the system specification document and identified during concern analysis. System abstractions (`AppModeType`, `TaskType`, and so on) are directly listed in the cross-cut table (Table 1, column 1) and represent the OS-managed entities that are instantiated on behalf of the application. Furthermore modeled as classes are the **system components**, which horizontally subdivide the architectural layers and represent its functional sub-domains (such as the `Scheduler` or the `AlarmManager` in the system layer, see Figure 3). Their identification is guided by the concern hierarchy, but also requires a decent amount of expert knowledge regarding (potential) synchronization and protection domains. The point, however, is: All classes that represent system abstractions and system components are *sparse* or even *empty*, that is, they implement only the *minimal base* of the respective concern (*minimal extensions*). Their major purpose is to provide a *distinct scope* for introductions of cross-component interactions (*visible transitions*). All further features are "filled in" by the aspects. During the development of CiAO we came up with three idiomatic roles of aspects:

1. **Extension aspects** add additional features to a system abstraction or component (*minimal extensions*), such as extending the scheduler by means for task synchronization (e.g., AUTOSAR OS resources).

2. **Policy aspects** "glue" otherwise unrelated system abstractions or components together to implement some kernel policy (*loose coupling*), such as activating the scheduler from a periodic timer to implement time-triggered preemptive scheduling.

3. **Upcall aspects** bind behavior defined by higher layers to events produced in lower layers of the system, such as binding a driver function to interrupt events.

Extension aspects can be identified in the cross-cut table by the fact that they affect especially the static structure, typically by introducing some system services. Most extension aspects accompany some system abstraction (e.g., `ResourceType`); they integrate the actual *implementation* of the respective concern (Resources) into the system components (`Scheduler`) and extend the interface layer by the corresponding services (`GetResource()`, `ReleaseResource()`).

Policy aspects, in contrast, lead to a different system behavior. In the cross-cut table they can be identified by seeking concerns that (mostly) affect the dynamic structure of the system, like Preemption.

concern	extension	policy	upcall	advice	join points	extension of | advice-based binding to
ISR cat. 1 support	1		m	$2+m$	$2+m$	API, OS control | m ISR bindings
ISR cat. 2 support	1		n	$5+n$	$5+n$	API, OS control, scheduler | n ISR bindings
Resource support	1	1		3	5	scheduler, API, task | PCP policy implementation
Resource tracking		1		3	4	task, ISR | monitoring of Get/ReleaseResource
Event support	1			5	5	scheduler, API, task, alarm | trigger action JP
Full preemption		1		2	6	| 3 points of rescheduling
Mixed preemption		1		3	7	task | 3 points of rescheduling for task / ISR
Wrong context check		1		1	s	| s service calls
Interrupts disabled check		1		1	30	| all services except interrupt services
Invalid parameters check		1		1	25	| services with an OS object parameter
Error hook			1	2	30	scheduler | 29 services
Protection hook	1	1		2	2	API | default policy implementation
Startup / shutdown hook			1	2	2	| explicit hooks
Pre-task / post-task hook			1	2	2	| explicit hooks

Table 2: Selected CiAO-AS kernel concerns implemented as aspects with number of affected join points. Listed are selected kernel concerns that are implemented as *extension*, *policy*, or *upcall aspects*, together with the related pieces of *advice* (not including order advice), the affected number of *join points*, and a short explanation for the purpose of each join point (separated by “|” into *introductions of extension slices* | *advice-based binding*).

Upcall aspects realize *loose coupling* with respect to upcalls, as described in Section 3.3.1. They are invisible in the cross-cut table, as most of them do not manifest before the implementation phase.

Table 2 displays an excerpt of the list of AUTOSAR OS concerns that are implemented as aspects in CiAO. The first three columns list for each concern the number of *extension*, *policy*, and *upcall aspects* that implement the concern. (The resource-support aspect and the protection-hook aspect have both an extension and a policy facet.) The majority of concerns contribute to the set of *policy aspects* (12 aspects), which is followed by the set of *extension aspects* (9 aspects). The number of *upcall aspects* ($3+n+m$) differs from these in so far as it does not only depend on the system configuration, but also on the application configuration: Each specified ISR in the application is bound with the respective interrupt source in the kernel or hardware access layer (HAL) by its own upcall aspect. These aspects are, however, not to be provided by the application developer; they are generated automatically from the application configuration.

3.3.3 Explicit Join Points

By consequent application of the fundamental principles of aspect-aware development in the architecture and design of the system, CiAO already offers a rich join-point interface “by structure”. Nevertheless, in many cases the implicit join-point interface is not ample enough. This has conceptual as well as technical reasons:

1. Implicit join points are inherently implementation dependent. Their amount – but especially their semantics – may be inconsistent between different implementations of the same concept. This is absolutely acceptable for component-specific extension aspects, as these aspects have to know the component they extend anyway. It is, however, not satisfying for system aspects that implement more general policies.

2. Some semantically important control-flow transitions are not visible at the join-point level because they do not occur on the boundary of function calls or executions. In other cases, their place of occurrence is configuration-dependent, or there are multiple places of occurrence. For example, *user* $\mapsto$ *kernel* transitions might occur if a kernel function is called, when a trap handler is activated, or during task switching to another task. However, in CiAO, this is a matter of configuration.

3. Several semantically important control-flow transitions are not available as join points because of technical reasons. This is often the case with low-level system abstractions, such as interrupt handlers or the implementation of the context switch mechanism.

For these reasons, many CiAO components and layers provide furthermore a well-defined **explicit join-point interface** that defines one or several *explicit join points*. An **explicit join point** is a named join point in the kernel control flow that bears precisely defined semantics and can safely be advised. Technically, explicit join points are implemented as empty methods – provided for the sole purpose that aspects can bind to them. The join-point provider invokes these methods at run time, either directly or indirectly by component-specific adapter aspects.

Conceptually, explicit join-point interfaces can be compared to hooks or interceptor interfaces in other component models. An advantage of explicit join points is, however, their low overhead. In most cases (that is, when they do not have to be triggered from parts written in assembly language) they can be implemented as empty inline methods, which get optimized away by the compiler if no aspect binds to them. Another advantage is the inherent support for $1:n$ relationships – handler chaining for shared interrupt sources, for instance, is supported “out of the box”.

We distinguish between **upcall join points** and **transition join points**. The former are the interface that *upcall aspects* bind to; the incarnations of hardware interrupts or threads, for instance, are provided in this realm. Another example is the system initialization handler `hw::hal::init()`, which is invoked during system startup. Upcall join points manifest naturally in a bottom-up development process.

Transition join points, in contrast, mark events that are important for the implementation of system policies. They are typically identified during concern analysis and can be taken directly from the bottom of the cross-cut table. An example we can find there (Table 1) are the already mentioned *user* $\mapsto$ *kernel* transitions, which in CiAO are provided as explict join points `os::krn::enterKernel()` and `os::krn::leaveKernel()`. Other examples include transitions

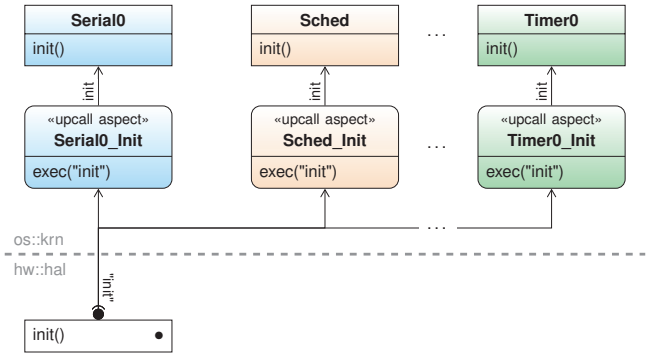


Figure 4: Self-integration of components. Depicted is the CiAO component initialization scheme. Every system component integrates itself into the system initialization handler `hw::hal::init()` by an accompanying `_Init` upcall aspect.

from thread level to interrupt level, or the context switch from one thread to another. These transitions often have *multiple* and *implementation-dependent* sources ($m : n$ relationships); or they occur in fragile, low-level parts of the implementation. By representing them as explicit join points, providers and publishers of transition events can be decoupled.

3.4 Problems Revisited in CiAO

The following two examples from CiAO resemble some of the issues we encountered in PURE and eCos (namely, *hidden concerns* and *missing join points*, see Section 2.3), in the sense that they demonstrate how such problems can be avoided by applying the principles of aspect-aware software development.

3.4.1 Self-Integration of Components

The key to *loose coupling* of policies and components is to provide the necessary explicit join points and then establish *all* bindings by advice. Figure 4 shows this on the example of component initialization: Every system component (which are singletons by definition) has an accompanying `_Init` aspect that gives advice to the system initialization handler `hal::init()` (an explicit join point) to invoke the component's `init()` method at system startup time. Thereby, the startup code does not have to know which components are present in the actual CiAO configuration. Nevertheless this flexibility does not come at a price, as all initialization code gets bound and inlined at compile time. This is not only more efficient than the initialization concept used in PURE (which was based on global instance construction, see Section 2.2.1), it also is a lot more flexible. Component initialization thereby becomes a *visible transition*, which we can *further* influence it by additional aspects: Consider, for instance, an (optional) extension aspect `Serial0Ext` that extends the serial driver from Figure 4 by a task of its own (e.g., for some background protocol handling). Similar to the `ServerSync` aspect in the PURE study (Section 2.2.1), this aspect effectively inserts a new functional dependency between the serial driver and the scheduler; the serial driver now *uses* the scheduler. The consequence for the implementation is that the scheduler has now to be initialized *before* the serial driver. In AspectC++, we can realize this new constraint relatively easy by employing *order*-advice [26]. Additional to the extension of the class `Serial0`, the aspect `Serial0Ext` can specify a partial invocation order for the *foreign* aspects `Sched_Init` and `Serial0_Init` at the join point `execution( "void hw::hal::init()" )`:

```
aspect Serial0Ext {
  ...
  advice execution( "void hw::hal::init()" ): order(
    "Sched_Init", "Serial0_Init" );
};
```

Essentially, the aspect thereby re-establishes a correct functional hierarchy of the system. This is possible because of the application of the principles of *loose coupling* and *visible transitions*.

3.4.2 Self-Integration of Policies

Another common use case for advice-based binding in CiAO is the self-integration of policies. Self-integration of policies is crucial for the aspired decoupling of policies and mechanisms. Most policy implementations induce new interactions between (otherwise unrelated) components. This may, again, lead to new functional dependencies that we also have to deal with. Figure 5 demonstrates self-integration of policies by the example of two variants of the CiAO preemption policy (which, to some degree, resembles the issues we found in eCos, see Section 2.2.2):

Generally, system components report the need for rescheduling (and, thus, potential preemption of the running task) by calling `Sched::setNeedReschedule()`. The actual activation of the scheduler is, however, delayed:

(a) The aspect `Sched_LeaveBinding` in Figure 5.a implements a simple delayed activation policy for a cooperative system; with this policy, preemption is only possible at the return from some system service. Technically, this is realized by binding the scheduler activation (`Sched::reschedule()`) to the explicit tranistion join point `leaveKernel()`, which is guaranteed to be triggered if some thread returns from the kernel.

(b) The aspect `Sched_ASTBinding` in Figure 5.b implements a more sophisticated delayed activation policy for an interruptive system; with this policy, preemption can also be triggered by interrupts. Technically, this is realized by binding the scheduler activation to the function `AST0::ast()`, which is the handler of an *asynchronous system trap*[1] (*AST*). Additionally, the triggering of the AST is bound to `setNeedReschedule()`. The fact that the scheduler is now activated from `AST0::ast()` leads to a new functional dependency, which has the consequence that the kernel now has to be synchronized on AST level. We can, however, easily enforce this constraint with additional pieces of advice that are given by the `Kernel_ASTSync` aspect:

```
aspect Kernel_ASTSync {
  advice execution( "os::krn::enterKernel()" ) : before() {
    AST0::Inst().disable(); // delay scheduling
  }
  advice execution( "os::krn::leaveKernel()" ) : after() {
    AST0::Inst().enable();  // point of rescheduling
  }
};
```

By *visible transitions* and advice-based binding we have achieved a completley *loose coupling* of the scheduler component and the preemption policy in the *implementation*. This makes it very easy to provide numerous variants of either concern an embedded systems engineer can choose from.

4. DISCUSSION

We discuss the combination of AOP and operating systems in general, how our approach can be applied to other system software, and the lessons learned with respect to language and tooling.

[1] An AST is a low-priority interrupt that can be triggered by higher-prority interrupts or the kernel to delay activities, such as scheduling, to a later point in time (e.g., when the kernel is left).

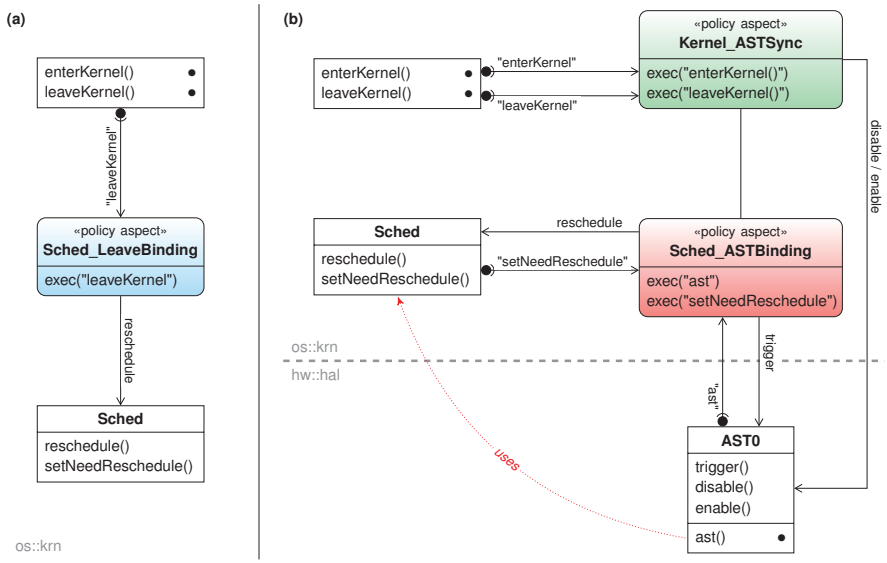


Figure 5: Self-integration of policies. Depicted are two alternatives for the delayed preemption policy in CiAO. **(a)** The aspect `Sched_LeaveBinding` binds to `leaveKernel()` to activate the scheduler when some task leaves the kernel (cooperative system). **(b)** The aspect `Sched_LeaveBinding` binds to the handler of an *asynchronous system trap* (*AST*) to activate the scheduler when all (potentially nested) interrupt handlers have terminated (interruptive system).

4.1 AOP and Operating Systems

4.1.1 Aspects as First-Class Entities

AOP has been facing much critique in the sense that aspects (in contrast to classes) do not represent real *domain concepts,* but (only) “aspects *of programming*”. Steimann details this in [27]: “literally all aspects discussed in the literature are technical in nature: authentication, caching, distribution, logging, persistence, synchronization, transaction management, etc.”

There might be some truth in this for the kind of software Steimann had in mind when writing his paper, but for the domain of system software, we have to clearly rebut this argument: System software *is* very technical in nature, too; the above mentioned “technical” aspects are text-book examples for *the* dominant concerns of system-software development! In the *specification* of AUTOSAR OS [3], for instance, we can find the *requirement* OS093:

> *If interrupts are disabled and any OS services, excluding the interrupt services, are called outside of hook routines, then the operating system shall return* `E_OS_DISABLEDINT`.

This requirement (which maps to the Interrupts disabled concern in Table 1) translates almost “literally” to an AspectC++ aspect:

```
aspect DisabledIntCheck {  // implements OS093
  advice call( pcOSServices() && !pcInterruptServices() )
  && !within( pcHookRoutines() ) : around() {
    if( interruptsDisabled() )
      *tjp->result() = E_OS_DISABLEDINT;
    else
      tjp->proceed();
} };
```

So for our domain, we can assess that aspects lead to a much more natural separation of domain-specific *concepts* – if considered as first-class design elements from the very beginning.

4.1.2 Quantification and Obliviousness

The `DisabledIntCheck` aspect is also a good example for the benefits of *quantification* because of homogeneous cross-cutting. Given that other studies [14] about applying AOP for the fine-grained configuration of system software (in this case embedded databases) came to the conclusion that quantification is “rarely applicable”, these benefits seem to be domain-specific to a certain degree. However, for the implementation of operating system policies, especially architectural ones, quantification clearly creates synergies. For 8 out of the 14 aspects listed in Table 2 this is the case.

With respect to *obliviousness*, the situation is less clear. In [11], Filman and Friedman describe the obliviousness *ideal* of AOP, according to which obliviousness can be a *bidirectional* relationship between components and aspects: The programmers of the base system and the aspect developers can work completely independently of each other. However, in actual applications of AOP, obliviousness is usually understood to be *unidirectional*: The components of the base system are kept oblivious of aspects – at the price that the aspects have to be perfectly aware of the components they affect. This often involves knowledge about certain implementation details, which in turn leads to fragile pointcuts if the component *developers* are kept oblivious of the aspects, too. Furthermore, this approach hits its limits when the base code just does not offer the required join-point shadows. The ambiguity problems we found in eCos are a good example here.

Aspect-aware operating system development moderates these issues by pragmatically considering *obliviousness* and *awareness* as two ends of a continuum: The more oblivious a component should be of the aspects that potentially engage with it, the more aware the aspects have to be of the component – and vice versa. Much of the flexibility and configurability of CiAO stems from the freedom to decide for each relationship about the placement on this continuum.

In our opinion, the advantage of the *advice*-mechanism of AOP is not so much quantification and obliviousness, but *loose coupling*: Essentially, advice inverts the direction in which control-flow relationships are specified. This facilitates the self-integration of the implementation of optional features into the control flows of the base system. Furthermore, advice-based binding is inherently loose – if the addressed join point is not present, the binding is sliently dropped. This property is useful for the implementation of *inter-*

acting optional features, which are difficult to tackle with other decomposition approaches [15].

4.2 Applicability to Other Domains

The presented methodology emerged from experiences in a special domain – highly-configurable system software for resource-constrained embedded systems. Nevertheless, it is at least partly applicable to a wider range of domains. For example, PUMA is a product line of C/C++ code analysis and transformation frameworks [28]. In this project we have not conducted the *concern identification* and *impact analysis* steps, but the principles of *aspect-aware design* and the underlying AspectC++ idioms including the three roles of aspects were applied and turned out to be generic and helpful.

4.3 Language and Tooling – Lessons Learned

4.3.1 AspectC++ – How the Language Is Evolving

When we started with the development of AspectC++ it seemed "natural to use AspectJ as a foundation when creating a set of extensions for the C/C++ language". This led to many similarities between the two languages such as advice code that is anonymous and, thereby, cannot be overridden by a derived aspect or the explicit interface for accessing join-point context information within advice code (thisJoinPoint-API).

However, it turned out that there are more differences between C++ and Java than initially expected, and also our application domain of deeply embedded systems forced us to rethink the language design with resource consumption in mind. In contrast to the beginning, AspectC++ now has a much stronger focus on static typing and language features that can be implemented completely at compile time. Run-time mechanisms such as the dynamic thisJoinPoint-API, which is typically used in combination with run-time reflection, are too expensive and, thus, have been mostly replaced by a static counterpart. For instance, the "join point API" of AspectC++ provides static type information for advice code. As a consequence, multiple variants of the same advice code can be instantiated at compile time, which depend on the matched set of join points. Additionally the advice can use the type information to instantiate C++ templates or even template meta-programs. Thereby, a complex chain of code generation steps can be triggered. It turned out that this combination of aspects and C++ templates is a very powerful mechanism that is a unique feature of AspectC++ [19].

Currently, a complete static introspection mechanism for all program entities – and not only join points – is under development. This will, for instance, allow generic aspects to very efficiently marshall/unmarshall any objects in order to transparently perform remote method invocations or to manage a persistent state. In the context of CiAO this feature shall be used to transparently copy objects between address spaces when isolation is turned on and tasks in different address spaces interact.

Even though AspectC++ is already very useful, we identified the following missing features, which are on the agenda for future enhancements:

Free variables in pointcut expressions. This is a language feature that is already known from LogicAJ [18]. It would significantly enhance the expressiveness of AspectC++ pointcut expressions.

Extensible pointcuts. Self integration of components such as device drivers would be easier if named pointcuts could be extended or composed from collected fragments. For instance, a driver has certain properties: It services interrupts, it handles a block device, and it needs a helper thread. Aspects should be able to affect all components with a specific property. However, the system configuration – including the set of configured drivers – is unknown before compile time. AspectJ 5 users can achieve this goal by exploiting Java 5 annotations. For AspectC++ a similar mechanism shall be integrated.

More control over code generation. When low-level assembler code and AspectC++ are combined it is often necessary to control the code generation very precisely. For instance, in a function or advice that implements a context switch between tasks and that contains inline assembler code, it is crucial to know whether the function will be inlined by the compiler. If the compiler behaves unexpectedly, a machine crash will be unavoidable.

Non join points. Some parts of the CiAO operating system should simply be guaranteed to never be touched by any aspect. We aim at providing mechanisms to specify these parts in a modular manner and a weaver extension that obeys these rules.

4.3.2 User Experience – AOP for "Hackers"

More than a dozen master students were involved in the development of PURE, the aspectized version of eCos, and CiAO, and contributed a significant amount of the aspect code to these systems. All of them were advanced C/C++ hackers, the majority already had some experience in low-level kernel programming, and all of them carried on with R&D in the domain of low-level system software after finishing their studies. So, to a certain degree this group represents the typical "kernel hacker", whose take on AOP might be interesting to the AOSD community. While we have not evaluated this in a systematic way, we nevertheless observed some recurring peculiarities:

AOP semantics is generally easy to grasp. To our (pleasant) surprise, the students generally had, after a brief introduction into the topic (a three hour lecture plus a "toy" exercise), little to no problems in understanding AOP concepts, the AspectC++ language, and the particularities of its application to embedded systems. They grasped the CiAO development idioms and application patterns by examining the existing code and were quickly able to contribute their own aspects.

Technical side effects of aspect weaving are more challenging. In theory, aspect weaving should be a transparent process, but in practice it is not – due to technical side effects. A frequent and always challenging issue, for instance, was the understanding and resolving of *`#include` cycles*. Such a cycle appears if two header files (indirectly) `#include` each other, which in most cases leads to uncompilable code. Unexpected `#include` cycles are a tough problem for any larger C/C++ project. The point is that they appear *a lot* more frequently with aspect weaving: An aspect that itself `#includes` some external module (a property that holds for any nontrivial aspect) thereby also contributes to the list of `#include` files of the modules it affects in the weaving process, which often results in `#include` cycles that are very hard to hunt down. As a consequence, we have improved the AspectC++ weaver to detect and report `#include` cycles caused by aspects already at weaving time. While this has certainly improved on the situation, it is still up to the developer to resolve the conflict (e.g., by means of forward declarations or by splitting larger aspects into smaller pieces).

"Hackers hate IDEs." Even though all students at some point ran into difficulties with respect to join-point tracking, it turned out to be more than difficult to convince them to use the AspectC++ plug-in for ECLIPSE (ACDT), which provides features (such as join-point visualization) for exactly this kind of problem. Even the majority of students working on CiAO – who *had* to use ECLIPSE anyway to configure the operating system – did not use it for *anything* else. They considered it to be "too clumsy" compared to the shell and their favorite VIM editor, and preferred hunting for join-point mismatches by analyzing the woven source code or by GREP'ing

through the (XML-based) join-point repository that AC++ generates for the ECLIPSE plug-in. We have learned from this that (even in the case of relatively young students) tool support has to fit the – domain-specific – habits of the developers to get accepted. As a consequence, we are now working on a more generic interface to the join-point repository and a set of command-line tools to query and analyze it in a "no-frills" fashion.

5. FURTHER RELATED WORK

There are several other research projects that investigate the applicability of aspects in the context of operating systems. Among the first was the α-kernel project [8], in which the evolution of four scattered OS concern implementations (namely: prefetching, disk quotas, blocking, and page daemon activation) between versions 2 and 4 of the FreeBSD kernel was analyzed retroactively. The results show that an aspect-oriented implementation would have led to significantly *better evolvability* of these concerns.

C4 [12, 24] is an example for a special-purpose AOP-inspired language. It is intended for the application of kernel patches in Linux. Other related work concentrates on dynamic aspect weaving as a means for run-time adaptation of operating system kernels: TOSKANA provides an infrastructure for the dynamic extension of the FreeBSD kernel by aspects [10]; KLASY is used for aspect-based dynamic instrumentation in Linux [29].

All of these studies demonstrate that there are good cases for aspects in system software. However, the work of ÅBERG in Linux [1] and our own work on eCos [21] show that a useful application of AOP to existing operating systems requires additional AOP expressivity that results in run-time overheads (e.g., temporal logic or dynamic instrumentation).

6. SUMMARY AND CONCLUSIONS

The CiAO project contributes a large-scale case study for the application of aspect technology in the domain of system software. From a systems researcher's perspective, the properties (such as code size, performance, and especially configurability) of the resulting systems are convincing [20, 21]. This paper has focused on the development methodology, which evolved over years. Two main insights can be learned: (1) Operating systems for the domain of resource-constrained embedded systems have to be highly configurable. Our analysis of the AUTOSAR OS specification revealed that these effects can already be found in the requirements; they are an *inherent phenomenon* of complex systems. (2) AOP is very well suited for the design and implementation of such systems under the premise that it is applied with the aspect awareness principles in mind. This paper has shown how this *aspect awareness* can be put into practice.

7. REFERENCES

[1] ÅBERG, R. A., LAWALL, J. L., SÜDHOLT, M., MULLER, G., AND MEUR, A.-F. L. On the automatic evolution of an OS kernel using temporal logic and AOP. In *ASE '03* (Mar. 2003), IEEE, pp. 196–204.

[2] ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. Adding trace matching with free variables to AspectJ. In *OOPSLA '05* (Oct. 2005), ACM, pp. 345–364.

[3] AUTOSAR. Requirements on operating system (version 2.0.1). Tech. rep., Automotive Open System Architecture GbR, June 2006.

[4] AUTOSAR. Specification of operating system (version 2.0.1). Tech. rep., Automotive Open System Architecture GbR, June 2006.

[5] BEUCHE, D., FRÖHLICH, A. A., MEYER, R., PAPAJEWSKI, H., SCHÖN, F., SCHRÖDER-PREIKSCHAT, W., SPINCZYK, O., AND SPINCZYK, U. On architecture transparency in operating systems. In *9th SIGOPS European Workshop* (Sept. 2000), ACM, pp. 147–152.

[6] BEUCHE, D., GUERROUAT, A., PAPAJEWSKI, H., SCHRÖDER-PREIKSCHAT, W., SPINCZYK, O., AND SPINCZYK, U. The PURE family of object-oriented operating systems for deeply embedded systems. In *ISORC '99* (May 1999), pp. 45–53.

[7] CAI, Y., AND SULLIVAN, K. J. Modularity analysis of logical design models. In *ASE '06* (2006), IEEE, pp. 91–102.

[8] COADY, Y., AND KICZALES, G. Back to the future: A retroactive study of aspect evolution in operating system code. In *AOSD '03* (Mar. 2003), ACM, pp. 50–59.

[9] eCos homepage. `http://ecos.sourceware.org/`.

[10] ENGEL, M., AND FREISLEBEN, B. TOSKANA: a toolkit for operating system kernel aspects. In *TAOSD II* (2006), no. 4242 in LNCS, Springer, pp. 182–226.

[11] FILMAN, R. E., AND FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA '00 Workshop on Advanced SoC* (Oct. 2000).

[12] FIUCZYNSKI, M., GRIMM, R., COADY, Y., AND WALKER, D. patch(1) considered harmful. In *HotOS '05* (2005), USENIX.

[13] HABERMANN, A. N., FLON, L., AND COOPRIDER, L. W. Modularization and hierarchy in a family of operating systems. *CACM 19*, 5 (1976), 266–272.

[14] KÄSTNER, C., APEL, S., AND BATORY, D. A case study implementing features using AspectJ. In *SPLC '07* (2007), IEEE, pp. 223–232.

[15] KÄSTNER, C., APEL, S., UR RAHMAN, S. S., ROSENMÜLLER, M., BATORY, D., , AND SAAKE, G. On the impact of the optional feature problem: Analysis and case studies. In *SPLC '09* (2009), IEEE.

[16] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP '97* (June 1997), vol. 1241 of *LNCS*, Springer, pp. 220–242.

[17] KLEIMAN, S., AND EYKHOLT, J. Interrupts as threads. *ACM OSR 29*, 2 (Apr. 1995), 21–26.

[18] KNIESEL, G., AND RHO, T. A definition, overview and taxonomy of generic aspect languages. *L'Objet, Special Issue on Aspect-Oriented Software Development 11*, 2–3 (Sept. 2006), 9–39.

[19] LOHMANN, D., BLASCHKE, G., AND SPINCZYK, O. Generic advice: On the combination of AOP with generative programming in AspectC++. In *GPCE '04* (Oct. 2004), vol. 3286 of *LNCS*, Springer, pp. 55–74.

[20] LOHMANN, D., HOFER, W., SCHRÖDER-PREIKSCHAT, W., STREICHER, J., AND SPINCZYK, O. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *USENIX '09* (June 2009), USENIX, pp. 215–228.

[21] LOHMANN, D., SCHELER, F., TARTLER, R., SPINCZYK, O., AND SCHRÖDER-PREIKSCHAT, W. A quantitative analysis of aspects in the eCos kernel. In *EuroSys '06* (Apr. 2006), ACM, pp. 191–204.

[22] MASSA, A. *Embedded Software Development with eCos*. New Riders, 2002.

[23] PARNAS, D. L. Some hypothesis about the "uses" hierarchy for operating systems. Tech. rep., TH Darmstadt, Fachbereich Informatik, 1976.

[24] REYNOLDS, A., FIUCZYNSKI, M. E., AND GRIMM, R. On the feasibility of an AOSD approach to Linux kernel extensions. In *AOSD-ACP4IS '08* (Mar. 2008), ACM, pp. 1–7.

[25] SPINCZYK, O., AND LOHMANN, D. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS European Workshop* (Sept. 2004), ACM, pp. 188–192.

[26] SPINCZYK, O., AND LOHMANN, D. The design and implementation of AspectC++. *Knowledge-Based Systems 20*, 7 (2007), 636–651.

[27] STEIMANN, F. Domain models are aspect free. In *MoDELS '05* (2005), vol. 3713 of *LNCS*, Springer, pp. 171–185.

[28] URBAN, M., LOHMANN, D., AND SPINCZYK, O. PUMA: An aspect-oriented code analysis and manipulation framework for C and C++. In *TAOSD VIII* (2011), no. 6580 in LNCS, Springer. To appear.

[29] YANAGISAWA, Y., KOURAI, K., CHIBA, S., AND ISHIKAWA, R. A dynamic aspect-oriented system for OS kernels. In *GPCE '06* (Oct. 2006), ACM, pp. 69–78.

CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack

Christoph Borchert[a] Daniel Lohmann[b] Olaf Spinczyk[a]
christoph.borchert@tu-dortmund.de lohmann@cs.fau.de
olaf.spinczyk@tu-dortmund.de
[a]Technische Universität Dortmund
[b]Friedrich-Alexander-Universität Erlangen-Nürnberg

ABSTRACT

Internet protocols are constantly gaining relevance for the domain of mobile and embedded systems. However, building complex network protocol stacks for small resource-constrained devices is more than just porting a reference implementation. Due to the cost pressure in this area especially the memory footprint has to be minimized. Therefore, embedded TCP/IP implementations tend to be statically configurable with respect to the concrete application scenario. This paper describes our software engineering approach for building CiAO/IP – a tailorable TCP/IP stack for small embedded systems, which pushes the limits of static configurability while retaining source code maintainability. Our evaluation results show that CiAO/IP thereby outperforms both *lwIP* and *uIP* in terms of code size (up to 90% less than *uIP*), throughput (up to 20% higher than *lwIP*), energy consumption (at least 40% lower than *uIP*) and, most importantly, tailorability.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*TCP/IP*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and Interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

AOP, Aspect-Oriented Programming, AspectC++, Operating Systems, Embedded Systems, TCP/IP, Internet Protocol, Network Protocol Stacks

1. INTRODUCTION

Communication has become an integral part of today's computer systems. Every personal computer is equipped with a built-in network interface and there is a clear trend towards smart (computerized) devices, which very often have the ability to interact with the Internet, such as modern TVs. Hence, small and, in the future, even smaller devices need to implement the TCP/IP protocol suite, which is the de-facto standard for data communication and by definition used in the Internet in order to achieve *interoperability*.

However, there is still a huge gap between vision and reality: Science and industry are thinking about *Service-Oriented Architectures (SOA) for embedded devices* [3], small devices being connected with the Internet using a lightweight middleware such as *Universal Plug'n'Play (UPnP)*[1] or its successor *Devices Profile for Web Services (DPWS)*[2]. Both are based on TCP/IP. On the other hand, the resource contraints of embedded microcontroller units often forbid to use the software-intensive Internet protocols, which in practice leads to hybrid systems and a "reinvention of the wheel".

Whether a full-featured TCP/IP stack is well suited for deeply embedded systems with tight resource constraints, has already been debated in great detail [11, 13]. The purpose of this paper is *not* to give the final answer. However, we can show *how* the TCP/IP protocol suite could be used in nearly any device, despite of the enormous diversity in hardware platforms and application requirements.

Today, many TCP/IP implementations are available. Most of them were designed as a component of a PC or server operating system – such as BSD, Linux, or Windows – and are quite resource intensive.

Common examples for TCP/IP stacks for resource-constrained embedded systems are *micro-IP (uIP)* and *lightweight-IP (lwIP)* [8]. uIP proved that a full TCP/IP implementation fits even into a small 8-bit microcontroller unit by leaving out all optional features, wheras lwIP provides better performance at the cost of higher memory consumption. It is remarkable that both, uIP and lwIP, stem from the same author and have been presented together in [8]. This underlines that a "one size fits all" TCP/IP implementation is not well suited for embedded systems. However, is a "two sizes fit all" solution so much better?

1.1 Contribution and Outline

In this paper we present the design approach of the CiAO/IP[3] stack and a comparison of CiAO/IP with *uIP* and *lwIP*. Our approach is to construct CiAO/IP as a highly configurable *Software Product Line* [26] that provides for very tailored stack implementations for embedded applications. In order to keep the highly configurable code maintainable, *Aspect-Oriented Software Development* [18] has been exercised not only for the implementation but already during the design phase.

In summary, our contribution is to show that *applying* aspect-oriented design and implementation techniques to a new domain – networked embedded systems – is promising from a code mainte-

MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.

[1] `http://www.upnp.org/`
[2] `http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01/`
[3] The source code is available under the terms of the GNU GPL at: `http://ess.cs.tu-dortmund.de/Software/CiAO-IP`

nance perspective without incurring energy and performance overhead.

Moreover, the fine-grained configurability of CiAO/IP allows developers to tailor the IP stack at compile time to their actual usage scenario, so that memory footprint, performance, and energy consumption are even improved over the state-of-the-art. We regard that knowledge of *how to build* network stacks for small-footprint devices as a useful contribution that deserves to be communicated more broadly.

The outline of this paper is as follows:

- We show that configurability is an important requirement on system software for networked embedded systems, which is not sufficiently fulfilled by other state-of-the-art IP stacks in this domain (Section 2). Despite the tension between source code maintainability and fine-grained configurability, the paper shows that CiAO/IP achieves *both* objectives (Section 6).
- We apply our unique design methodology, which is based on several years of experience with the aspect-oriented operating system CiAO[4] [22], to the domain of the Internet Protocol stack. Starting from requirements with explicit variability we step-by-step derive an aspect-oriented software design. Reusable programming idioms help to turn the design into code (Section 3 and Section 4).
- We show how the methodology led to our CiAO/IP design and implementation. Thereby, we document the structure of CiAO/IP, which is the first aspect-oriented TCP/IP stack we are aware of.
- By comparing CiAO/IP with *uIP* and *lwIP* we prove the implementation’s scalability (8 to 64-bit systems) and high configurability (in terms of functional features), which we could achieve without any drawbacks in terms of code size, performance, and energy consumption (Section 5).

2. CONFIGURABILITY OF IP STACKS

The development of an IP stack for small mobile and embedded devices is more than just porting a reference implementation.

2.1 Diversity of Platforms and Requirements

The first challenge is that two embedded systems rarely look the same. The microcontroller market is heterogeneous and offers many entirely different devices. Simple control tasks can be performed by small 8-bit CPUs, which are still dominating the market in terms of shipped units [34]. At the other end of the spectrum, complex signal-processing often requires high-performance 32-bit or even 64-bit devices. The memory sizes are as diverse as their layouts, but share the commonality that they define hard constraints that a software developer must fulfill. When it comes down to networking, the number of possible interfaces seems to be vast. Both wired and wireless technologies are available in many different kinds, such as Ethernet, wireless LAN, GSM, and IEEE 802.15.x.

System software that should support and cope with all this variability has to be exceptionally *flexible* and *portable*, and must be *minimalistic* in order to fit into the tiny memories of even the most miniaturized devices – often only a few KiB of RAM!

The second challenge is that the requirements on system software vary a lot in different embedded systems. Depending on the application scenario, the operating system has to support best-effort scheduling for concurrent tasks, while other scenarios require strict spacial and temporal isolation of each task in order to meet certain dependability properties. The requirements on network stacks are diverse as well: Multimedia applications, such as *Voice over IP*, call for low latencies and guaranteed quality of service. Video streaming, on the other hand, requires high throughput and is not tied to low latency constraints. Furthermore, systems differ in the protocols they use. Information exchange such as E-Mail requires a reliable transport protocol that confirms data reception, whereas best-effort data delivery is sufficient for sensor applications. In short: the functionality required from the network stack strongly depends on the concrete application.

The big challenge from a software engineer’s point of view is to overcome this multi-dimensional diversity and to design software that fits into all those situations in an optimal way. Static (compile-time) *configurability* is the key to support a broad variety of hardware platforms and application requirements without sacrificing efficiency.

The CiAO/IP stack aims at a very high degree of configurability to tailor the stack for the chosen hardware platform and application scenario. The goal is to be able to leave everything off that is *not* needed – in order to achieve minimal resource consumption.

2.2 State of the Art

Table 1 gives an overview on the features provided by our implementation of CiAO/IP and those of uIP and lwIP[5] [8].

A configurable feature that can be omitted if not needed, is denoted by ✓. This kind of feature contributes to functional scalability. A fixed feature, which is always present and not removable by the end-user, hinders scalability and is denoted by ∗. Unimplemented features of each IP stack are shown as whitespace.

uIP has by far the most reduced feature set. It supports solely a single networking interface, manages a single buffer for exactly one packet and provides no support for an operating system. The variability of uIP is very limited, too, for instance the protocols ICMP and TCP are fixed[6]. Furthermore, uIP does not implement a sliding window for TCP, so that poor throughput can be expected. Thus, uIP is only suited for application scenarios on deeply embedded devices that require TCP connectivity .

lwIP provides rich functionality in terms of features. As shown in Table 1, lwIP implements almost every feature that we consider for this comparison. Nevertheless, the configurability of networking protocols is coarse-grained. For example, the feature TCP includes client *and* server functionality (often only one side is needed), a sliding window (again, only for *both* Tx and Rx) and a bunch of common optimization algorithms, such as Silly Window Syndrome (SWS) avoidance, Round-Trip Time estimation and congestion control. None of these optional protocol features is actually optional in the lwIP implementation – TCP can only be included and excluded at a whole. Because of this fixed set of rich functionality, which might be welcome on powerful devices, lwIP does not scale to devices with tighter resource constraints. The large number of fixed features sacrifices scalability.

The CiAO/IP implementation differs substantially from the preceding ones: In CiAO/IP *every* feature is optional and, thus, configurable by the application developer. We abstain from fixed features in favor of scalability. Thereby, the developer has full control and does not have to pay for features that are not needed. The overall functionality of CiAO/IP is comparable to lwIP, but configurability is much more fine-grained.

[4] CiAO is Aspect-Oriented

[5] versions 1.0 and 1.32 respectively

[6] Contiki 2.5 (`http://www.contiki-os.org/`) includes an updated version of uIP that provides configurability of TCP and ICMP by `#ifdef` directives.

Feature	uIP	lwIP	CiAO/IP
Multiple Networking Interfaces		*	✓
Checksum Offloading per Interface			✓
Multiple Connections (Concurrency)	*	*	✓
Buffers per Connection (Isolation)			✓
Operating System Support		✓	✓
IPv4	✓	✓	✓
IPv4 Tx	*	*	✓
IPv4 Rx	*	*	✓
IPv4 Fragment Reassembly	✓	✓	✓
IPv4 Fragmentation		✓	
IPv6	(✓)[a]	(✓)[a]	(✓)[b]
ARP	✓	✓	✓
ARP Reply	*	*	✓
ARP Request	*	*	✓
ARP Cache Timeout	*	*	✓
Static ARP Cache Entries			✓
ICMP	*	✓	✓
UDP	✓	✓	✓
UDP Tx	*	*	✓
UDP Rx	*	*	✓
UDP Checksumming	✓	✓	✓
TCP	*	✓	✓
Client (Connect)	✓	*	✓
Server (Listen)	*	*	✓
Sliding Window (Tx)		*	✓
Sliding Window (Rx)		*	✓
Avoid Silly Window Syndrome (Tx)		*	✓
Avoid Silly Window Syndrome (Rx)		*	✓
Round-Trip Time Estimation		*	✓
Congestion Control (Slow-Start)		*	✓
TCP Urgent Data	✓	*	
Limit Excessive Retransmissions	*	*	✓
MSS Option	*	*	✓
Timestamp Option		✓	

[a] provided by a different code base, no dual stack
[b] under development

Table 1: Features provided by uIP, lwIP and CiAO/IP

2.3 A Software Engineering Challenge

Traditionally, configurability in system software has been expressed by the use of the C preprocessor. It supports textual substitution by macro expansion and conditional compilation by means of `#ifdef` directives. The preprocessor transforms the source code before it is parsed by the C compiler. A typical idiom is to represent all configurable features as macros in common header file, which controls the source code adaptation process of the system.

Expressing software variability in that way has been heavily criticized as error-prone, unreadable and unmaintainable [29]. A recent study of Linux kernel code shows that `#ifdef`-based configuration led to hundreds of bugs [33]. There are two main problems:

1. Dependencies between features are represented on the source-code level: This leads to a mix of abstraction levels within the code, because for each particular feature the *high-level* requirements and dependencies become intermixed with *low-level* implementation details.

2. The implementation of features often "crosscuts" various modules (functions, files, etc.): For many configurable features, the implementation is "scattered" over many modules and "tangled" with the implementation of other features. The consequence is that configurable features lead to code that is bloated with lots of macros and `#ifdef` directives. The whole code base becomes hard to understand and maintain.

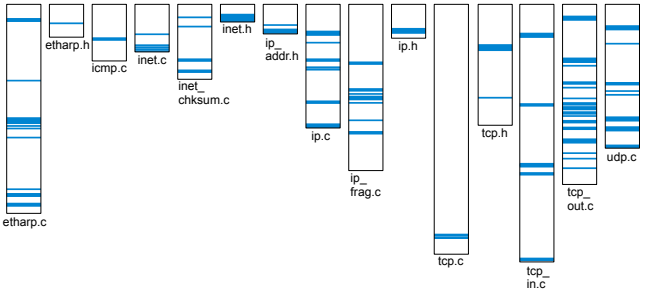


Figure 1: Scattered implementation of byte order conversion in lwIP. Each bar represents a source code file of lwIP; a horizontal line indicates the use of a conversion macro (`htons, ntohs,...`) in the respective source line.

Figure 1 illustrates these problems on the example of byte ordering in lwIP: In lwIP (as in uIP, Linux, and BSD) the conversion of 16-bit and 32-bit words to the network byte order is performed by dedicated preprocessor macros (`htons, ntohs, htonl, ntohl`). Access to network protocol headers is coupled to these macros – their use is compulsory to ensure data correctness. Hence, these macros appear 210 times in 15 files of the analyzed lwIP configuration. This implementation of the byte order conversion is tedious and error-prone, as a single forgotten macro call can already lead to data corruption in terms of swapped bytes.

The conclusion is that traditional preprocessor-based configuration does not scale: Software with lots of configurable features almost automatically leads to a maintenance nightmare [33]. This is a software engineering challenge. The problems can only be solved by using a configurability-oriented design approach combined with suitable programming language abstractions.

3. DESIGN APPROACH

The CiAO/IP stack is a generic platform for IP networking software. It supports the automatic *derivation* of custom IP implementations that fulfill application-specific constraints. An application developer just has to specify the relevant constraints by selecting software features in a graphical user interface. After that, he instantly gets a highly optimized IP implementation that meets his requirements. Even architectural properties, such as simultaneous support for different link layers, are configurable and do not cause any overhead if not needed.

CiAO/IP has been developed as an aspect-oriented software product line. A software product line is "a *family* of systems in a domain, rather than a single system" [15]. Due to systematic *reuse*, the software is better tested and, thus, less error-prone. From an economic perspective, the effort for maintaining a software product-line is much less than for several individual products.

One of the key ideas from the product-line engineering community is to strictly separate the so called *problem space* from the *solution space*. This addresses and solves the first problem identified in Section 2.3, namely the mix of abstraction levels in the source code. Typically a problem-space model of a product line describes the common and variable features and their dependencies in an abstract and problem-oriented manner. The problem-space model is independent from the product line's implementation. The solution space is the set of code artifacts that form the implementation.

Aspect-Oriented Programming is a paradigm that provides pro-

gramming language means to modularize the implementation of crosscutting concerns [18]. This allows us to avoid the second problem from Section 2.3. As a result the highly configurable code of CiAO/IP is completely free of `#ifdef` directives.

In the following, we will first explain feature modeling, which is the methodology that we used to model the problem space of the CiAO/IP product line. Then we briefly introduce the concept of Aspect-Oriented Programming.

3.1 Feature Modeling

Feature modeling [15] is a mature technique for formalizing variability. The idea is to identify and document requirements in terms of *features*. A feature is a requirement of a specification or an informal demand of a stakeholder. Each feature can be either mandatory or optional – the latter case is where variability comes into play. Furthermore, features may have restrictions and dependencies among each other. A feature model encompasses all requirements, both mandatory and optional, that are imposed for the product being developed. The success of a fine-grained configurable software product-line relies mainly on high-quality feature models: It is crucial to have detailed information about the expected variability in advance in order to guide the subsequent design and implementation process.

We believe that the methodology of feature modeling is well suited for analyzing the complex requirements that IP networking software must fulfill. As a starting point for a feature model of the Internet TCP/IP protocol suite, we use the official specification published by the Internet Engineering Task Force (IETF). There are several RFC documents which define requirements for each protocol. In RFC1122 [6], the most important requirements for Internet hosts are summed up and categorized into three different kinds: *MUST*, *SHOULD*, and *MAY*. These capitalized verbs determine the significance of each particular requirement; in terms of a software product line *MUST* requirements map to mandatory features, whereas *SHOULD* and *MAY* requirements describe optional features.

Figure 2 shows a simplified feature diagram of the Internet Protocol. A feature diagram graphically represents the variability of a feature model by using a tree structure. Each node represents a feature that depends on all of its ancestors. A filled circle at the lower end of an edge indicates a mandatory feature (c.f. RFC *MUST*), whereas a nonfilled circle describes an optional one (c.f. RFC *SHOULD* or *MAY*). Cumulative features, of which a least one must be present, are grouped together by a filled arch at the upper ends of their edges. As shown in Figure 2, the Internet Protocol can be implemented as version 4 (IPv4) as well as 6 (IPv6). For each version, there is a distinction between sending and receiving, and each particular version requires at least one of them. Additionally, the Internet Protocol is extensible via *IPv4/IPv6 Options*, which are in fact optional features.

The development of feature models for each protocol of the TCP/IP protocol suite has been necessary for the sequel of this paper. A more complex example is given in Figure 3, which contains a still heavily simplified feature diagram of TCP, the most complex protocol we dealt with.

Compared to the RFC, the feature diagram is far more fine-grained. For instance, the feature of a *Sliding Window* is considered optional, because the minimal size of that window is not specified, which makes it effectively optional. A window size of exactly one segment is equivalent to a nonexisting Sliding Window.

Hence, feature modeling is a challenging discipline, which relies on the knowledge and farsightedness of experts in the analyzed domain in order to grasp all hidden features, which are not explicitly

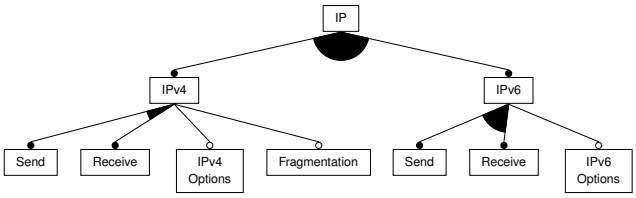


Figure 2: Feature Diagram of the Internet Protocol

stated in the specification, but relevant to stakeholders. On the other hand, no expertise in programming languages or specific paradigms is needed. This part can be left to solution space experts.

3.2 Aspect-Oriented Programming

Traditionally, the developer of an Internet Protocol stack decides upfront which features to implement, so that a particular system consists of a fixed subset of the RFC1122 requirements. Our idea is to postpone this decision as far as possible to preserve this freedom for the actual user of the protocol stack.

The design of a software architecture that adheres to the variability of feature models is a nontrivial task. We propose *Aspect-Oriented Programming (AOP)* [18] as a solution for this problem in order to avoid the already discussed pitfalls of the C preprocessor. AOP supports the decomposition of a system into orthogonal modules by featuring *implicit invocation*. In AOP, an *aspect* contains one or more pieces of *advice*, that intercept the control flow and extend the underlying types. A piece of advice targets several *join points*, which are either locations in the dynamic control flow or part of the static program structure, where arbitrary code can be invoked. Join points are described declaratively via *pointcut expressions* in a textual form.

AspectC++ [30], which has been developed by our group over the last 10 years, extends the C++ programming language by AOP mechanisms. It consists of an *aspect weaver* that processes ordinary C++ code and aspect code, and weaves the latter type of code into the C++ files at the relevant locations (i.e., join points). Thus, the advice code is *inlined* into existing C++ code and produces no overhead compared to an implementation by hand (see [23]). In constrast to C preprocessor directives, the AspectC++ language elements are fully integrated into the syntax and semantics of C++.

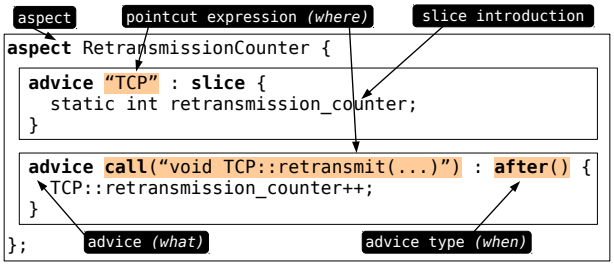


Figure 4: Syntax of AspectC++

Figure 4 outlines the essential syntactic elements of AspectC++ by taking the example of a retransmission counter for TCP. The given aspect implements the counter in a modular way by two pieces of advice. The first piece of advice extends the class `TCP` by an `int` member to store the counter's value. This way, the static program structure is modified by introducing additional class members, which is called a *slice introduction*. The second piece of advice targets the function `TCP::retransmit(...)` and implicitly

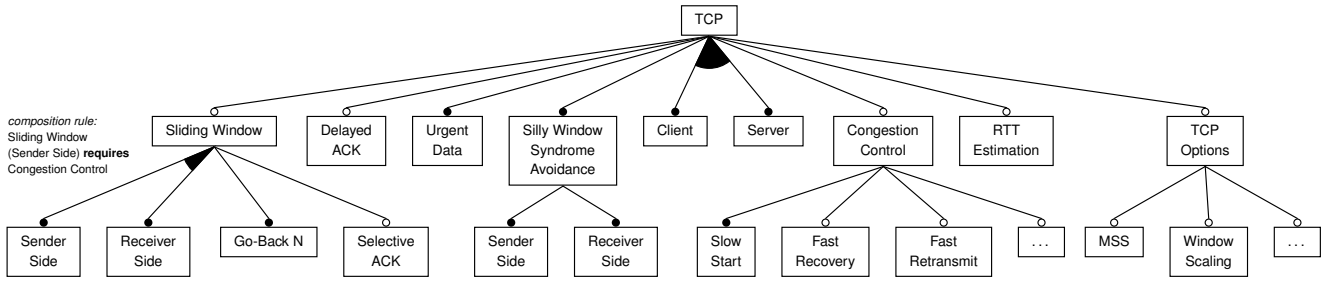


Figure 3: Simplified Feature Diagram of TCP

increases the counter *after* each *call* to that function. The ellipsis of the function's arguments in the pointcut expression ensures that in case of an overloaded function all overloads are matched regardless their arguments.

Furthermore, multiple pointcut expressions can be combined using the algebra of sets. Besides *after* advice, *before* and *around* (the latter replaces the original behavior of the join point) advice can be used to specify when advice code shall be invoked.

In the following section, we show that AOP provides a rich toolkit for the design and implementation of a highly modular networking stack. Even crosscutting features, which are conceptually scattered over many different software modules, can be developed as separate modules by using AOP. We provide a design method for deriving a mapping of features to aspects and ordinary C++ classes and answer the emerging question: Which features should be turned into aspects?

4. INSIDE THE IP STACK

The previous section outlined basic methods we use for the design and implementation of fine-grained configurable networking software. Feature modeling reveals all features that have to be considered during the following design process. Before the actual design of the software system, the interactions of all features and their impact on the system must be analyzed. The system designer needs to gather this information in order to design the fundamental software architecture. Section 4.1 covers the necessary analysis that results in a reference software architecture, which is elaborated in Section 4.4. In between, general design principles and idioms are introduced that guide the design process.

4.1 Concern Impact Analysis

Concern Impact Analysis, which originates from the design of automotive operating systems [21], is a method for the investigation of interrelation between features. Both, concerns that are part of the system's specification (e.g., RFCs) and internal concerns serve as input for the analysis. Internal concerns are inherent properties of the system that are not documented in the specification. For example, the network byte order is not discussed in RFC1122 although it is crucial for interoperability. The starting point for the following analysis is a set of both explicit concerns (i.e., features) and internal concerns. Since it is hardly possible to be aware of all internal concerns in advance, missing ones can be added iteratively. Networking software in general consists of

1. an *API*, that provides accessible system services to applications,

2. *connection state* to distinguish between different connections and their actual usage,

	Protocols				Options			Internal	
	ARP	IP	UDP	TCP	UDP Checksum	TCP Sliding Window	TCP MSS Option	Checksum Algorithm	Byte Order
`send()`		⊕	⊛	⊛	◐	●	◐		◐
`receive()`		⊕	⊛	⊛	◑	●	◑		◑
`connect()`				⊕		◐	◐		
`listen()`				⊕		◑	◑		
`close()`				⊕					
`bind()`			⊕						
`unbind()`			⊕						
`set_ipv4_addr()`		⊕							◐
`set_dst_port()`			⊕	⊕					◐
`set_src_port()`			⊕	⊕					◐
Packet Buffers	⊛	⊛	⊛	⊛		⊛			
ARP Cache	⊕								⊛
IP Addresses		⊕							⊛
Port Numbers			⊕	⊕					⊛
TCP States				⊕		⊛	⊛		
Receive IRQ	◑	◑	◑	◑					
Timeout	◑			◑		◑			
TCP/UDP ⇄ IP					◑			◑	
IP ⇄ Data Link	◐							◐	
Initialization		◑							

Table 2: Impact of configurable concerns on configurable IP networking software. Impact of concerns (columns) on the API, internal states, and events (rows).

3. and external events or rather internal *state transitions* that alter the system's behavior.

These three ingredients are of special interest because they emphasize places where various configurable concerns can interfere. An arrangement of all identified configurable concerns over these three ingredients leads to a comprehensive *crosscut table matrix* that reveals critical join points.

Table 2 shows an excerpt of the crosscut table matrix that we prepared for IP networking software. The columns list a few of the configurable concerns, which are themselves grouped into sets of networking *protocols*, *options* that extend protocols, and *internal* concerns. The aforementioned three categories (API, state, and transitions) form the rows of Table 2. Each category is separated by a horizontal line and covers a few relevant examples.

After the columns and rows are identified, the crosscut table matrix has to be populated. The influence of each concern on each item (row) is analyzed and classified into mainly three different types of impact:

1. **Introduction** of system services or states (static structure). This kind of impact implies the introduction of basic functionality that is added to the system and typically becomes visible

in the system's API. This fundamental type of influence provides join points for other concerns, denoted by a ⊕ sign in the crosscut table matrix.

2. **Extension** of existing system services or states (static structure). Some concerns modify already existing API services or existing system states. This less invasive type of impact, compared to the preceding one, is visually denoted by a ⊛.

3. **Modification** of the run-time behavior (dynamic structure). Concerns may have impact on the execution of API functions, the occurrence of external events or internal state transitions. The actual modification can take place before / after / around the event, as denoted by ◐ / ◑ / ●.

The crosscut table matrix provides a comprehensive overview of the concerns present in the system and their (subtle) relationships. It enables the developer to make informed decisions in the subsequent architecture design process – in which the concerns ultimately have to be mapped to classes and aspects.

4.2 Mapping to Classes and Aspects

As a driver for this process, the crosscut table matrix can be read in two directions:

A **horizontal analysis** focuses on a technical element, that is a state or service provided by the API for which it yields the contributing and dependent concerns. Once a state or service is introduced by a concern, each modifying concern (denoted by a ⊛) explicitly "*uses*" the introducing one. For example, the concerns *UDP* and *TCP* directly "use" *IP* – they built on the respective `send()` and `receive()` primitives, as observable in the first two rows of Table 2. With this information, the system designer is able to develop a *concern hierarchy* [21] by arranging the concerns according to their "uses" relationships. Figure 5 summarizes the concern hierarchy that encompasses the subset of concerns used in the preceding crosscut table matrix[7].

A **vertical analysis** provides more detailed information about a conceptual concern. Basic concerns can be identified by at least one introduction (⊕), for instance *ARP* in the first column. A new state, namely the *ARP Cache*, is introduced by this concern. Crosscutting concerns can easily be spotted by encountering multiple modifications (◐ / ◑ / ●) for a given concern, such as the *Byte Order* in the last column, which crosscuts several API functions and states.

The information from both analyses can then be used to achieve an initial mapping of concerns to classes and aspects by some simple rules of thumb:

- A *basic concern*, such as *ARP*, *IP*, *UDP*, and *TCP*, is mapped to a class.
- If a basic concern "uses" another basic concern, it becomes a derived class. For example, *UDP* and *TCP* both will be derived from *IP* (see Figure 5).
- A crosscutting concern, such as *Byte Order* or *Checksum Algorithm*, becomes an aspect.
- A configuration option, such as *TCP Sliding Window* or *UDP Checksum*, is also often crosscutting and thus becomes an aspect.

Of course, these rules of thumb only provide an initial software structure, that has to be refined in the subsequent design process, in which usually additional aspects and classes are employed to implement the binding and configurability of the identified concerns. In order to ease the following fine-grained design process, we propose design principles that aid the system designer to make "right" design decisions.

[7] Once again, all concerns for IP networking software do not fit into a single comprehensive diagram.

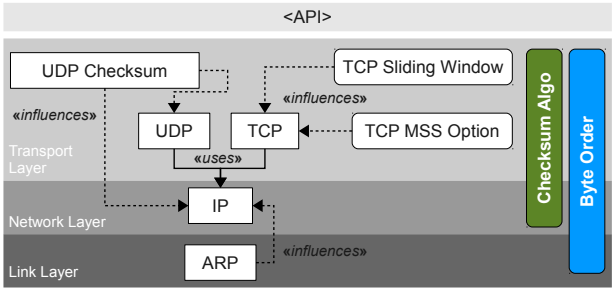


Figure 5: Layered concern hierarchy. Rectangles denote basic concerns, rounded boxes indicate crosscutting concerns. Concerns have "*uses*" and "*influence*" relationships, the latter denoted by a dashed arrow.

4.3 Design Principles

Based on our experience with the automotive operating system CiAO [22, 32], the following design process is guided by four reusable principles that lead to an aspect-aware system design. The corresponding AspectC++ idioms are beneficial for a clean system design and fine-grained configurability that does not induce run-time overhead.

4.3.1 Loose Coupling

The principle of *loose coupling* describes the relation between two software modules. Consider a networking device driver and the IP networking protocol implementation. Both modules should be exchangeable – the device driver should work with different protocol stacks and a protocol stack should be able to use various device drivers. Using AOP, loose coupling between these modules can be established easily:

The device driver exposes an *explicit join point*, which is an empty function without arguments named `ready()`, that is called after the device driver is initialized. Aspects can hook into this function and bind the device driver module to the IP stack on the initialization event (see last row of Table 2). The *binding aspect* is a recurring aspect role whenever several software modules, possibly developed by independent working groups, need to be integrated in a noninvasive manner.

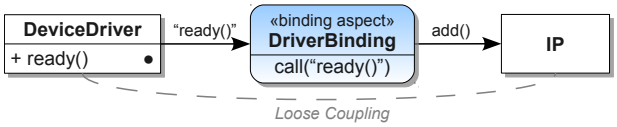


Figure 6: The principle of *loose coupling*. A *binding aspect* establishes a relation between two otherwise unrelated modules. The invocation of `ready()` is translated to an `add()` call, which registers the `DeviceDriver` at the `IP` stack.

4.3.2 Minimal Extensions

The principle of *minimal extensions* aims at incremental system design. Software modules are designed to provide only the minimal functionality that is required for a working system – no more, no less. Additional features are only introduced by *extension aspects*.

For instance, the TCP module itself does neither support a sliding window for transmitting nor for receiving. In Section 4.2, these features were already identified as crosscutting concerns and mapped to aspects. Thus, the sliding window for transmitting and for receiving are both designed as minimal extensions to the TCP module. As shown in Figure 7, the sliding window crosscuts the module TCP as well as both receive and transmit buffering.

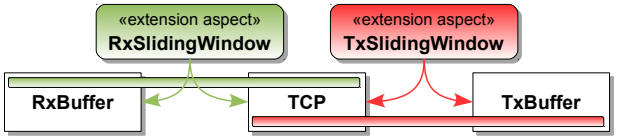


Figure 7: The principle of *minimal extensions*. *Extension aspects* introduce optional concerns by means of *slices*. Existing software modules are extended by additional member variables and functions in order to fulfill their duty.

4.3.3 Visible Transitions

The aspect-aware design enables the substitution of internal system policies. Thus, all relevant transitions need to be accessible by aspects. The system designer has to ensure that control-flow transitions are visible as unambiguous join points, that are preferably distinguishable at compile time. A fine-grained class hierarchy enclosed in expressive C++ namespaces provides a rich set of join points in the sense of *visible transitions*.

Often, a highly crosscutting concern represents a system policy. For example, the byte order conversion, which is examined in the last column of Table 2, is an exchangeable policy. On machines that internally operate at the network byte order, no conversion has to be performed. Otherwise, a conversion of every multi-byte entry of each protocol header takes place. We designed all protocol header structures as C++ classes, whose attributes are exclusively accessed throughout `get()` and `set()` functions. Therefore, aspects can alter the result and arguments of these functions and perform the byte order conversion directly in place. Figure 8 shows such an aspect for `get()` functions of 16-bit words. The source code for 32-bit words and `set()` functions is similar. Other software modules do not have to be aware of network byte ordering at all, because this concern is entirely encapsulated by a single *policy aspect*.

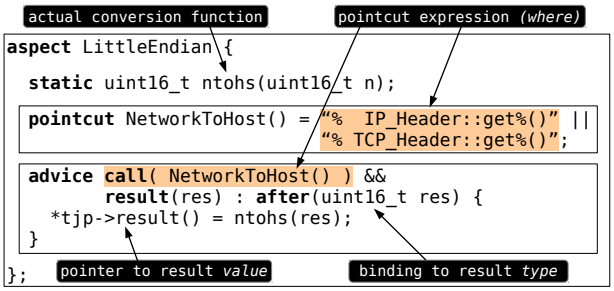


Figure 8: Byte order conversion by *policy aspect*. Conversion logic is encapsulated in the *static* function `ntohs`, which is implicitly invoked by the advice. The pointcut declaration describes *where* the advice shall take effect, that is all `get()` functions exposed by protocol header structures (*visible transitions*). `tjp` (this join point) provides access to context information, and `tjp->result()` yields a typed pointer to the return value.

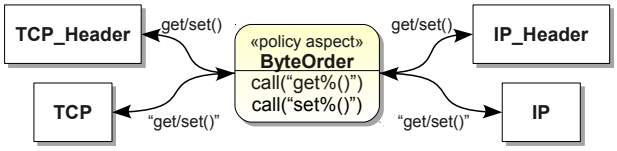


Figure 9: The principle of *visible transistions*. A fine-grained call structure makes system internal transitions explicit. Thus, system policies are designed to be exchangeable *policy aspects*, that alter the control-flow at relevant locations.

4.3.4 Upcall Dispatcher Hierarchy

A layered system defines a "uses" relation among its layers – typically top-down (see Figure 5). Each module is aware of all subordinate layers and directly depends on the ones it uses. The other way around, a module that is used by superordinate modules must not have knowledge of them. Otherwise, circular dependencies occur and configurability is lost.

Upcalls, which are invocations in the opposite direction of the "uses" relation, cause difficulties since they manifest in a tight coupling between layers due to circular dependencies. Consider an IP module that receives an IP packet and propagates it to superordinate UDP and TCP modules, which are in turn configurable and thus not always present. There may be even more subscribers for the content of an IP packet, such as a configurable ICMP module. The problematic nature of upcalls becomes even worse if multiple layers are involved. For example, an Ethernet packet, that is dispatched from the bottom layer up to the top layer, forces circular dependencies between all traversed layers.

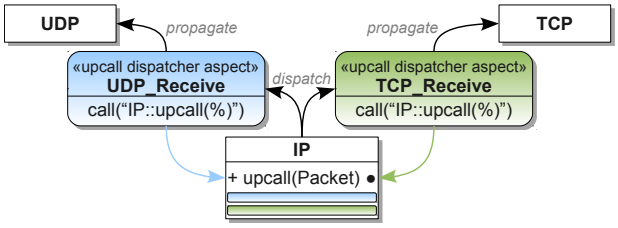


Figure 10: The principle of *upcall dispatcher hierarchy*. Bottom-up data propagation in a layered system is performed by independent *upcall dispatcher aspects*. Each aspect evaluates inherent dispatch conditions and eventually lifts data upwards.

The *upcall dispatcher hierarchy* circumvents this issue by strictly sticking to the inherent "uses" relation. Figure 10 illustrates this design principle based on the aforementioned example. The bottom-up propagation of data is implemented by explicit join points, which are empty functions (see Section 4.3.1), whose arguments contain the data to be propagated. In Figure 10, the module `IP` offers the public function `upcall(Packet)`, that is called by `IP` in order to dispatch a packet to superordinate modules. The actual dispatch is performed by independent *upcall dispatcher aspects*, which hook into the `upcall` function and evaluate whether the packet is to be accepted. Dispatch criteria are usually a check of the protocol's header identification field and a verification of the checksum. The dispatch process itself is distributed to several aspects, which provide configurability because they can be omitted easily.

A key benefit of upcall dispatcher aspects is that they are stackable. Consider a module `Ethernet`, that also provides an explicit join point `upcall(Ethernet_Frame)`. The upcall dispatching to superordinate modules, for instance `ARP` and `IP`, can be performed

in the same way. Thus, circular dependencies between layers are avoided and the already developed “uses” relation is enforced by this design principle.

4.4 Architectural Design Decisions

The efficiency of a system is vitally determined by crucial design decisions during the development process. Several concerns are not covered in the system’s specification (i.e., RFCs), although they define fundamental properties of the software architecture. These *hidden concerns* have to be identified, evaluated and carefully designed in order to narrow the design space. Thus, the unnecessary variability is eliminated that is known in advance to bear no convincing benefits. This is the major software engineering challenge at the very end of the design phase, which is involved with the overall performance of the system.

4.4.1 Memory Management

The most important concern with regard to performance and scalability is memory management [7]. Each network packet has to be buffered in data memory, and access to it must be very fast. On typical embedded systems, memory and especially data memory (RAM) is a scarce resource, so that an economical utilization is necessary. Furthermore, on such systems, a dynamic memory management (heap) is often not economical and thus not available.

In general, there are two possible designs for the memory management of an IP stack:

1. **Linear buffers**, large enough to hold a contiguous packet.
2. **Linked lists** of small buffers, each buffer containing a fragment of the packet.

Both designs are reasonable and used in popular networking systems – Linux applies linear buffers whereas BSD favors linked lists. The advantage of linear buffers is speed, because a contiguous packet often fits into a single cache line. Extra effort has to be paid to packet construction in the space of linear buffers. The size of all applied protocol headers has to be known in advance and an appropriate amount of memory has to be reserved in the front of each buffer. Linked lists simplify packet construction, because arbitrary protocol headers can be linked in front of the list. This flexibility comes at the price of poor cache locality due to the scattering of list elements across the data memory.

In order to gain maximal efficiency, we chose to use linear buffers with two configurable sizes. This design decision takes into account that the distribution of packet sizes tends to be bimodal [25]. Large packets deliver the payload while small packets acknowledge successful reception. Because the size of the payload heavily depends on the application, we leave the configuration of the buffer sizes to the application programmer. We designed the memory management to be part of a parametrized API, that is implemented as generic C++ templates.

Thus, the application programmer can tune the memory utilization for each particular connection by instantiating a template class of the API. For example, the memory management parameters for high-throughput connections should satisfy the *bandwidth-delay product*. In general, the optimal amount of memory is approximately $throughput_{desired} \times delay_{end-to-end}$ for a sliding window.

4.4.2 Concurrency

An implicit requirement for IP networking software is the concurrent handling of multiple connections. Full concurrency is achieved by threads that are scheduled by an operating system. Thus, IP networking software could be run inside the context of threads or inside the operating system itself.

Our design decision is to reuse the runtime context of threads in order to implement concurrency. This approach has several advantages: A *prioritization* of threads is enforced, because the networking routines of high-priority threads are preferred over low-priority ones. This is essential for real-time systems, where hard timing deadlines have to be met. As a consequence, the feature of prioritized connection comes for free if thread priorities are supported by the underlying operating system.

The second important advantage of this design is that *isolation* between different connections can easily be established. Each connection can constitute separate pools of buffers in the context of the actual thread, so that no interference between concurrent connections can occur. The isolation of memory regions is especially interesting for safety-critical applications. On the other hand, shared buffers use the memory resources more efficiently. Therefore, CiAO/IP implements buffer sharing as a configurable feature.

4.4.3 Synchronization

A structured coordination between multiple threads running IP networking software and the underlying operating system is crucial for the performance of the system. The actual synchronization primitives depend highly on the services provided by the operating system, and are even nonexistent if no operating system is used. Therefore, synchronization is an exchangeable system policy.

According to the design principles (see Section 4.3), we make synchronization explicit by providing unambiguous join points for visible transitions. Thus, policy aspects implement synchronization by using operating system services. For example, we integrated our IP stack into the automotive operating system CiAO. CiAO itself provides *events* and *alarms* for synchronization, both specified by the AUTOSAR [2] standard. Policy aspects invoke these services implicitly and allow a noninvasive integration, so that neither the operating system nor the IP stack have to be modified. Therefore, the integration of this IP Stack into other operating systems is straightforward.

5. EVALUATION

To evaluate the CiAO/IP stack, we have implemented most features specified by the relevant RFC documents on a wide range of platforms:

8-bit: A BTnode[8] sensor network platform, which uses an Atmel ATmega128L *AVR* microcontroller with 128KiB ROM and 4KiB RAM combined with a sub 1 GHz CC1000 radio.

16-bit: A TI EZ430-Chronos[9] development system, which comes in a sports watch. It combines a *MSP430* microcontroller with 32KiB ROM, 4KiB RAM and a sub 1 GHz CC1101 radio device.

32-bit: An *IA-32* PC with Intel Core 2 Quad 6600 CPU, 4GiB RAM and an Intel 82566DM Gbit Ethernet adapter.

64-bit: Same as above, but running in *AMD64* mode.

Table 1 in Section 2 has already shown that all features of CiAO/IP that we identified as “optional” during the domain analysis became configurable in the implementation. This section will prove that this high degree of variability does not come a cost. CiAO/IP clearly outperforms uIP and lwIP in terms of memory consumption (Section 5.1), performance (Section 5.2), and energy efficiency (Section 5.3).

[8] `http://www.btnode.ethz.ch/`
[9] `http://processors.wiki.ti.com/index.php/EZ430-Chronos`

5.1 Memory Consumption

Memory is a scarce resource on embedded devices. This is true for both program memory (ROM) as well as data memory (RAM). For instance, the aforementioned 16-bit MSP430 platform contains only 32KiB ROM and 4KiB RAM, which software must not exceed. Especially system software that offers services to the actual application and is itself not self-contained, has to be exceptionally small and specifically designed for the embedded systems domain. A counterexample is the Linux[10] kernel, whose basic feature *TCP/IP Networking* increases the uncompressed kernel size by about 234KiB on IA-32. Hence, we focus our evaluation on uIP, lwIP and CiAO/IP.

The requirements on data memory are dominated by buffering. Network packets are inherently dynamic content that has to be stored in the RAM. Besides buffers, networking software stores connection state in the RAM, which requires an infinitesimal amount of memory compared to a buffer for a single IP packet. The buffer sizes of uIP, lwIP and CiAO/IP are statically configurable and, thus, their RAM consumption is comparable.

However, CiAO/IP is more flexible as it offers to configure whether buffers for a particular connection should be allocated on the data memory section, on the dynamic heap (if present), or even on the runtime stack of the current application. This feature is especially useful for temporary connections that are active only for a short period of time (for instance, during a software update), and, hence, consume data memory only within this period. Buffers, that are globally allocated on the data memory or the heap at system startup, constantly take RAM.

Besides data memory, an efficient usage of program memory is crucial. The requirements to ROM directly depend on the functionality of the software: More features lead to higher ROM consumption. In the sequel, we discuss the issue of program memory in detail. All following values are obtained by using the *gcc*[11] with optimization for size (-0s).

5.1.1 The Cost of TCP

TCP is certainly the most complex part of the Internet protocol suite. As a consequence, an implementation of TCP requires more program memory than other protocols. Nevertheless, TCP is essential for interoperability since many applications rely on it, for example web services. A realistic scenario for TCP also includes IP, and on IA-32 and AMD64 additionally Ethernet and ARP, since almost every personal computer comes with an Ethernet adapter.

Figure 11(a) shows the memory consumption of a TCP client in the described scenario. We use the minimal possible feature selections of uIP, lwIP and CiAO/IP to evaluate the minimal costs for a TCP client. The stacked bar diagram outlines the costs for the IP stacks themselves (lower section) and the cost for the application code that has to be added for a working TCP client (upper hatched section). This is relevant, as the required application code differs substantially between the IP stacks: uIP, for instance, does not perform retransmissions of lost TCP segments itself – it is up to the application logic to detect packet loss and to retransmit missing packets. For lwIP, we use its memory-efficient event-driven, nonsequential API that, however, also requires more effort on the application side than the UNIX-like sockets of CiAO/IP. For all IP stacks, the application code also handles synchronization with network devices and timer events. The necessary synchronization primitives are provided by the underlying CiAO operating system.

Figure 11(b) shows the memory consumption of a TCP server in the same scenario. For both client and server, the cost of TCP varies among the different architectures. It is notable that uIP has been heavily optimized for 8-bit architectures by using 8-bit data types and arithmetic whenever possible [9]. This specialization saves about 50% of the TCP code size on AVR and MSP430 compared to CiAO/IP, whereas it leads to an increased code size on IA-32 and AMD64.

5.1.2 The Cost of UDP

UDP is the second essential transport protocol of the Internet protocol suite. Compared to TCP, UDP is almost a null protocol, which is reflected by its low memory consumption. Consider a node of a sensor network that periodically transmits sensor measurements. For such a scenario, UDP is sufficient.

Figure 11(c) summarizes the cost of transmitting UDP packets without optional checksumming[12]. Since TCP is a mandatory feature of uIP, there is a high overhead if used for UDP only. lwIP consumes even more program memory due to architectural properties, such as flexible packet buffers and generic driver interfaces. CiAO/IP is by far the most efficient implementation with regard to UDP. On the average over all architectures, CiAO/IP requires only twelve percent of the memory that uIP respectively lwIP take.

For receiving UDP datagrams the results are similar. Figure 11(d) outlines our results. The application code (hatched section of top the bars) for receiving is larger than for transmitting, because synchronization with receive interrupts is involved.

5.1.3 Feature Scalability

We have measured the code size induced by each configurable feature: Starting with the minimal possible feature selection of the particular IP stack we have incremented the number of active features one by one. The results are depicted in Figure 12.

The minimal code size of uIP (see Figure 12(a)) is rather high due to the mandatory TCP feature, ranging from 4253 byte on MSP430 to 6481 byte on AMD64. When all features of uIP are active, the code size almost doubles.

lwIP scales from a minimum of 6836 byte on IA-32 up to a maximum of 38.7kB on AVR. This spread indicates scalability to some extend, but the absolute values are quite high. For example, the sum of all considered lwIP features on MSP430 occupies 30KiB of its 32KiB internal ROM (94%).

In contrast, CiAO/IP scales from a minimum of 789 byte on AMD64 up to a maximim of 20.8kB on AVR. In between, each configurable feature increases the memory consumption by a very small amount. The sharp spike in this diagram is caused by the TCP feature, which adds an exceptional large portion of code due to its complexity.

5.2 Performance

The aforementioned 8-bit and 16-bit hardware platforms are ill-suited for performance benchmarking, since their gross networking throughput is limited to 76.8 kbit/s respectively 500 kbit/s by the radio hardware. Moreover, the net throughput of payload (*goodput*) is primarily determined by the link quality[13] and timing of TCP retransmissions. To analyze the influence of the different IP stacks on high-speed networking performance, we therefore focus on the Gbit Ethernet adapter of the IA-32 platform:

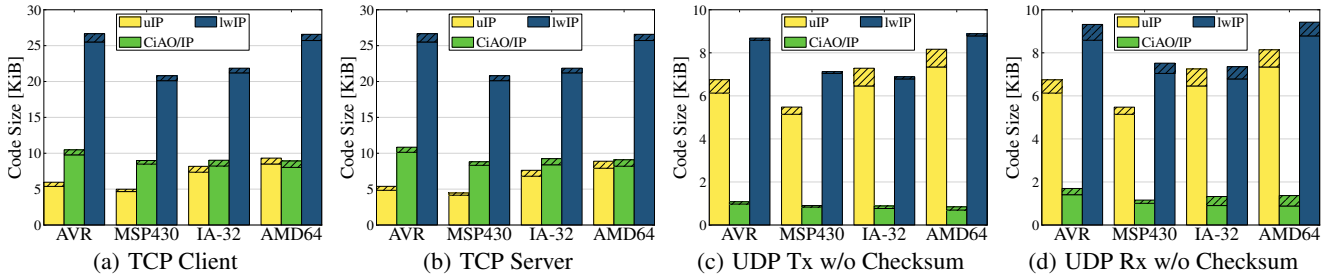
We have connected two PCs (Intel Core 2 Quad 6600 CPU, 4GiB RAM, Intel 82566DM Gbit Ethernet) via a crossover cable to create

[10] version 2.6.33

[11] version 4.4 (MSP430/IA-32/AMD64), version 4.3 (AVR)

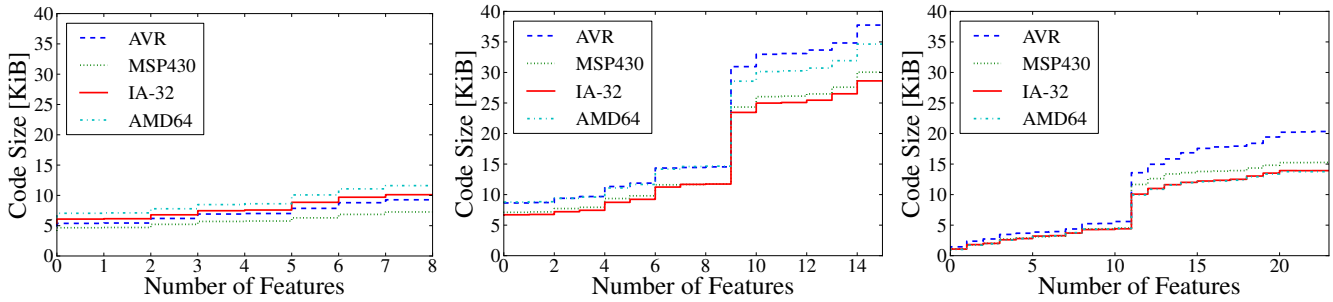
[12] The values for IA-32 and AMD64 also encompass the cost for Ethernet and ARP.

[13] We measured a goodput of 1161 bit/s for CiAO/IP and 680 bit/s for uIP on the average in the wireless setup described in Section 5.3. Both systems were configured with exactly equivalent features.



(a) TCP Client (b) TCP Server (c) UDP Tx w/o Checksum (d) UDP Rx w/o Checksum

Figure 11: Code size (ROM) for common use cases. Each use case includes IPv4. On IA-32 and AMD64, Ethernet and ARP is also included to get a working system. Figures (a) - (d) show the minimal memory consumption of each particular IP stack for the desired functionality. The lower sections of the stacked bars constitute the code size of the IP stacks themselves. The upper hatched sections indicate the cost for the application code that has to be added for full operation. Hardware platforms range from 8-bit (AVR) to 64-bit (AMD64).



(a) **Feature increment of uIP:** TCP Server (IPv4) and ICMP, TCP Client, TCP Urgent Data, IPv4 Fragment Reassembly, UDP, UDP Checksumming, ARP, IPv4 Forwarding, NDP

(b) **Feature increment of lwIP:** UDP (IPv4), UDP Tx Checksumming, UDP Rx Checksumming, UDP Lite, IPv4 Forwarding, IPv4 Fragment Reassembly, IPv4 Fragmentation, ARP, ARP Queuing, VLAN, TCP, TCP Queuing, TCP Backlog, TCP Timestamps, ICMP, IGMP

(c) **Feature increment of CiAO/IP:** UDP Tx (IPv4), UDP Tx Cksum., UDP Rx (IPv4), UDP Rx Cksum., IPv4 Frag. Reass., Ethernet, Tx Cksum Offload., Rx Cksum Offload., ARP Reply, ARP Request, Static ARP Cache, ARP Timeout, TCP Client, TCP Server, Reset Invalid TCP Packets, Tx Slid. Window, Rx Slid. Window, SWS Tx, SWS Rx, MSS Opt., RTT Est., Cong. Control, Generic Buffer, Generic Rx Queue

Figure 12: Code size (ROM) vs. features. For each TCP/IP stack, the first measuring point shows the code size of the initial minimal feature selection. Each subsequent measuring point constitutes the addition of one more configurable feature. Thus, figures (a) - (c) outline an incremental feature selection and visually evince scalability.

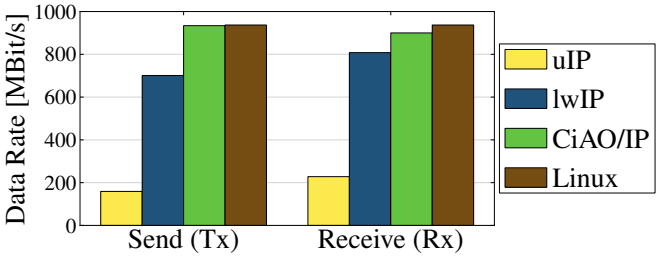


Figure 13: Goodput of a TCP connection. Depicted is the goodput between two local IA-32 machines, that are connected via gigabit Ethernet (crossover). One peer is always a Linux 2.6 box, whereas the machine under test runs the different IP stacks.

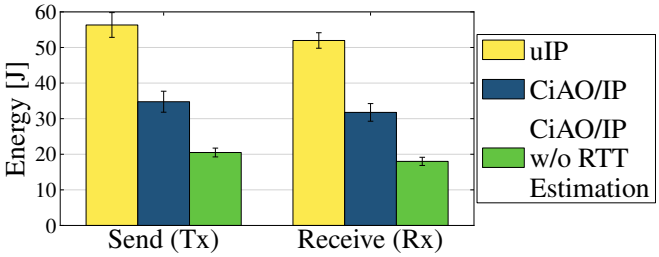


Figure 14: Energy consumption. Shown is the mean energy consumption and the sample standard deviation of unidirectional TCP data transfers (32 kB) between two wireless sensor networking nodes. *CiAO/IP w/o RTT Estimation* uses a fixed retransmission timeout (200 ms) for lost packets.

an actual loss-free environment. One box constantly runs Linux 2.6, whereas the other box runs the IP stack under test. We have established a single TCP connection between the two machines and measured the achieved goodput. Each IP stack is equipped with 100 KiB RAM for buffering and exactly equivalent feature configuration, except for uIP which solely supports a single buffer (1514 byte) due to the missing sliding window. For fairness, uIP is extended by *uip_split* [9] in order to reach maximal throughput. This uip_split extension avoids the default 250 ms delay between packet transmits. As uIP implements a stop-and-wait procedure, it otherwise would suffer from delayed acknowledgments of the Linux peer. Figure 13 shows the results of our measurements. We classify the results into sending and receiving, which describe the operational mode of the IP stack under test.

With Linux on both ends a maximal goodput of 937 Mbit/s is achieved, which here can be understood as the upper bound.

CiAO/IP, however, achieves almost the same transfer rates (934 Mbit/s for Tx, 900 Mbit/s for Rx).

lwIP clearly lacks behind. This is mostly caused by lwIP's linked packet buffer management (*pbufs*), which have to be copied into a contiguous memory region before delivery to the device driver. We intentionally do not use vectored I/O (scatter/gather) hardware accelerators that might overcome this design decision, since embedded devices generally do not provide such hardware support.

uIP is not competitive due to the missing sliding window, which results in a significant performance degradation.

5.3 Energy Consumption

Energy is probably the most scarce resource for battery-powered devices. We have chosen the aforementioned BTnode[14] sensor-net platform for determining the impact of IP stacks on energy consumption. A typical duty for sensor nodes is a firmware update: a unidirectional, reliable data transfer of a single file. Thus, our setup for energy measurements[15] consists of a TCP client, that transmits 32 kB of data to a listening TCP server.

On the link layer, we use the *B-MAC* [27] protocol for collision avoidance and preamble sampling with an interval of 25 ms (*Low Power Listening*). The packet size is limited to 255 byte, and each IP stack is equipped with 300 byte for buffering. Due to a quite noisy radio channel, which leads to a high percentage of corrupt packets (approximately 50%), we disabled the exponential backoff algorithm of every TCP implementation, which otherwise would slow down the retransmission of lost packets unnecessarily. Unfortunately, we did not get lwIP working under these harsh conditions, so that we had to omit lwIP in the following measurements.

Figure 14 presents the mean energy consumption and the sample standard deviation of uIP and CiAO/IP. For fairness, we did not activate the sliding window feature of CiAO/IP, because uIP does not support it and would otherwise be disadvantaged. To sum up, both IP stacks were configured with exactly equivalent features. Consequently, no congestion control was performed, because the slow-start algorithm can only be applied to a sliding window.

However, even without this beneficial feature CiAO/IP consumes significantly less energy: A uIP sender consumes a mean of 56.3 Joule, whereas CiAO/IP takes a mean of 34.8 Joule for the same task. This is caused by the coarse granularity of uIP's round-trip time estimation, which can only be multiple of 500 ms. In contrast, CiAO/IP performs this estimation much more accurate in units of 1 ms. Thus, lost packets are detected earlier (up to 499 ms per lost packet). To prove this claim, we deactivated the round-trip time estimation of CiAO/IP (an optional feature in CiAO/IP) and fixed the retransmission timeout to 200 ms. This way, the energy consumption of the sender could be further reduced to 20.5 Joule. These observations are also valid for the receiver, whose energy consumption is determined by the time the transfer takes to complete.

[14] AVR microcontroller (8-bit) running at 7.37 MHz, CC1000 transceiver operating at 868 MHz and 19.2 kbit/s

[15] *Hitex PowerScale* with *ACM probes* used for energy measurements

5.4 Summary of Results

CiAO/IP outperforms uIP and lwIP in almost all scenarios. The only exception is the TCP client/server, where uIP requires on the AVR/MSP430 platform only half the code size of CiAO/IP – which is exactly the scenario uIP has been optimized for [8]. However, even in this configuration uIP consumes significantly more energy (1.6x) and delivers a much smaller throughput (0.59x). In all other configurations, our aspect-oriented CiAO/IP stack outperforms the C implementations of uIP and lwIP notably with respect to code size, throughput, energy, and feature-wise scalability.

6. DISCUSSION

The main goal of this work is to achieve feature-wise scalability of the layered TCP/IP protocols by a software engineering approach for static configurability. As shown in the previous section, CiAO/IP offers fine-grained configurability and thereby achieves excellent results with respect to code size, throughput and energy. In the following, we discuss the pros and cons of our approach with respect to more "soft" properties, including maintainability, configurability, portability, and optimization.

6.1 Maintainability

The complexity and maintainability of some piece of software is generally hard to quantify. However, source-code metrics, such as lines of code (LOC) can serve as an indicator: The minimalistic uIP consists of 2796 LOC[16] whereas CiAO/IP consists of 4943 LOC and a comparable subset of lwIP[17] encompasses 11987 LOC. Clearly, more features correspond to more LOC, but in general aspect-oriented software consists of less LOC than its C counterparts, because crosscutting concerns are modularized into aspects and, thus, avoid scattered fragments of similar code.

A good example for this is the macro-based implementation of the byte order concern illustrated in Figure 1. As pointed out in Section 2.3, every header access has to be surrounded by one of the byte ordering macros (`htons, ntohs, ...`). This is a tedious and error-prone design rule. Our aspect-oriented CiAO/IP stack circumvents this issue by a much better separation of concerns: Byte ordering is a policy, hence we implement the byte order conversion as a modular policy aspect (see Section 4.3.3) that implicitly affects the `get()` and `set()` functions of all classes that describe network protocol headers. The respective aspect consists of a single source code file with less than 50 lines of code. This reduces complexity, as the byte order in CiAO/IP is correct by construction – it is not possible to create an invalid packet by a forgotten call of a conversion macro. Therefore, AOP contributes to maintainability of source code by modularization and separation of concerns.

On the other hand, AOP may impair the independent development of modules of the system [19]. Since aspects do not expose an explicit interface, the interaction of the system's modules can become complicated, because conceptually, each aspect can potentially affect each module [31]. To obviate this drawback, our aspect-oriented

[16] effective lines of code (excluding empty lines and comments), obtained with *cloc*: `http://cloc.sourceforge.net/`

[17] version 1.32, without IPv6, DNS and DHCP

design principles (see Section 4.3) are also applied as a means to make such interactions explicit.

6.2 Configurability

The excellent configurability we achieved in CiAO/IP is primarily the result of a software product-line development process that considers configurability as a fundamental design goal from the very beginning. However, eventually the individual features have to be implemented in a way that makes it possible to "leave them off" if not needed. The advantage of AOP here is *loose coupling* by the advice concept: Essentially, advice inverts the direction in which control-flow relationships are specified. This facilitates the self-integration of optional features into the control flows of the base system. Furthermore, advice-based binding is inherently loose – if the addressed join point is not present, the binding is silently dropped. This property is useful for the implementation of *interacting* optional features, which are difficult to tackle with other decomposition approaches [17]. In CiAO/IP we thereby could keep a textbook-like strict layering in the design and implementation of the IP stack without scarifying efficiency.

6.3 Portability

Loose coupling is also beneficial with respect to portability to other platforms. The aspect-based integration of our IP stack into the operating system makes it easy to use CiAO/IP with other systems or to use it without any operating system. We are currently working on implementations for eCos [24] and FreeRTOS [4].

From the perspective of application developers, switching to CiAO/IP is relatively easy as CiAO/IP offers the well-understood BSD socket concept and does not, like uIP and lwIP, require the application engineer to implement and maintain stack-specific state machines on the application level.

6.4 Optimization

The main benefit of the high configurability offered by CiAO/IP is optimization for *nonfunctional properties*, such as memory footprint, throughput, and energy consumption. This is done by first activating only those network protocols that meet the *functional* requirements depending on the actual field of application, for example TCP and IPv4. Other protocols should be disabled. This way, the design space of valid feature selections is significantly narrowed. As outlined in Table 1, several optional features do not provide basic networking protocols by themselves, and, thus still remain *open* for this particular feature selection. The assignment of these open features, for instance the TCP sliding window and round-trip time estimation to either *on* or *off*, leads to a *design space exploration*.

We developed the *feedback approach* [28] for statistical reasoning on nonfunctional properties and assignment of open features. Information about previously configured feature selections have to be captured by run-time tests and the results feed into a database. Thus, nonfunctional properties for further feature selections that are not yet evaluated can be estimated, and a testing set for validation is generated automatically. This is a semi-automated process that relies on testing of specific feature selections until the database is saturated.

7. RELATED WORK

We originally developed our aspect-aware design method as well CiAO/IP for our CiAO operating system family for deeply embedded devices [22, 21]. CiAO implements the automotive AUTOSAR OS standard [2], a rapidly growing domain with respect to “mobile Internet” applications. Other operating system projects that aim to bridge deeply embedded systems and the Internet world are mostly from the domain of sensor networks: Recent versions of *TinyOS* [20] provide the BLIP IPv6 stack, which also provides configurability of UDP and TCP and is implemented with the component model offered by the NesC language [12]. BLIP has the distinction of operating solely at IPv6. All other systems we are aware of are based on either uIP or lwIP, both of which clearly dominate the domain: *Contiki* [10] uses uIP; *eCos* [24] supports lwIP and, alternatively, a BSD-derived stack; *FreeRTOS* [4] allows developers to choose between uIP (called FreeTCPIP) and, again, lwIP.

AOP as a modularization approach for configurable system software has also been applied in the domain of middleware [35, 14] and embedded databases [16]. Other modularization approaches in the domain are too numerous to be discussed here. However, several researchers favor *feature-oriented programming* [5] – and even the C preprocessor is experiencing a renaissance by the provisioning of better tool support towards a “virtual separation of concerns” [1].

8. CONCLUSIONS

Network protocol stacks for the domain of resource-constrained embedded systems have to fullfill a broad variety of functional requirements, while at the same being thrifty with respect to hardware resources, especially memory and energy. This calls for static configuration approaches to tailor the provided functionality to the actual application’s needs.

We presented the CiAO/IP stack and its underlying design approach for embedded system software, which pushes the limits of static configurability to a new level. In our design approach, the domain (mainly given by RFCs) is analyzed with *software product line* methodology; the resulting features are then structured by a *concern impact analysis* and implemented with *aspect-oriented programming* as fine-grained and loosly coupled implementation artefacts.

By following this design approach in the development of CiAO/IP, we did not only achieve a clear and complete separation of concerns in the code (thus, maintainability and portability), but also excellent configurability and scalability of the resulting protocol stack: CiAO/IP outperforms *uIP* and *lwIP* in terms of code size (up to 84% / 88% less than uIP / lwIP for a UDP sender on an AVR), throughput (up to 587% / 33% higher than uIP / lwIP for a TCP sender on IA-32 with Gbit Ethernet) and energy consumption (up to 63% lower than uIP on AVR for a TCP sender).

We hope that our results encourage other developers of system software to follow the guidelines in this paper.

9. ACKNOWLEDGEMENTS

We wish to thank the anonymous reviewers for their helpful and encouraging comments. Special thanks go to Prabal Dutta, whose shepherding helped us to clarify the content of this paper.

This work was partly supported by the German Research Council (DFG) under grant no. SP 968/4-1, SP 968/5-1, and SFB 876 projects A1 and A4.

10. REFERENCES

[1] Sven Apel and Christian Kästner. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[2] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[3] D. Barisic, M. Krogmann, G. Stromberg, and P. Schramm. Making embedded software development more efficient with SOA. In *21st International Conference on Advanced*

Information Networking and Applications Workshops, 2007 (AINAW '07), volume 1, pages 941–946, May 2007.

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

[5] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *26th Int. Conf. on Software Engineering (ICSE '04)*, pages 702–703. IEEE, 2004.

[6] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989.

[7] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27:23–29, 1989.

[8] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98. ACM, 2003.

[9] Adam Dunkels. *The uIP Embedded TCP/IP Stack. The uIP 1.0 Reference Manual*. Swedish Institute of Computer Science, 2006.

[10] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki — a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[11] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 263–270, New York, NY, USA, 1999. ACM.

[12] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '03)*, pages 1–11, San Diego, CA, USA, 2003. ACM.

[13] Jonathan W. Hui and David E. Culler. IP is dead, long live IP for wireless sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 15–28, New York, NY, USA, 2008. ACM.

[14] Frank Hunleth and Ron Cytron. Footprint and feature management using aspect-oriented programming techniques. In *2002 Joint LCTES & SCOPES Conferences (LCTES/SCOPES '02)*, pages 38–45, Berlin, Germany, June 2002. ACM.

[15] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[16] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *11th Software Product Line Conf. (SPLC '07)*, pages 223–232. IEEE, 2007.

[17] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: Analysis and case studies. In Dirk Muthig and John D. McGregor, editors, *13th Software Product Line Conf. (SPLC '09)*, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.

[19] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th Int. Conf. on Software Engineering (ICSE '05)*, pages 49–58, New York, NY, USA, 2005. ACM.

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

[21] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Aspect-aware operating-system development. In Shigeru Chiba, editor, *10th Int. Conf. on Aspect-Oriented Software Development (AOSD '11)*, pages 69–80, New York, NY, USA, 2011. ACM.

[22] 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 ATC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[23] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In Yolande Berbers and Willy Zwaenepoel, editors, *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM.

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

[25] Bosko Milekic. Network buffer allocation in the FreeBSD operating system. In *Proceedings of BSDCan*, Ottawa, ON, Canada, May 2004.

[26] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. AW, 2001.

[27] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 95–107, New York, NY, USA, 2004. ACM.

[28] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Approaching Non-Functional Properties of Software Product Lines: Learning from Products. In IEEE Computer Society Press, editor, *Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC 2010)*, pages 147–155, Los Alamitos, CA, USA, 2010.

[29] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *1992 USENIX ATC*, Berkeley, CA, USA, June 1992. USENIX.

[30] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[31] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *21st ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '06)*, pages 481–497, New York, NY, USA, 2006. ACM.

[32] Jochen Streicher, Christoph Borchert, and Olaf Spinczyk. Upcall dispatcher aspects: Combining modularity with efficiency in the CiAO IP stack. In *1st AOSD W'shop on*

Modularity in Systems Software (AOSD-MISS ’11), pages 23–27. ACM, March 2011.

[33] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In Christoph M. Kirsch and Gernot Heiser, editors, *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2011 (EuroSys ’11)*, pages 47–60, New York, NY, USA, April 2011. ACM.

[34] Jim Turley. The two percent solution. *embedded.com*, December 2002. `http://www.embedded.com/story/OEG20021217S0039`, visited 2011-04-08.

[35] Charles Zhang and Hans-Arno Jacobsen. Quantifying aspects in middleware platforms. In *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD ’03)*, pages 130–139, New York, NY, USA, 2003. ACM.

SLOTH: Threads as Interrupts*

Wanja Hofer, Daniel Lohmann, Fabian Scheler, Wolfgang Schröder-Preikschat
Friedrich–Alexander University Erlangen–Nuremberg, Germany
E-mail: {hofer,lohmann,scheler,wosch}@cs.fau.de

***Abstract*—Traditional operating systems differentiate between threads, which are managed by the kernel scheduler, and interrupt handlers, which are scheduled by the hardware. This approach is not only asymmetrical in its nature, but also introduces problems relevant to real-time systems because low-priority interrupt handlers can interrupt high-priority threads.**

We propose to internally design all threads as interrupts, thereby simplifying the managed control-flow abstractions and letting the hardware interrupt subsystem do most of the scheduling work. The resulting design of our very light-weight SLOTH system is suitable for the implementation of a wide class of embedded real-time systems, which we describe with the example of the OSEK-OS specification. We show that the design conciseness has a positive impact on the system performance, its memory footprint, and its overall maintainability.

I. INTRODUCTION

One of the core responsibilities of an operating-system kernel is the management of control flows in the system. Traditionally, these encompass synchronously executed *threads*, and asynchronously triggered *interrupt handlers*. The latter ones are usually signaled by hardware devices and have an implicitly higher priority than synchronous control flows by being able to interrupt the CPU at any time. This bifid priority space—divided up into interrupt priorities and thread priorities—induces a problem termed *rate-monotonic priority inversion*: Interrupt-handler control flows that have a semantically lower priority than a real-time thread can interrupt and delay the execution of that real-time thread [4].

In previous work, we have tackled that problem by using a coprocessor that pre-handles all interrupts [18]. In this paper, we show how to overcome rate-monotonic priority inversion by making use of more sophisticated interrupt systems as available on many newer hardware platforms, without the need for a coprocessor. In our SLOTH[1] system, we propose to internally design every control flow in the system as an interrupt—even regular threads—by implementing thread-related system calls using the interrupt system. The SLOTH approach has the following advantages:

- It implements a unified priority space, allowing for arbitrary distribution of priorities to both thread and interrupt control flows.
- The kernel implementation can be kept extremely concise and is therefore well maintainable and subject to easy and comprehensive testing.
- By letting the hardware schedule the control flows, the performance of the system calls and context switches is very high compared to regular, purely software-based thread implementations, providing for both very low and deterministic overhead.

At the same time, the application programmer can still use the notion of a thread as a unit of decomposition; the API that SLOTH offers remains the same as in a traditional implementation, eliminating the need for porting.

We have implemented the conformance class BCC1 of the OSEK–operating-system specification [17], which targets event-driven embedded real-time systems, for the Infineon-TriCore microcontroller [5], which features an interrupt subsystem that fulfills the requirements for a SLOTH system. This way, we can show that our SLOTH design can be implemented using state-of-the-art commodity hardware, and we can evaluate the advantages of such a design.

II. DESIGN

In a seminal paper entitled *Interrupts as Threads* [7], Kleiman and Eykholt describe the implementation of control flows in the Solaris-2 kernel, in which interrupt handlers can become full-fledged threads if they need to block. We propose quite the opposite approach, which treats all threads as interrupt handlers and thereby lets the hardware handle most of the scheduling work implicitly.

A. Overview of OSEK OS

Our kernel design targets an embedded, event-driven real-time system. In order to simplify the description, we use the terminology and system-service grouping as specified by the OSEK-OS standard [17], an operating-system specification widely used in the automotive domain. The feature diagram in Figure 1 gives an overview of the features of an OSEK system.

Among the offered control-flow abstractions, *tasks* (traditionally called threads) are managed by the OS scheduler, whereas interrupt service routines (ISRs) are triggered by the hardware. The OS is oblivious of *category-1 ISRs*, which are not allowed to use its system services, whereas *category-2 ISRs* have to be synchronized with the kernel since they *are* allowed to use system functions. Whether a task is preemptable by higher-priority tasks or not is configured globally (full preemption or no preemption) or locally on a task-by-task basis (mixed preemption). Furthermore, whether multiple activations of a task can be stored by the OS and whether it supports multiple tasks with the same priority are optional

* This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7-1. Wanja Hofer was supported by the German Academic Exchange Service (DAAD) under grant no. D/09/40595.

[1]The name honors both the lazy animal breed and the deadly sin.

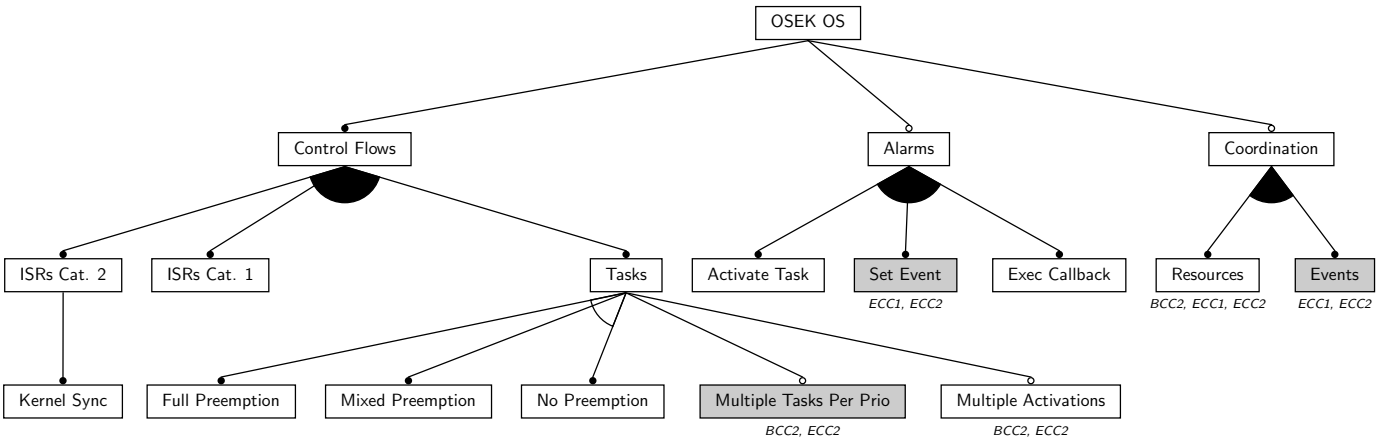


Fig. 1. Feature diagram of the OSEK–operating-system specification. Feature types include mandatory features (filled circle), optional features (hollow circle), minimum-one feature sets (filled arc), and exactly-one feature sets (hollow arc). Features not yet integrated in the SLOTH design are depicted in gray color. If a particular feature is mandatory only in conformance classes other than the basic BCC1, this information is given below that feature.

system features. *Alarms* are timer abstractions that can activate a task, execute a callback function, or set an event upon expiry after a specified period of time. To wait for an *event* is the only possibility for a task to become blocked; it is unblocked when that event is set by another control flow. The other coordination abstraction—*resources*—is used to synchronize critical sections within the application by mutual exclusion.

OSEK also defines four conformance classes (BCC1, BCC2, ECC1, ECC2), which define minimum requirements on which of the optional features have to be provided (see also Figure 1). In our SLOTH design, we target the OSEK conformance class BCC1. Thus, we have a statically configured system with static task priorities (no task creation and altering of the task priorities at run time is possible) and run-to-completion tasks only (i.e., tasks cannot block by waiting for an event), supporting only one task per priority level. Apart from that, the application can be as complex as any other OSEK application, and it is configured and programmed using the same OSEK system API that any software implementation offers, so no porting is required.

B. SLOTH *Design Overview*

An overview of our design is given in Figure 2. Tasks and ISRs are represented by an abstract interrupt source that has an appropriately configured priority. The corresponding request is triggered either synchronously by the CPU when the `ActivateTask()` system service is invoked, or asynchronously by connected hardware devices. Additionally, tasks that are configured to be activated by OSEK alarms after a specified time period are represented by interrupt sources that are connected to the timer system.

The scheduling of the system is done completely in hardware. First, an IRQ arbitration unit decides which of the attached interrupt sources (and, therefore, which of the attached control flows) has the highest priority. After that, the CPU is interrupted by an interrupt request, but only if its current priority is lower than the one of the requested control flow.

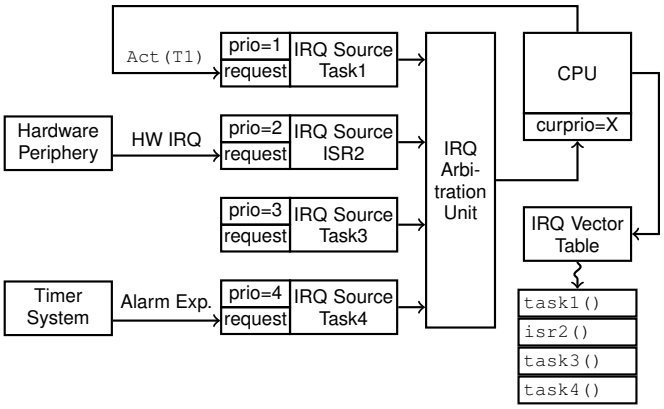


Fig. 2. Design of a SLOTH system, using interrupt handlers for the implementation of threads. The interrupt sources have a statically configured priority and are either triggered synchronously by the CPU through a system-service call (e.g., Task1), through hardware-periphery IRQs (e.g., ISR2), or through the timer system after setting a task alarm (e.g., Task4).

In that case, the corresponding task or ISR is dispatched by looking it up in the vector table. Note that the current priority level of the CPU does not necessarily have to be the one of the executing task. The CPU priority level is also altered for synchronization purposes—for instance, in order to implement resources for mutual exclusion (see Section II-E).

The rest of this section details the design of typical embedded–operating-system services on the example of the major system-service groups offered by the OSEK operating system. In parallel, refer to Figure 3 for an example control flow in a SLOTH system. It uses the application configuration as depicted in Figure 2; that is, Task1, ISR2, Task3, and Task4 have the priorities 1, 2, 3, and 4, respectively.

C. *Task Management*

In SLOTH, tasks (OSEK's name for threads) are identified by their priority; that is, a task's ID is the same as its priority. Activating a task corresponds to merely triggering the corresponding interrupt source. The resulting interrupt

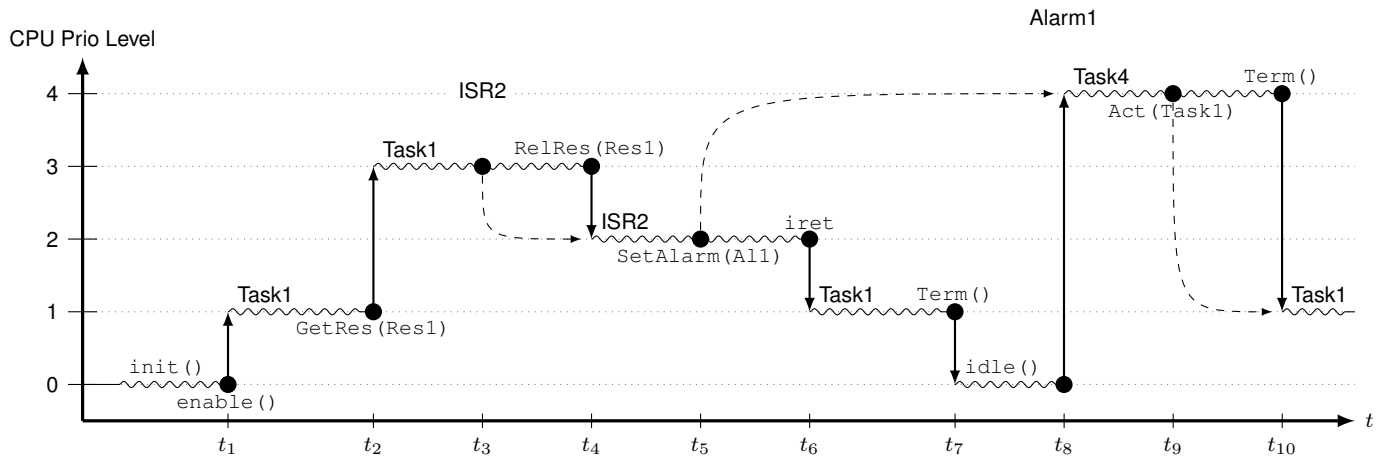


Fig. 3. Example control flow in a SLOTH system. The execution of most system calls leads to an implicit or explicit altering of the current CPU priority level, which then leads to an automatic and correct scheduling and dispatching of the control flows by the hardware.

request is then immediately handled if the priority of the new task is higher than the one performing the activation, given that the currently running task is configured to be preemptable. Termination of a task is a simple return from the interrupt handler, which then leads to an automatic dispatch of the pending task with the next-highest priority. For task chaining, the specification demands that the task performing the chain operation is completed before the chained task starts to execute. In SLOTH, this behavior is ensured by disabling interrupts for a short and bounded time, then activating the task to be chained, and then returning from the interrupt handler, which implicitly re-activates interrupts.

In the example depicted in Figure 3, when Task1 terminates at t_7, it restores the previous priority 0 by executing a return-from-interrupt instruction. Likewise, when Task4 terminates at t_{10}, it also tries to restore the previous priority 0, which leads to an automatic scheduling of Task1 first, because it is still pending with priority 1. Its activation by Task4 at t_9 was automatically delayed, because of the lower priority of Task1.

D. Interrupt Handling

In our SLOTH system, tasks and those kinds of ISRs that are allowed to perform system calls (named category-2 ISRs in OSEK) are completely identical, thereby unifying the priority space and allowing for mixed priorities between them. Only those ISRs that are guaranteed *not* to perform any system calls (category-1 ISRs) have priorities higher than all tasks and category-2 ISRs. Hence, category-2 interrupts can be suspended by setting the current CPU priority level to the highest priority of all category-2 ISRs. All interrupts (including the ones of category 1) can be suspended or disabled by the application by setting the CPU priority level to the highest priority of *all* ISRs, or by disabling interrupts completely.

Both kinds of ISRs are dispatched by the hardware whenever the CPU priority level is below the one of the interrupt request. In the example control flow in Figure 3, for instance, ISR2 is not dispatched until Task1 lowers its priority to 1 by releasing Resource1 at t_4, although ISR2 was already requested at t_3.When ISR2 terminates at t_6, it executes a regular return-from-interrupt instruction and thereby implicitly re-actives the pending control flow with the next-highest priority, Task1.

E. Resource Management

Resources (OSEK's terminology for mutex synchronization objects) are used to protect critical sections. In order to avoid deadlocks and priority inversion, OSEK prescribes a stack-based priority ceiling protocol similar to the stack resource policy by Baker [1]. This protocol mandates immediately raising a task's priority to the resource ceiling priority upon acquiring the resource, and lowering it to the original priority upon releasing it. This way, tasks can never become blocked upon resource acquisition and the acquisition will always succeed; otherwise, another task with a higher priority (gained by acquiring that same resource) would be running instead.

In our SLOTH kernel, a resource ID is equal to its ceiling priority—that is, the highest priority of all tasks and category-2 ISRs that can acquire it. Thus, acquiring a resource means simply raising the current CPU priority level to the ceiling priority (i.e., the resource ID), and releasing it means re-setting the level to the original value. Since multiple resources can be acquired, the previous priority has to be saved on a stack. Because of the static system configuration, the stack usage induced by resource acquisition can be bounded at compile time.

In the example sketched in Figure 3, since Resource1 can be acquired by both Task1 and Task3 (not active in the example control flow), its ceiling priority is 3. Thus, when Task1 acquires it at t_2, it raises the CPU priority level to 3, and it tries to re-set it to the previous priority 1 upon releasing it at t_4, leading to the dispatching of the pending ISR2 as described in Section II-D.

F. Alarms

Alarms are offered by the OSEK operating system to enable the application to take action after a specified time budget has elapsed. If an alarm is configured to activate a task, an interrupt source that is connected to the hardware timer system is chosen for that task and configured with its priority. The service call setting an alarm can then be simply implemented by programming the connected hardware timer appropriately; the timing parameters have to correspond to the ones provided to the system call. When the timer expires, the configured task is then activated automatically by triggering the interrupt source, leading to preemption if the currently running task has a lower priority. Since most of these actions are done by hardware, the alarm-service implementation itself can be kept very light-weight.

In the example in Figure 3, ISR2 sets an alarm at t_5, which is configured to activate Task4 upon expiry. When the hardware timer fires at t_8, it automatically activates Task4, because the corresponding interrupt source has the priority 4 of Task4.

If an alarm is configured to execute a callback function, that function can be treated the same way—as a special, high-priority task. Callback functions were originally introduced in OSEK in order to offer a very light-weight reaction possibility, but with SLOTH's light-weight thread design, this is not an issue to be concerned about (see also Section V).

G. Nonpreemptive Systems

The SLOTH design as described in this paper targets a preemptive system, in which each activation of a higher-priority task leads to a rescheduling and dispatching. In order to implement a nonpreemptive system, only a few details have to be adjusted in the design.

First, every nonpreemptable task starts at the priority level of the highest-priority task in the system instead of at its own priority. This way, when a task activates a higher-priority task, that task is not dispatched immediately. Second, an explicit point of rescheduling (e.g., the OSEK system service `Schedule()`) is implemented by lowering the priority to the original task priority before raising it again. This way, any pending tasks of higher priority are allowed to run and to complete at this point before the original task is executed again. Note that, in a preemptive system, `Schedule()` is effectively empty since rescheduling is always performed immediately anyway.

Using the same idea, the special scheduler resource `RES_SCHEDULER` is implemented by setting its ceiling priority to the one of the highest-priority task in the system. By acquiring this virtual resource for a limited period of time, preemptive tasks can delay preemption in critical sections until after releasing the resource—as demanded by the specification. Groups of tasks that *do not preempt* each other *within groups* but *do preempt* each other *between groups* can be designed the same way; for this purpose, OSEK offers *internal resources*. By letting each task run with the priority of the highest-priority task in its group (i.e., by acquiring this internal resource), preemption within the group is delayed until the task reschedules explicitly. This rescheduling system call temporarily lowers the current priority to the task's original priority—like in a completely nonpreemptive system as described above.

H. Multiple Task Activations

The optional OSEK feature to support multiple activations of the same task can be easily integrated by an additional activation counter per task. When activating a task, in addition to requesting an interrupt, the corresponding counter is increased. Upon termination of the task, the counter is simply decreased, and—if the number of activations is greater than zero—the interrupt is requested again before really terminating the task.

This mechanism only works for tasks activated through the corresponding system call; it does not work for real ISRs that are triggered by hardware periphery, since to the best of our knowledge there is no interrupt controller that can store more than one activation. Because SLOTH implements tasks activated by alarms by letting the timer system simply set the interrupt-request bit (see Section II-F), those tasks have only limited multiple-activation support in SLOTH.

I. Summary of the SLOTH Thread Abstraction

Compared to traditional OS thread implementations, SLOTH threads are different in several points.

First, SLOTH threads run to completion and are only preempted by higher-priority threads. Conventional threads can wait for an event and block, letting lower-priority threads run. This is a limitation that we want to tackle in future work (see also Section VII), but which still allows for a broad range of applications (see also Section V).

SLOTH's run-to-completion property leads to a strictly stack-like control-flow dispatching, which is also illustrated in Figure 3. This way, SLOTH can use only a single shared stack—the interrupt stack—for all its threads, and the preempted-thread context is stored on that stack. Traditional threads have a stack of their own and have their context saved by the kernel in an additional structure.

Traditional threaded OS kernels maintain a *software* ready queue and running pointer, and they need additional information in software, such as the priorities of the threads, to make scheduling decisions whenever the state of one the threads changes, possibly leading to a new thread being dispatched. SLOTH has all this information implicit in the interrupt hardware subsystem, with the ready queue being represented by the interrupt-pending bits of the hardware, relying on the hardware to do the scheduling and the dispatching.

To the application programmer, all of these differences are hidden beneath the same thread API; SLOTH currently offers the same OSEK task abstraction and system services like any other, software-based implementation.

J. Requirements on the Hardware Interrupt System

For our approach to be feasible, we have two requirements on the interrupt subsystem of the hardware platform that our SLOTH kernel is implemented on:

1) Interrupt priorities: The interrupt system shall offer as many different interrupt priorities as there are threads and interrupt handlers in the system.
2) Interrupt triggering: The interrupt system shall support manual, software-based triggering of interrupts. This can be offered through a special instruction or through the modification of corresponding hardware registers.

Note that these are the only requirements for a SLOTH implementation. Some platforms fulfill these requirements natively (such as the Infineon TriCore detailed in Section III-A, or the ARM Cortex-M3), whereas others have an external interrupt controller that provides the corresponding functionality (such as the APIC present on all modern Intel-x86 systems).

III. IMPLEMENTATION

We have implemented our SLOTH approach for the Infineon TriCore [5], an embedded microcontroller platform commonly used in the automotive domain. We shortly describe the relevant features of the platform before sketching our implementation.

A. The Infineon-TriCore Platform

The TriCore platform has a sophisticated interrupt subsystem that fulfills our requirements as stated in Section II-J.

Interrupt sources are represented by *service request nodes* (SRNs), which encapsulate all the relevant properties such as priority, enable status, and request status. All SRNs are connected to an *interrupt arbitration unit* (IAU) through a special bus for exchanging priority information in order to find a precedence among the pending interrupts. This process, called *arbitration*, takes a defined number of system-bus cycles, which itself depends on the system clock frequency and the priority range of the SRNs actually competing in the arbitration. Hence, the fewer tasks and ISRs are configured in a system, the fewer arbitration cycles are needed to prioritize the concurrent requests.

Most of the SRNs are connected to an actual hardware source (e.g., the general-purpose timer array of the TC1796 derivative features 92 SRNs), but there are special SRNs available for software-only access (named `CPU_SRCx`). Additionally, hardware-connected SRNs that are not used in a given application can also be used to implement threads as interrupts, because every SRN has its registers memory-mapped, allowing for software-based interrupt triggering as required by SLOTH (see Section II-J).

B. Task-Activation Implementation

The implementation of the SLOTH design as sketched in Section II is very straight-forward on the TriCore platform. However, special attention has to be paid to the synchronous task-activation mechanism.

Since a task is implemented as an interrupt handler, a prolog is included in the interrupt vector that saves the context of the interrupted task (which is a single instruction on the TriCore), re-enables interrupts, and then jumps to the actual task function. If a task wants to terminate, this corresponds to a simple return-from-interrupt instruction, which restores the previous CPU priority level and implicitly schedules and dispatches the pending control flows in the system. Before the actual return, the context of the interrupted task is restored first.

Synchronous task activation is performed by requesting the corresponding interrupt using the appropriate SRN. Basically, this is compiled to a single store instruction to a memory-mapped register. However, it takes a while until the interrupt request is propagated to the CPU, depending on the current state of the arbitration system. Since an activation of a higher-priority task is supposed to happen *synchronously* in a preemptive system, this activation has to be synchronized. This is done by first disabling interrupts, and by then reading back the request bit in order to synchronize the hardware and software [6]. After that, `nop` instructions are inserted to accommodate for the worst-case latency, which arises if an arbitration round has just begun. The number of `nop` instructions to be inserted is calculated and bounded statically, depending on the number of arbitration rounds and the number of cycles per arbitration round as demanded by the application configuration (i.e., number of tasks and system frequency)[2]. The subsequent enable-interrupts instruction is then the defined, synchronous point of preemption:

```
void ActivateTask(TaskType id)
{
    _disable();
    setr(id); /* set service request flag */
    srr(id); /* read back to sync HW/SW */
    /* worst case: wait for 2 arbitrations */
    nopsForOneArb();
    nopsForOneArb();
    _enable(); /* defined preemption point */
}
```

The same applies to the chaining of another task: The executing task relies on the chained task being executed immediately after it terminates if that new task has the highest priority in the system at that point. As described in Section II-C, interrupts are also disabled before the activation in order to prevent the new task from running until the old one has terminated.

Even when a *lower-priority* task is activated, this situation may require synchronization. Consider, for instance, that directly after the (nonsynchronized) activation of a lower-priority task, the priority level is lowered by terminating the running task. This has to be the defined point for the context switch, and not when the interrupt actually occurs at the CPU. If the activation is not synchronized, a lowest-priority task may execute for a few cycles *after* the termination of the high-priority task and *before* the interrupt dispatches the activated task—which is a clear violation of the specification. Hence, *every* task activation is synchronized with `nop` timing as described above, independently of its priority.

All of the cases described in this section where interrupts need to be suspended temporarily for synchronization purposes

[2]The timing properties of the TriCore platform that are needed for this calculation are exactly defined by Infineon in an application note [6].

only disable them for a *short* and *bounded* amount of time. That way, that time can be accounted for during the schedulability and latency analysis of the whole real-time system. The number of introduced `nop` instructions varies between 8 and 22, depending on the configuration, and is effectively time when the CPU cannot do useful work (although the interrupt system is performing the priority arbitration during that time). However, this is a small price to pay compared to the overhead of a traditional, software-based scheduler implementation (see also Section IV).

Note that after an interrupt request has been triggered, its source—a periphery device or the CPU itself—and the requested type of control flow—task, ISR, or callback—is completely oblivious to the CPU; it simply and automatically dispatches the corresponding control flow if the current CPU priority is below the requested priority.

C. Application Configuration and System Generation

Since the system is statically configured and tailored to the needs of the application, this information can be used to generate static dispatching code that is highly optimized (see Figure 4). As the configuration describes the mapping from task IDs to interrupt sources, the essence of the `ActivateTask()` implementation (i.e., its subfunction `setr()`), for instance, is an `if–else` cascade that sets the request bit in the appropriate SRN depending on the task-ID parameter, which is also its priority. This implementation and the corresponding application calls can be statically analyzed and optimized by the compiler, resulting in an inlined piece of code consisting of a single instruction—namely the one that sets the correct bit. Similar code is generated for querying that bit to see if a task is in ready state, for setting an alarm that activates a specific task upon expiration, and for initializing the SRNs with the request bit already set, depending on the auto-start properties of the corresponding tasks. These implementations are also extremely light-weight since they are subject to the compiler's static analysis.

Furthermore, the interrupt vector table needs to be generated to jump to the correct task functions from the interrupt handlers of the different priorities as configured for the current application.

Additionally, a couple of system-relevant constants are extracted from the application configuration (see also Figure 4):

- The task IDs are set to their configured priorities, and the resource IDs are set to their ceiling priorities depending on the tasks that are configured to potentially acquire them.
- The ceiling priority of the virtual resource `RES_SCHEDULER` is set to the highest priority of all configured tasks.
- The number of needed arbitration cycles is derived from the configured system frequency and the priorities of the configured tasks, and the corresponding `nop` timing for synchronous task activation is calculated.

D. System Startup

Upon startup of the SLOTH system (here, in `main()`, after the start-up code has initialized the stack, the interrupt vectors, and some platform-specific registers), the interrupt system needs to be initialized accordingly. This boot process basically encompasses the initialization of the SRNs according to the priorities in the application configuration; the corresponding code can easily be generated as described in Section III-C. If the configuration has any tasks declared to be auto-started upon startup, the request bit in the corresponding SRNs is set in addition to the priority. Note that the system is started with a CPU priority level of 0 but with interrupts still disabled; hence, these auto-start task activations will not take effect until interrupts are enabled after the system initialization is complete (see also t_1 in Figure 3).

The initial CPU priority level of 0 in `main()` leads to a fallback to that routine whenever there is no ISR or task ready to be scheduled—otherwise, the pending priority is greater than 0. Thus, appropriate idling action can be taken in an infinite loop in `main()`, putting the microcontroller unit to sleep or in a low-power mode until an interrupt (representing a control flow ready to be dispatched) requires servicing (see also t_7 and t_8 in Figure 3).

Additional initialization of the general-purpose timer array and the I/O-line–sharing unit of the Tricore is needed if the application uses alarms to activate tasks or execute callbacks.

IV. EVALUATION

Since the design of our SLOTH system aims at making more use of existing hardware features than other operating systems, the software implementation is accordingly very concise.

A. Lines of Code

The whole system implementing the OSEK conformance class BCC1 for the TriCore-TC1796 board as described above takes less than 200 source lines of code to be implemented[3]. This number includes code that is generated from the application configuration (see also Figure 4) with one instance per task, resource, and alarm configured; additional code for more instances is similar and adds to the number of lines of code, but not to its complexity. The start-up code for the platform is not included in those 200 lines of code; it was basically taken as supplied by the compiler (`tricore-gcc` by HighTec; programmed in assembly).

B. Memory Footprint

Due to the concise system code base, the resulting footprint of the compiled system image is also small; the kernel implementing the conformance class BCC1 takes about 700 bytes[4]. This number again reflects the whole kernel with one task, resource, and alarm instance; additional instances can add to the memory footprint because additional interrupt handlers

[3]Logical, semicolon-terminated lines; measured with CCCC [9], version 3.pre84.

[4]Compiled with `tricore-gcc` by HighTec, version 3.4.5, with `-O3` optimizations.

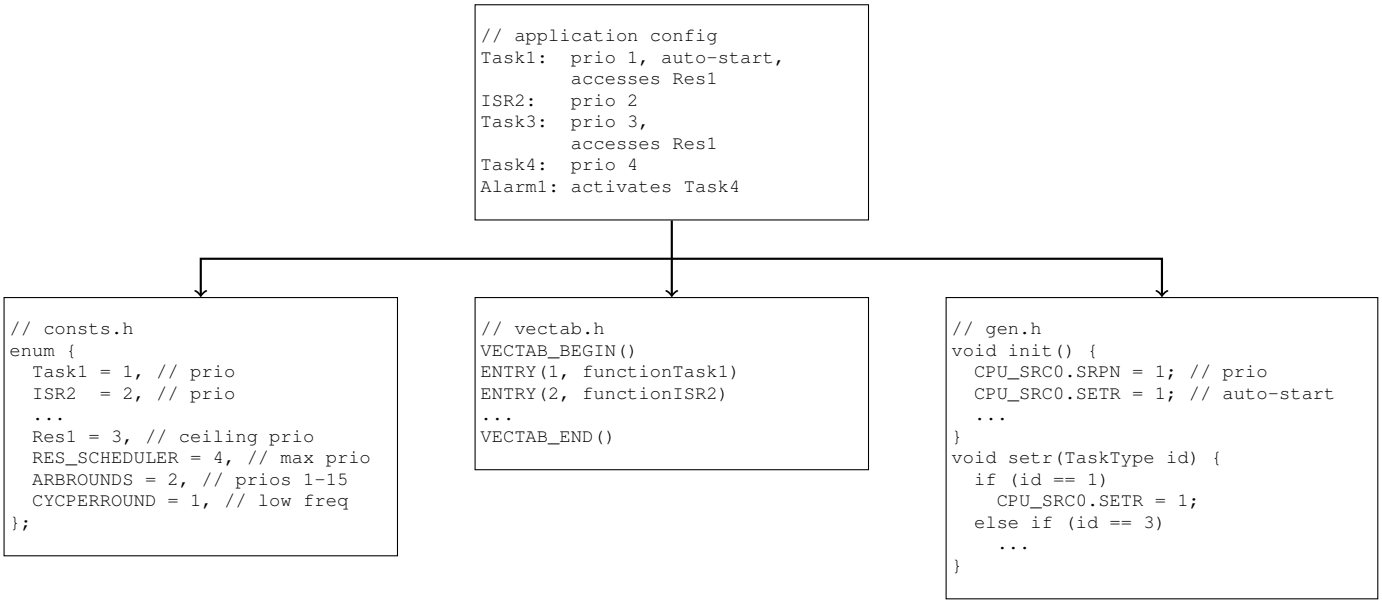


Fig. 4. SLOTH application configuration and system generation.

in the vector table are needed for additional tasks, for instance. The compiled start-up code as provided by the compiler takes up an additional 1,000 bytes, which can be reduced to about 500 bytes by tailoring its initialization functionality to the one actually needed by SLOTH.

Note that due to the system's hardware proximity, most system calls are very short and therefore subject to function inlining. Consider, for instance, the `setr()` function (see generated code in Figure 4), which is the essence of the system call `ActivateTask()` (see implementation sketch in Section II-C). Since in many static applications, the system-call parameter is constant at compile time, the dispatching through the `if–else` cascade can be statically optimized by the compiler. The result is a single store instruction to the corresponding memory-mapped register (without the following `nop` synchronization). Additionally, the functionality of the operating system is tailored to the application's needs by excluding system functions that are not referenced by the application; this is done through function-level–linking support by the compiler and linker.

C. Execution Performance

In order to assess the quantitative effects of our SLOTH approach on the operating-system kernel, we have performed an analysis of run times of selected scenarios in a preemptive system with the features of the OSEK conformance class BCC1 (i.e., without events, without multiple tasks per priority, and without multiple activations). The selected scenarios encompass those system calls that are implemented differently in SLOTH because of its hardware-based nature. The other system calls will have similar performance as in a traditional, software-based kernel, as well as the application itself. The evaluated scenarios include:

1) Synchronously activating a task of *lower* priority, does *not* lead to dispatching: execution time of the `ActivateTask()` system service.
2) Synchronously activating a task of *higher* priority, *does* lead to dispatching: execution time from the point before `ActivateTask()` to the first user instruction of the activated task.
3) Terminating a task and returning to the previously running task: execution time from the point before `TerminateTask()` to the point after the task was dispatched.
4) Chaining a task: execution time from the point before `ChainTask()` to the first user instruction of the chained task.
5) Acquiring a resource: execution time of the `GetResource()` system service.
6) Releasing a resource *without* inducing another task to be dispatched: execution time of the `ReleaseResource()` system service.
7) Releasing a resource *with* inducing another task to be dispatched: execution time from the point before `ReleaseResource()` to the first user instruction of the dispatched task.

We have evaluated all of those scenarios with two different interrupt-system configurations that reflect the best case and the worst case regarding the interrupt-arbitration latency on the TriCore platform (see also Section III-A):

A) Best case (minimum number of arbitration cycles): 1 arbitration round (suitable for up to 3 interrupt priorities), 1 bus cycle per arbitration round (only good for lower system frequencies).
B) Worst case (maximum number of arbitration cycles); 4 arbitration round (suitable for up to 255 interrupt priorities), 2 bus cycles per arbitration round (also good for high system frequencies).

	1) Act() w/o dispatch	2) Act() w/ dispatch	3) Term() w/ dispatch	4) Chain() w/ dispatch	5) GetRes() w/o dispatch	6) RelRes() w/o dispatch	7) RelRes() w/ dispatch
SLOTH A) (best case)	34	60	14	67	19	14	36
SLOTH B) (worst case)	48	74	14	81	19	14	36
CiAO	75	206	107	139	19	66	204

TABLE I
SLOTH BEST-CASE AND WORST-CASE PERFORMANCE IN SELECTED SCENARIOS, COMPARED TO PERFORMANCE USING THE CIAO OS. DEPICTED IS THE NUMBER OF 20-NS CLOCK CYCLES NEEDED TO EXECUTE THE PARTICULAR TEST CASE.

The measurement results for SLOTH are depicted in Table I[5]. For comparison purposes, we have deployed and measured the same application scenarios on CiAO, a configurable, OSEK-like embedded operating system for which an implementation for the TriCore platform is also available. CiAO has a traditional software scheduler, and its competitive performance compared to other commercial implementations has previously been published [12]. For the CiAO tests, we have configured the operating system to provide the minimal amount of features necessary for the scenarios so that it provides the same capabilities that SLOTH does. Since both CiAO and SLOTH have the same OSEK API, the test applications run are the same.

Because `nop` timing is required in SLOTH for synchronous task dispatching (see also Section III-B), the scenarios 1), 2), and 4) depend on the hardware-arbitration configuration, with the extremes being the best-case configuration A) and the worst-case configuration B). The other scenarios only alter the priority level of the CPU, which is independent of the number of arbitration cycles; hence, the run times for SLOTH are the same for both configurations. Note that scenario 1) differs in the configurations A) and B) although it does not lead to dispatching. This is because, as argued in Section III-B, situations may arise where synchronization is necessary nevertheless.

Compared to CiAO, SLOTH performs equally well or better in all scenarios. Especially the scenarios 2), 3), 4), and 7), all of which include a scheduling and dispatch operation, are significantly faster on SLOTH, which relies on the interrupt system to perform these tasks. Depending on the hardware configuration and whether a new task is activated or a running one terminates, SLOTH only needs between 280 ns and 960 ns for a task switch (including the actual context switch) on a 50-MHz system.

V. DISCUSSION

The SLOTH system design, relying on extensive use of the hardware interrupt system, leads to a small kernel bearing several advantages over traditional kernel designs, with a good range of application fields nevertheless.

[5]Measurements were performed on a TriCore TC1796B with 50 MHz system frequency and CPU frequency (cycle time of 20 ns). The run times were obtained with a TRACE32 hardware debugging and tracing unit by Lauterbach and averaged over 5,000 iterations each.

In its current shape, SLOTH does not support blocking functionality for threads. It can therefore exploit the resulting strictly stack-based nature to implement its dispatcher using interrupt levels. As can be seen from the results of the evaluation in Section IV, this does not only lead to a concise system *design*, but also to a concise *implementation*. The small kernel code base is very well manageable and therefore maintainable with regard to possible requirement adaptations. Additionally, it is an ideal candidate to verification, a property of utmost importance to many real-time systems of the class targeted by SLOTH.

Furthermore, the evaluation revealed that the memory footprint of a SLOTH implementation is extremely small, which is another important property for the domain of deeply embedded systems, where single superfluous bytes in memory demand can lead to significant overall cost increase. Because of the strictly stacked nature of SLOTH, stack-sharing techniques can be used to reduce the stack part of the application’s RAM demand to a minimum; the dispatcher only uses a single interrupt stack from the very beginning of the system startup. Moreover, the increased use of hardware functionality leads to a superior system performance compared to traditional, software-based implementations, which was shown in the evaluation in Section IV.

The fact that the SLOTH design maps control flows that are of different kinds in other systems (e.g., OSEK tasks, category-1 ISRs, category-2 ISRs, and callbacks) to a single abstraction has a major influence on the system conciseness, leading to the advantages described above. Additionally, the system *synchronization*—a major concern in all concurrent systems—is tremendously simplified, because the adjustment of the current CPU priority level is the single measure needed for all kinds of synchronization demands. This includes both demands by the application itself and demands internal to the system to keep its data structures from being corrupted by asynchronous control flows. The application demands are satisfied by raising the CPU priority level to the resource ceiling priority to acquire a resource, by raising it to the highest level of all configured tasks to disable preemption in a critical section (this is prescribed in OSEK by the special resource `RES_SCHEDULER`), by raising it to the highest level of all category-2 ISRs to implement `SuspendOSInterrupts()`, and by raising it to the highest level of *all* ISRs to implement `SuspendAllInterrupts()`. The system-internal

demands to keep the kernel synchronized are implemented by raising the level to the highest priority of all tasks and category-2 ISRs (both of which can access system data structures) configured in a given system.

SLOTH's unified control-flow design also introduces an additional degree of freedom for the system designer, who can decide upon the system's priority space independent of the synchronous/asynchronous nature of the distinct control flows. In other systems, asynchrononous interrupt handlers always have precedence over synchronous, scheduler-managed threads, which leads to a bifid priority space bearing the problem of rate-monotonic priority inversion [4], amongst others[6]. Furthermore, functionality that is described as optional in the specification because of its complexity can be offered along the way. For instance, the OSEK-OS specification says that the participation of category-2 ISRs in the priority ceiling protocol for resources (see Section II-E) is optional. If interrupts and threads are designed the same way like in SLOTH, they can automatically take part in that protocol, allowing for more complex application synchronization possibilities.

Additional types of control flows that were introduced to offer more light-weight alternatives to the traditional threads and ISRs (like OSEK callbacks and category-1 ISRs) are superfluous in SLOTH systems because the offered control-flow type already has a very low overhead to begin with. In fact, SLOTH can offer OSEK tasks and category-2 ISRs at the price of an OSEK callback or category-1 ISR.

Despite its simple design, SLOTH is suitable for the implementation of a wide range of real-time systems. This includes event-triggered systems with fixed priorities, as targeted by the widely-spread OSEK-OS specification, for instance. The missing blocking functionality can be tolerated by many real-world applications, which avoid making use of that feature because of reasons of memory demand (e.g., stack sharing is hampered) and analyzability of the system behavior. SLOTH is suitable to implement the most well-known fixed-priority scheduling algorithms—like the rate-monotonic algorithm [10] and the deadline-monotonic algorithm [11], for instance.

Legacy applications that are programmed using the API described in the OSEK standard can be used with SLOTH without modifications, since SLOTH implements the OSEK specification. The existing application configuration, including task priorities and other properties (also defined by OSEK, in its OSEK implementation language [16]), can also be used unmodified by the SLOTH generator to produce the configuration-dependent code (see also Section III-C). Hence, no porting is needed for an OSEK application to benefit from SLOTH's advantages, and the application programmer can rely on the programming model and abstractions he is used to.

VI. Related Work

We are not aware of any work that is really similar to our approach in handling threads as interrupts.

Vice versa, Kleiman and Eykholt [7] proposed to handle *interrupts* as full *threads* so that interrupt handlers can use system services if they need to. They can even wait for an operating-system event and block, leading to the dispatching of another thread that is ready. This model is implemented in the Solaris kernel for desktop and server systems and was adapted by Lohmann et al. for an embedded-system kernel [13]. However, the overhead introduced by their approach leads to interrupt handlers having a performance overhead similar to that of threads, whereas our approach gives threads the (lower) overhead of interrupt handlers.

There are several approaches to aid the operating-system scheduler by using hardware abstractions; however, all of them rely on *customized* hardware. All of those approaches—including cs2 [14], FASTCHART [8], Silicon TRON [15], HW-RTOS [3], and Atalanta [19]—move operating-system functionality to the hardware level by synthesizing special circuits on FPGA boards and offering that functionality on a co-processor–like basis. Our approach, however, is applicable to commodity off-the-shelf hardware.

As previously mentioned, some of the implications of stack-like control-flow scheduling as used in SLOTH (and as prescribed by OSEK BCC1) were investigated by Baker. This includes the possibility for efficient process stack allocation by means of stack sharing and the stack resource policy to avoid priority inversion [1], [2]. Our implementation uses both techniques and can benefit from them.

VII. Future Work

Our current system design supports the features of the OSEK conformance class BCC1; its main shortcoming is the missing support for blocking by events, which do not fit in with the current stack-oriented design. In order to be compatible to the classes ECC1, ECC2, and BCC2, we plan to carefully sketch a design for event support and support for multiple tasks per priority[7] (see also Figure 1). Both of those features are rather simple to implement in software—which we could do to integrate the functionality in our SLOTH system—, but we aim for a more sophisticated design that preserves the benefits of an operating system implementing threads as interrupts. For instance, the peripheral control processor of the Infineon-TriCore platform can be configured to be the primary service provider for all interrupts. This co-processor could then be used to implement part of that additional functionality by filtering the events and interrupting the main CPU only when needed.

Furthermore, we plan to investigate the applicability to and suitability of other hardware platforms for our SLOTH approach. For instance, all modern Intel-x86 systems have an advanced programmable interrupt controller (APIC) available, which can compensate for the interrupt-system shortcomings of the x86 CPU architecture. By programming the I/O APIC accordingly and by using inter-processor interrupts sent

[6]In fact, this problem is the reason why programmers are taught to keep ISRs short. In SLOTH, the ISRs can be long, since they reside in the same priority space as the system tasks.

[7]This is especially problematic together with multiple activations, since the activation order within a priority class is prescribed to be preserved.

through the processor's local APIC, we are positive that we can implement our design on that well-known platform. We also want to investigate what kinds of features hardware platforms have to offer to support our SLOTH concept *in an ideal way*. This includes the analysis of available hardware features and their shortcomings with respect to our approach.

Finally, we want to explore the feasibility to extend our SLOTH design to multiprocessor systems, and we want to investigate whether similar approaches can be used to implement time-triggered systems.

VIII. SUMMARY

We have presented our SLOTH operating-system design, which uses interrupt handlers as its universal control-flow abstraction—also to implement synchronously activated threads. This model allows for a simple implementation of all major services expected from a statically configured, event-driven operating system, providing the same programming model and interface, which we have shown using the example of the OSEK-OS specification. As a side effect, the resulting unified priority space completely avoids the real-time problem of rate-monotonic priority inversion.

In order to evaluate the properties of the SLOTH design, we have implemented it on the Infineon-TriCore platform. We have shown that the unification of the control flows in the system has a significant impact on the operating system's conciseness in the design, in its implementation code, and in its compiled memory footprint. Furthermore, since SLOTH uses the hardware interrupt system instead of software-implemented routines to schedule the system's control flows, the resulting performance is more than competitive.

In our opinion, the results of our SLOTH work should encourage OS engineers to make better use of the hardware abstractions that a given platform offers. Especially in the domain of *embedded* OSes, where a small footprint and efficient execution are crucial, a small limitation in portability can often be traded for an improvement of those properties.

REFERENCES

[1] Theodore P. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings of the 11th International Conference on Real-Time Systems (RTSS '90)*, pages 191–200. IEEE Computer Society Press, Dec 1990.

[2] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[3] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*, pages 324–329, New York, NY, USA, 2006. ACM Press.

[4] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[5] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *TriCore 1 User's Manual (V1.3.5), Volume 1: Core Architecture*, February 2005.

[6] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *AP32009, TC17x6/TC17x7 – Safe Cancellation of Service Requests*, July 2008.

[7] Steve Kleiman and Joe Eykholt. Interrupts as threads. *ACM SIGOPS Operating Systems Review*, 29(2):21–26, April 1995.

[8] Lennart Lindh and Frank Stanischewski. FASTCHART – a fast time deterministic CPU and hardware based real-time-kernel. In *Proceedings of the 1991 Euromicro Workshop on Real-Time Systems*, pages 36–40, Jun 1991.

[9] Tim Littlefair. CCCC - C and C++ Code Counter homepage. http://cccc.sourceforge.net/.

[10] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[12] 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 *Proceedings of the 2009 USENIX Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.

[13] Daniel Lohmann, Jochen Streicher, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Interrupt synchronization in the CiAO operating system. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, New York, NY, USA, 2007. ACM Press.

[14] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 869–875, New York, NY, USA, 2004. ACM Press.

[15] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *Proceedings of the 12th TRON Project International Symposium (TRON '95)*, pages 34–42, Nov 1995.

[16] 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 2009-09-09.

[17] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2009-09-09.

[18] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, New York, NY, USA, 2009. ACM Press.

[19] Di-Shi Sun, Douglas M. Blough, and Vincent John Mooney III. Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications. Technical report, Georgia Institute of Technology, 2002.

SLEEPY SLOTH: Threads as Interrupts as Threads*

Wanja Hofer, Daniel Lohmann, Wolfgang Schröder-Preikschat
Friedrich–Alexander University Erlangen–Nuremberg, Germany
E-Mail: {hofer,lohmann,wosch}@cs.fau.de

***Abstract*—Event latency is considered to be one of the most important properties when selecting an event-driven real-time operating system. This is why in previous work on the SLOTH kernel, we suggested treating threads as ISRs and thereby reducing event latencies by scheduling and dispatching solely in hardware. However, to achieve these benefits, SLOTH does not support blocking threads or ISRs, but requires all control flows to have run-to-completion semantics.**

In this paper, we present SLEEPY SLOTH, an extension of SLOTH that provides a new universal thread abstraction that overcomes this limitation, while still letting the hardware do all scheduling and dispatching. SLEEPY SLOTH abolishes the (artificial) distinction between threads and ISRs: Threads can be interrupt handlers and interrupt handlers can be threads.

Our SLEEPY SLOTH implementation of the automotive OSEK OS standard provides much more flexibility to application developers while maintaining efficient execution of application control flows. SLEEPY SLOTH runs on commodity off-the-shelf hardware and outperforms a leading commercial OSEK implementation by a factor of 1.3 to 19.

I. INTRODUCTION AND MOTIVATION

The core task that an operating system has to fulfill in an event-driven real-time system is to manage the control flows present in the computing system, which encompass *threads* and *interrupt service routines* (ISRs).

Threads are managed by *software*: They are activated on behalf of *software events* only (such as posting a semaphore), and they are scheduled and dispatched by *software mechanisms*, usually provided by the operating system (OS). ISRs, on the other hand, are managed by *hardware*: They are activated by *hardware events* only (such as a periphery device requiring service), and they are scheduled and dispatched by *hardware mechanisms*, usually provided by the interrupt controller. Table I lists those two types of control flows in Lines 1 and 2, together with a comparison of their semantics: Threads can block and resume execution at a later point in time, while ISRs have to run to completion. Traditional threads and ISRs form a dual priority space, with ISRs having a higher priority than all threads in the system.

A. The Issue

From the conceptual point of view, the forced distinction between threads and ISRs is problematic: The control flow semantics of an event handler (blocking or run-to-completion, as well as its priority relative to other event handlers) should not be defined by the source of the event (hardware or software), but by the requirements of the real-time application. Furthermore, the dual priority space makes the system susceptible to the real-time issue of *rate-monotonic priority inversion*[1] [3].

Several solutions have been proposed to overcome these issues by employing threads only: In the Solaris OS kernel, ISRs can be promoted to threads upon request, enabling blocking semantics in hardware event handlers [8]. At that point, the management and scheduling of the ISR control flow is switched from the hardware to the OS (see Line 4 in Table I). However, to implement this flexibility, Solaris still has to keep the dual priority space, with "ISR threads" having a higher priority than all other, "regular" threads. To tackle that problem, solutions have been proposed in which short ISRs always activate a corresponding thread to run and then terminate immediately [3], [4]. This way, all event handlers (hardware and software) are scheduled and dispatched by the OS as threads in a single priority space. However, since OS-managed threads incur significant software overhead compared to ISRs, this advantage comes at a major performance and latency cost: The latency of ISRs becomes 3–10 times higher—even if parts of the ISR processing are outsourced to an external co-processor [21].

That is why in previous work on the SLOTH embedded kernel, we have proposed the opposite approach: to have *threads run as ISRs* [5]. By triggering the corresponding interrupt from within kernel software when activating a thread, SLOTH relies on a single priority space—the interrupt priority space managed by the hardware—and lets the hardware interrupt arbitration system perform the priority-based scheduling and dispatching (see Line 5 in Table I). The SLOTH kernel, which implements this idea, is simple, small, and fast in scheduling and switching control flows (2–7 times faster than in a traditional kernel). Nevertheless, the SLOTH kernel has a significant drawback: It does not support blocking threads, which need an execution stack of their own. Since it only supports run-to-completion control flows, its execution and preemption pattern is strictly last-in, first out (i.e., stacked)—which is why it is a perfect match for execution using an interrupt controller with multiple interrupt levels, since interrupt activations are also strictly stacked and preempted based on their priorities.

*This work was partly supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-1 and SFB/TR 89.

[1]This term describes the phenomenon that a high-priority thread can be interrupted and delayed by a low-priority ISR because the hardware-managed ISR priorities are inherently higher than the OS-managed thread priorities.

		Activation	Scheduling/Dispatching	Execution Semantics
1	Traditional Threads / OSEK Extended Tasks	by OS	by OS	Blocking
2	Traditional ISRs	by HW	by HW	Run-to-Completion
3	OSEK Basic Tasks	by OS	by OS	Run-to-Completion
4	Solaris IRQ Threads [8]	by HW	by HW → by OS	Blocking
5	SLOTH Tasks [5]	by OS or HW	by HW	Run-to-Completion
6	SLEEPY SLOTH Threads	by OS or HW	by HW	Run-to-Completion or Blocking

TABLE I
TYPES OF CONTROL FLOWS AND THEIR PROPERTIES (ACTIVATION AND SCHEDULING/DISPATCHING BY HARDWARE (HW) OR THE OPERATING SYSTEM (OS), AS WELL AS EXECUTION SEMANTICS) IN TRADITIONAL OPERATING SYSTEMS, IN OSEK, IN SOLARIS, AND IN SLOTH, COMPARED TO THE THREAD ABSTRACTION PROVIDED BY SLEEPY SLOTH.

B. About This Paper

In this paper, we aim to remedy this situation by designing a new thread abstraction that combines the advantages of the SLOTH control flow with blocking functionality. These new kinds of threads can be triggered by software events and hardware events, they can have run-to-completion or blocking semantics, and they run in a single priority space (see Line 6 in Table I). We argue that this new thread abstraction combines the best properties of threads—blocking flexibility—and traditional ISRs—low execution latency. Our proposed thread model is *flexible* since it allows those control flows to block that need to block, and it is *fast* since it relies on commodity interrupt hardware to perform scheduling and dispatching in hardware. Thus, in the SLEEPY SLOTH kernel[2] that we have implemented as an extension of the original SLOTH kernel, we have removed the artificial distinction between threads and ISRs: *Threads can be interrupt handlers and interrupt handlers can be threads*. SLEEPY SLOTH implements the widely used OSEK operating system standard [18], and it outperforms a leading commercial implementation of that standard by a factor of 1.3 to 19.0.

This paper provides the following contributions:

- We present a new universal thread abstraction and discuss the challenges in implementing it to provide blocking flexibility while being scheduled and dispatched efficiently using interrupt hardware (see Section III).
- We present our SLEEPY SLOTH kernel design that tackles these challenges by providing a tailored task prologue and a static analysis engine (see Sections IV and V).
- We evaluate our prototypical SLEEPY SLOTH implementation on the Infineon TriCore microcontroller, showing that it provides the extra blocking flexibility without harming performance and latency for tasks that do not need it (see Section VI).
- We discuss the necessity for different types of control flows in operating systems and the general applicability of the hardware-centric SLEEPY SLOTH approach (see Section VII).

II. SLOTH REVISITED

The original SLOTH kernel described in [5] implements the BCC1 conformance class of the OSEK operating system standard [18], which is omnipresent in the automotive industry. In the following section, we briefly describe OSEK's features and the corresponding IRQ-based SLOTH design for them; Figure 1 illustrates this by showing an example SLOTH system. Note that OSEK systems (and therefore also SLOTH systems) are configured statically, and, thus, all control flows and their priorities are known at compile time.

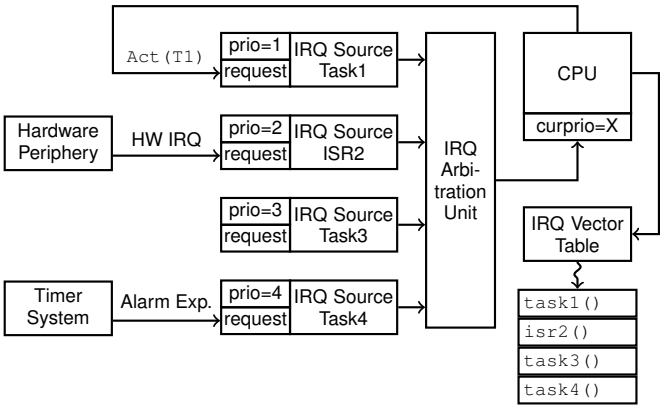


Fig. 1. Design of a SLOTH system, using interrupt handlers for the implementation of threads. The interrupt sources have a statically configured priority and are either triggered synchronously by the CPU through a system-service call (e.g., Task1), through hardware-periphery IRQs (e.g., ISR2), or through the timer system after setting a task alarm (e.g., Task4).

a) Task Management: The main idea behind SLOTH is to design every task as an interrupt handler. Every task is assigned an IRQ source with the corresponding priority at compile time, and the corresponding interrupt handler is set to be the user task function. Activation of a task by another control flow is performed by setting the corresponding IRQ request bit by the kernel software (see Task1 in Figure 1), letting the IRQ controller's arbitration unit decide about preemption depending on the current CPU priority and pending IRQ priorities. A task terminates by returning from the interrupt routine, again relying on the hardware to schedule the task with the next-highest priority.

b) Resource Management: Resources are OSEK's way of designating critical sections in applications, which are synchronized against preemptions by competing tasks using a stack-based priority ceiling protocol. By acquiring a resource, a task's priority is lifted to the ceiling priority of all tasks that could acquire that resource; SLOTH simply raises the CPU

[2]The name honors the deadly sin and the lazy animal breed named *sloth*, which likes its "control flows" to *sleep* (i.e., to block).

priority to accomplish this. This way, the dispatching of a task that competes for the same resource is delayed until after the critical section is left. Upon release of a resource, the priority is lowered to the previous level, potentially dispatching delayed activated tasks.

c) Alarm Management: Alarms are OSEK's timer abstraction and allow activating a task after a specified amount of time has elapsed. To every task that is configured at compile time to be activated by an alarm at run time, SLOTH assigns an IRQ source that is connected to the timer system (see Task4 in Figure 1). This way, when the timer expires, the task is automatically scheduled by the hardware by triggering the corresponding interrupt, dispatching it depending on the system's current priority situation.

d) ISR Management: OSEK distinguishes between two types of interrupt service routines (ISRs): Category-2 ISRs are allowed to perform system calls and therefore need to be synchronized with tasks in order not to corrupt kernel state, whereas the kernel is oblivious of category-1 ISRs, which are *not* allowed to invoke the kernel. In SLOTH, there is no difference in the handling of category-2 ISRs and tasks; the kernel is oblivious to whether the interrupt request was triggered by a hardware periphery device (see ISR2 in Figure 1) or by software (see Task1 in Figure 1).

III. SLEEPY SLOTH REQUIREMENTS

The only class of OSEK system calls that the original SLOTH kernel does not implement is the one that manages OSEK events. Events are OSEK's means for task notification and, therefore, its means for a task to block (system call `WaitEvent()`) and to be unblocked (`SetEvent()`). OSEK calls tasks that are allowed to potentially block *extended tasks* (which need a *full context* of their own, including a task stack, to be continued in their execution), whereas run-to-completion tasks are called *basic tasks* (which can share parts of their contexts, including their stack, since they preempt each other in a strictly last-in, first-out—that is, stacked—manner).

The overall goal in extending SLOTH to SLEEPY SLOTH is therefore to provide applications the ability to include extended blocking tasks while preserving SLOTH's performance and latency benefits by having threads run as interrupt handlers.

A. SLEEPY SLOTH *Challenges*

In the original SLOTH kernel, where only basic run-to-completion tasks are present, the control flow hierarchy is strictly stacked and strictly nested, which means that control flows are only preempted by higher-priority control flows and returned to after those have run to completion. The SLOTH execution model corresponds exactly to the one that traditional ISRs support using multi-level interrupt controllers, plus the ability to be activated by the operating system (compare Lines 2 and 5 in Table I).

The first and main challenge in SLEEPY SLOTH, however, is to be able to suspend task execution and resume its execution later, which interrupt controllers do not support for interrupt handler execution. This is due to the fact that interrupt handlers are supposed to run to completion *transparently* to the interrupted control flow. Thus, SLEEPY SLOTH needs to find a way to implement both the *suspension* of a blocked ISR and the *re-activation* of an unblocked ISR, saving and restoring its full context including its stack appropriately.

Second, by nature, interrupts are *asynchronous* in their occurrence; that is, no prediction can be made as to where exactly a control flow yields the CPU when interrupted. Thus, performing the necessary context and stack switch in the *preempted* control flow *before* dispatching is impossible in a SLOTH-like system with hardware-triggered preemptions, since the interrupt scheduler and dispatcher are provided by the hardware.

The third challenge regards the execution efficiency of the resulting SLEEPY SLOTH system: The added flexibility for the application developer should not come at the price of lowered system performance. Especially the latencies for scheduling and executing basic run-to-completion tasks should remain comparable to the original SLOTH kernel.

B. Hardware Environment and Requirements

SLEEPY SLOTH's requirements on the hardware platform shall remain the same as stated in [5] for SLOTH; thus, the approach is applicable to any platform with a modern multi-level interrupt controller:

1) The platform needs to be able to trigger interrupts from within software—for instance, by setting a bit in a dedicated register or by offering a special instruction for that matter. This is needed to implement synchronous task activation.
2) The number of available interrupt priorities needs to be at least as high as the number of threads and ISRs in the SLEEPY SLOTH system, since every thread and ISR is assigned a dedicated interrupt source and priority (plus one dedicated priority for each resource; see Section V-E). Our two prototypes run on the Infineon TriCore and the ARM Cortex-M3 platforms, both of which feature 256 interrupt priority levels, enabling SLEEPY SLOTH systems with about 256 real-time control flows.[3] The assignment of a dedicated priority slot per task does not allow for multiple tasks per (semantically equal) priority if the order of activations needs to be preserved.

IV. SLEEPY SLOTH OVERVIEW

In this section, we present the central design ideas in SLEEPY SLOTH to meet the requirements and challenges discussed before. A prototypical control flow in the system illustrates these ideas by example, followed by a description of the SLEEPY SLOTH analysis and generation architecture to provide application-tailored context switching.

[3]For the rest of this paper, we assume higher priority levels to be assigned to higher priority numbers (as is the case on the Infineon TriCore platform).

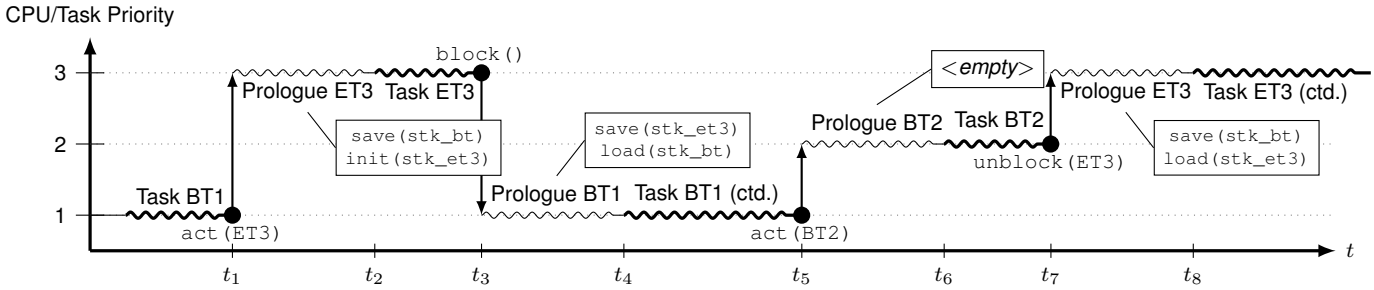


Fig. 2. Example control flow in a SLEEPY SLOTH system with two basic tasks, BT1 and BT2 (with priorities 1 and 2, respectively), which run to completion and share a common stack, `stk_bt`, and one extended task, ET3 (with priority 3), which can block during its execution and therefore has a stack of its own, `stk_et3`. The figure does not reflect timing proportions, as the task prologues are usually very short compared to the task functions themselves.

A. *Central Design Ideas*

In order to meet the requirements and tackle the challenges stated before, SLEEPY SLOTH is based on three central design ideas.

1) The Task Prologue: SLEEPY SLOTH provides support for blocking tasks by prepending a task prologue to every task function. This prologue is executed whenever a task is dispatched by the interrupt hardware, both when the task is about to run for the first time and when its execution is resumed after being blocked or preempted. The task prologue is the single point to decide whether to save the stack of the interrupted task and whether to restore or initialize the stack of the dispatched task. Note that parts of the task context are saved by the hardware automatically upon dispatching an interrupt handler. The prologue concept is the key enabler for interrupt *re*-activation/resumption, and it addresses the challenge that IRQs occur asynchronously by performing the stack switch (if necessary) in the newly dispatched successor control flow.

2) Threads as ISRs: To provide a performance comparable to SLOTH, SLEEPY SLOTH also relies on the hardware to do as much as possible for it. This mainly entails the kernel relying on the interrupt system to do the scheduling and dispatching work for it by having all threads run as ISRs with appropriate priorities. Thus, SLEEPY SLOTH follows the rule to have every task run as an ISR as often as possible, and to only run it as a thread with a stack of its own if the application semantics needs it (in order to be able to block).

3) Static Analysis and System Generation: To achieve its goal to enable efficient execution for basic non-blocking tasks, SLEEPY SLOTH makes use of static application knowledge available at compile time. By statically analyzing the control flow configuration of the application, it can infer information about which task or ISR can preempt which other tasks or ISRs at run time. This information is then used to generate tailored task prologues that omit unnecessary run time checks for preemption conditions before performing stack switches or not. Thus, SLEEPY SLOTH evaluates those conditions statically where possible and dynamically otherwise.

B. SLEEPY SLOTH *Example Control Flow*

Figure 2 shows an example trace of a SLEEPY SLOTH application with two basic tasks, BT1 and BT2 (which share a common stack, `stk_bt`), and a high-priority extended task, ET3 (with a stack of its own, `stk_et3`). Suppose that at some point, only BT1 is running, which then activates ET3 (t_1). ET3 is immediately dispatched because of its high priority—prepended by its prologue, which saves BT1's stack and initializes ET3's stack before executing the actual user function for ET3 (t_2).

ET3 then blocks and releases the CPU, giving control back to BT1 (t_3). Its prologue notes that it has interrupted an extended task and therefore performs a full context switch by saving ET3's stack and loading the common BT stack before resuming execution (t_4). The following activation of BT2 (t_5) dispatches the BT2 prologue, which observes that it has interrupted a basic task and therefore starts executing the task function at t_6 without having to switch stacks.

BT2 then unblocks ET3, triggering its prologue once again (t_7). At this point, the ET3 prologue saves the basic task stack and restores its own stack, resuming execution after the point it had blocked at (t_8).

C. SLEEPY SLOTH *Architecture*

Like OSEK OS, SLEEPY SLOTH is configured completely statically; that is, all threads and ISRs and their priorities are known and configured before compile time. Many crucial parts of the SLEEPY SLOTH system are therefore generated specifically for an application after being analyzed as depicted in Figure 3.

As its input, the SLEEPY SLOTH system takes the configuration of the application as specified by the application programmer. This comprises application objects such as tasks, interrupt service routines, resources, and alarms, together with their names, priorities, and other properties such as whether a task is basic or extended (i.e., it is allowed to block) or which tasks share a given resource.

1) Analysis: The SLEEPY SLOTH analyzer then performs several kinds of analyses on that configuration to provide the subsequent generation step with additional input. First, the configured application control flows (i.e., tasks, category-1 ISRs, and category-2 ISRs) are analyzed for their interactions

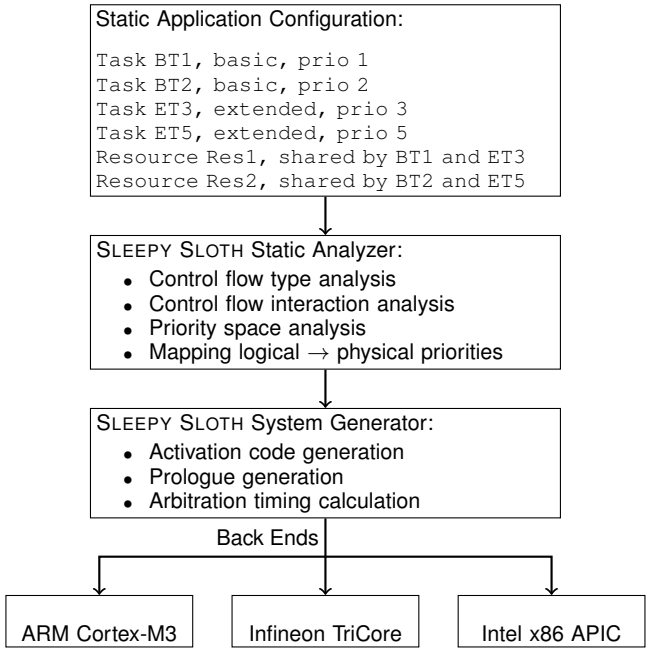


Fig. 3. SLEEPY SLOTH configuration analysis and generation architecture.

based on their properties and priorities. Internally, this analysis step calculates a preemption graph, which encompasses information about which control flow can be preempted by which other control flows and, therefore, which of the preemptions actually need a stack switch (e.g., a basic task preempting another basic task does not need one). Second, the priority space as specified by the application programmer is analyzed, and the given logical priorities are mapped to physical interrupt priorities. This step comprises both the compacting of the logical priority space if the priority configuration as provided by the application is sparse, and it assigns additional, dedicated priority slots in between control flow priorities for resources. This way, a task holding a resource can be unambiguously identified by its execution priority (see Section V-E).

2) Generation: The actual SLEEPY SLOTH generator then generates application-specific code for the system. This mainly entails code for the activation of a task by setting the interrupt request bit in the appropriately configured IRQ source, and it entails the prologue code for every application task. This prologue code includes only those run time checks that can actually occur in the configured system depending on the calculated preemption graph. Furthermore, as interrupt arbitration systems bear latencies that the kernel needs to consider and as those latencies depend on the involved interrupts, the corresponding timing properties are also calculated in that module (see Section V-C).

3) Back Ends: The back end parts of the SLEEPY SLOTH generator finally generate the architecture-specific parts of the code, which are then compiled with static, non-generated system code (both architecture-dependent and architecture-independent) to produce the combined binary of the SLEEPY SLOTH application and kernel. By producing a single compilation unit using the C preprocessor, the combined application and kernel code is subject to comprehensive compiler optimization, which inlines many of SLEEPY SLOTH's system calls due to their brevity.

V. SLEEPY SLOTH IMPLEMENTATION

The following section first details the SLEEPY SLOTH task prologue and how it interacts with explicit scheduling points such as task termination and task blocking and unblocking. Additionally, resources need to be re-designed in SLEEPY SLOTH, and basic tasks are handled specially to preserve SLOTH's performance characteristics for them as far as possible.

Note that all additional system services and the task prologues in SLEEPY SLOTH as well as SLOTH's original system calls have a statically bounded worst-case execution time for real-time operation.

A. Task Prologue

Whenever the hardware dispatches an interrupt handler that is assigned to an extended task, the task prologue, which is at the core of the SLEEPY SLOTH design (see also Section IV-A) takes action as outlined in Figure 4, with IRQs disabled by the hardware at that point. Depending on the actual task, some of the condition checks and steps are omitted by SLEEPY SLOTH's static analyzer for situations that can never occur at run time (see also Section IV-C).

First, the prologue saves the extended context of the interrupted task (i.e., those registers that have not been saved by the hardware when dispatching the interrupt handler[4]) to the corresponding task stack, whose stack pointer in turn is saved to a kernel context array (Step 1 in Figure 4). Next, the prologue checks if the interrupted task was either preempted or blocked, or if the interrupted task terminated (2). If it did *not* terminate, the prologue re-activates the task to be continued later by triggering the task's interrupt source at the priority it was running at (2a). This is what ET3's prologue does at t_1 and t_7 in the example control flow in Figure 2. Note that the interrupted task's priority might have been raised at the time of preemption due to the possession of a resource in combination with the stack-based priority ceiling protocol; in that case, the continuation of the task needs to be treated specially (see Section V-E). The check for that condition itself is done via a kernel-maintained bit array (`hasSeenCPU[]`). After that, the kernel variable holding the current task is set to the dispatched task ID, which corresponds to the current IRQ number (3).

If the dispatched task has run before (checked by comparing its `hasSeenCPU` property; Step 4 in Figure 4), its context is restored from the kernel context array (entailing its stack and registers; 5a), IRQs are enabled (6a), and execution of the task is continued by returning using the return address in the saved context (7a). If it has *not* run before, its context is initialized (entailing re-setting its stack pointer; 5b) and its `hasSeenCPU` property is set to true to be considered by

[4]Note that, with appropriate compiler support, the prologue can save only those registers that are actually *used* by the dispatched task.

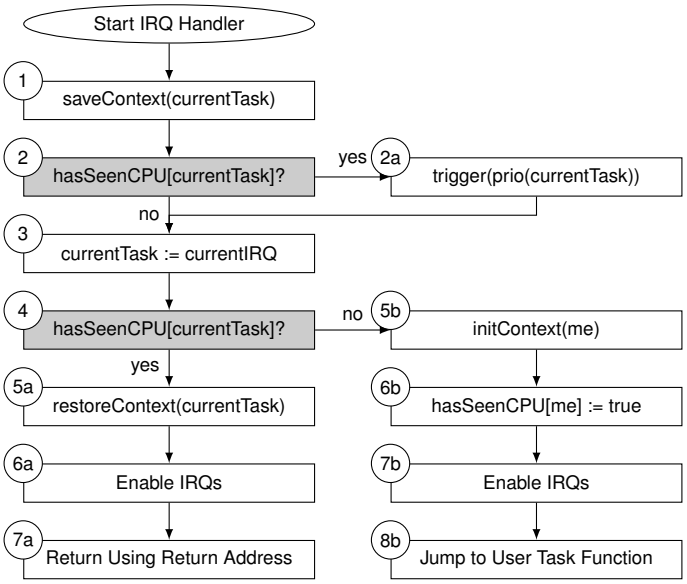


Fig. 4. State diagram of the SLEEPY SLOTH task prologue for extended tasks in its maximum version. Depending on the results of SLEEPY SLOTH's static analysis, each task prologue is tailored to the functionality actually needed at run time.

further preemptions by other tasks (6b). Eventually, the task prologue enables IRQs (7b) and jumps to the actual user task function to start executing user code (7b).

B. Task Termination

Task termination in SLEEPY SLOTH differs from the way it works in the original SLOTH since a stack switch might be needed after termination—for instance, if another task (with the next-highest priority) was unblocked and needs to be continued in its execution after termination of the current task. Again, SLEEPY SLOTH solely relies on the prologue of the next task to determine if a context switch is needed or not. The next task is scheduled and dispatched by the hardware by letting the terminating task set the CPU priority to zero. The interrupt system then determines the next-highest priority task that is ready to run. Before that, the terminating task indicates to the next prologue that it has terminated by re-setting its `hasSeenCPU` flag to false. This way, the interrupt source of the terminating task will not be re-triggered for continued execution (see Steps 2 and 2a in Figure 4).

C. Task Blocking

The main additional system call that SLEEPY SLOTH provides over the original SLOTH kernel is `WaitEvent()`. The implementation compares the event mask of the current task to the event mask to be waited for, and, if they do not match, blocks the task. This is done by disabling the task's IRQ source; this way, it will not be considered in the interrupt arbitration to determine the highest-priority interrupt to be handled. After that, the CPU is yielded by setting the CPU priority to zero (much in the same way that a task terminates; see also Section V-B) and letting pending interrupts (corresponding to tasks that are ready to run) trigger (see also t_3 in the example control flow in Figure 2). The whole blocking mechanism is synchronized against preemption by other tasks and interrupt handlers by disabling all IRQs at the beginning and re-enabling them at the end:

```
void WaitEvent(EventMaskType mask)
{
  if ((eventMask[currentTask] & mask) == 0) {
    /* none of the events has already been set */
    eventsWaitingFor[currentTask] = mask;
    /* block task */
    disableIRQs();
    disableIRQSource(currentTask);
    setCPUPrio(0);
    waitForArbitration();
    enableIRQs(); /* point of preemption */
  }
}
```

Note that, like when activating a task in the original SLOTH kernel [5], modifying the hardware interrupt priority situation of current and pending priorities—in this case, disabling an IRQ source and setting the CPU priority to zero—might require synchronization with the interrupt arbitration system. This is due to latencies during the arbitration in the interrupt system; for instance, for the TriCore microcontroller platform, those latencies are defined by Infineon in an application note [6]. The maximum number of clock cycles that the arbitration process takes depends on several system properties like the system frequency and the number of involved IRQ sources—and, therefore, the number of SLEEPY SLOTH tasks in the system. Thus, this number can be calculated statically by the static analyzer (see Section IV-C) and is inserted in the form of `nop` instructions in `waitForArbitration()` before enabling the IRQs again[5]. This way, SLEEPY SLOTH ensures that the defined point for preemption after blocking the current task will always be the point after the IRQ enable instruction, independent of the current state and latency of the interrupt arbitration system. Note that this arbitration delay makes up for most of the hardware-induced costs, which are significantly lower than any software-induced costs (see also evaluation in Section VI).

D. Task Unblocking

Tasks are unblocked in OSEK by setting one of the events that the task has been waiting for. SLEEPY SLOTH's system call `SetEvent()` therefore first checks if that condition is met, and then it unblocks the task by re-enabling its IRQ source and triggering its IRQ (see also t_7 in the example control flow in Figure 2). This makes the interrupt controller consider the task in its priority arbitration mechanism and schedule the task according to the system's priority situation:

```
void SetEvent(TaskType id, EventMaskType mask)
{
  eventMask[id] |= mask;
  if ((eventMask[id] & eventsWaitingFor[id]) != 0) {
    /* at least one of the events that
```

[5]Before waiting for the arbitration by executing `nop` instructions, the TriCore SLEEPY SLOTH implementation also reads back the interrupt register to synchronize hardware and software as demanded by Infineon's specification [6].

```
      * the task has been waiting for is set */
    eventsWaitingFor[id] = 0;
    /* unblock task */
    disableIRQs();
    enableIRQSource(id);
    waitForArbitration();
    enableIRQs(); /* point of preemption */
  }
}
```

As elaborated in the description of the task blocking mechanism (see Section V-C), due to the synchronization with the interrupt arbitration system, the defined point for preemption will be the enable IRQ instruction at the end of the system call.

E. Resources in SLEEPY SLOTH

In SLEEPY SLOTH, OSEK resources, used to synchronize accesses to critical sections by application tasks by raising their priorities according to a stack-based priority ceiling protocol, need to be handled in a special way. Consider the case where a task that had acquired a resource is preempted by a higher-priority extended task, which performs a stack and context switch (see example control flow in Figure 5). In that case, the preempted task needs to be re-activated *at the raised priority* of the resource when being continued in its execution (t_d in Figure 5). In the original SLOTH kernel, a resource's ceiling priority was set to the highest priority of all tasks that can acquire that resource. In order for SLEEPY SLOTH to be able to distinguish between an activation of that highest-priority task and a re-activation of a task that had acquired that resource, the ceiling priority is set to one level higher than that, to a dedicated priority slot. The resource is therefore also assigned a dedicated IRQ source.

That resource IRQ source is only triggered when a task that had acquired the resource is preempted by a higher-priority task (t_b in Figure 5), whose prologue will re-trigger the resource IRQ source at its ceiling priority (see Step 2a in Figure 4 and t_b in Figure 5). This way, the dedicated resource IRQ handler will be dispatched once the CPU priority is lowered again—for instance, when the higher-priority task terminates or blocks (t_d in Figure 5). The resource IRQ handler then checks which task had acquired it (noted down when the corresponding system call `GetResource()` is executed) and restores its context. The execution priority is left unchanged at the resource ceiling priority, which is the correct priority for the preempted task to continue its execution at (t_e in Figure 5).

F. Basic Tasks in SLEEPY SLOTH

The main goal in designing SLEEPY SLOTH is to support blocking extended tasks while preserving SLOTH's advantages and performance as far as possible for non-blocking basic tasks. Since basic tasks run to completion in a strictly priority-ordered stacked way, SLEEPY SLOTH, just like SLOTH, lets all basic tasks run on a single stack. This makes context switches between basic tasks—both on preemption and on termination—extremely lightweight and fast, since the hardware automatically saves and restores part of the register set upon interrupt entry and return and no additional stack switch is needed.

In SLEEPY SLOTH, however, additional overhead is incurred to *determine* if a stack switch is needed—that is, if either the interrupted or the newly dispatched task or both are extended tasks. This property is configured by the application programmer at compile time and stored in a bit field that is used for that check to keep it as fast as possible. Apart from that, during times in the application when only basic tasks are scheduled and dispatched, the overhead incurred by the SLEEPY SLOTH kernel is minimal and comparable to the one incurred by SLOTH (see also the empty prologue at t_5 in the example control flow in Figure 2 and the evaluation in Section VI). Additionally, if the static configuration permits, run time checks can be omitted altogether in the tailored basic task prologues, further reducing overhead (see Section IV-C).

VI. EVALUATION

We have evaluated our SLEEPY SLOTH reference implementation on the Infineon TriCore platform, a 32-bit microcontroller widely used in the automotive industry, featuring a RISC load/store architecture and a Harvard memory model. The interrupt system has 256 priority levels and the TC1796 chip that we use has about as many interrupt sources, whose registers are mapped into memory, enabling SLEEPY SLOTH to modify their enable and pending bits for its purpose. A specialty of the TriCore platform is its separation of the data stack from the call stack, which is managed in separate so-called context save areas; SLEEPY SLOTH therefore has to save and restore both stacks when switching between two extended tasks. We clocked the chip at 50 MHz (clock cycle of 20 ns), although we state our measurements in numbers of clock cycles to be frequency-independent.

To assess the performance gain achieved by implementing thread scheduling using interrupt hardware, we have performed several microbenchmarks comparing SLEEPY SLOTH to a leading commercial OSEK implementation. We set up several test applications for that purpose and compiled them unaltered for both SLEEPY SLOTH and the commercial OSEK, measuring only the latencies of the involved system calls. All numbers were obtained using a Lauterbach hardware trace unit and averaged over at least 10,000 samples.

A. Basic Task System

In [5], we published the execution time numbers of the basic SLOTH kernel for seven microbenchmarks related to task switching. Table II reproduces those numbers and additionally shows the measurements for the SLEEPY SLOTH kernel and the commercial OSEK implementation.

For one, the results show that the numbers for SLOTH and SLEEPY SLOTH are almost identical. This is because SLEEPY SLOTH can be entirely tailored to the functionality that the application actually needs; thus, in a setting with *only* basic tasks, SLEEPY SLOTH will behave identically to SLOTH. The occasional and small deviations are due to bug fixes since the old measurements were performed.

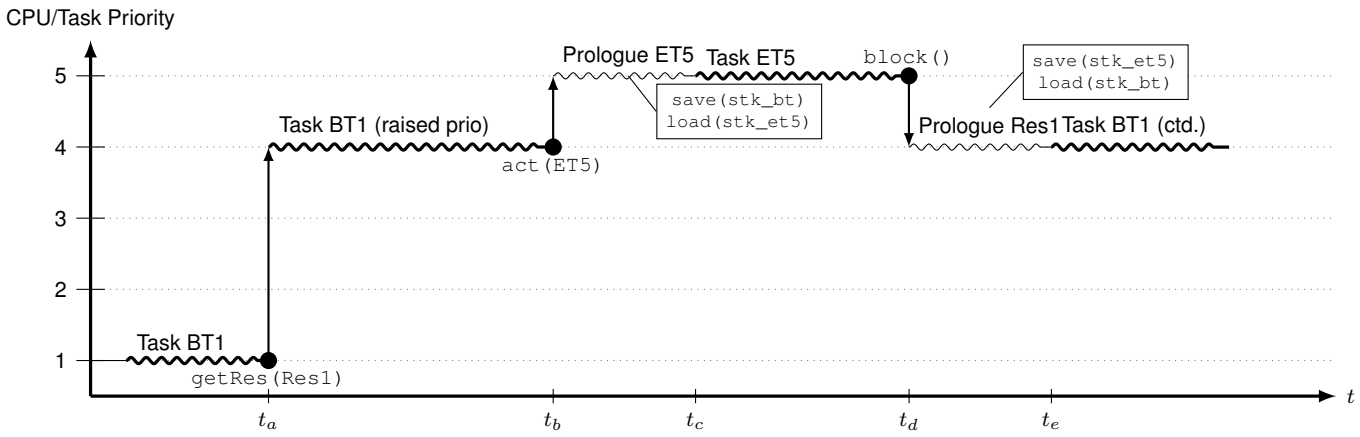


Fig. 5. Example control flow with an OSEK resource in a SLEEPY SLOTH system with one basic task BT1 (with priority 1) and one extended task ET5 (with priority 5). BT1 accesses a resource it shares with extended task ET3 (with priority 3); in SLEEPY SLOTH, the resource is therefore assigned the dedicated priority slot 4 due to the stack-based priority ceiling protocol.

Test Case	SLOTH [5]	SLEEPY SLOTH	OSEK
A1) Task Activation w/o Dispatch	34	38	75
A2) Task Activation w/ Dispatch	60	60	273
A3) Termination w/ Dispatch	14	14	266
A4) Chain w/ Dispatch	67	67	327
A5) Resource Acquisition	19	18	66
A6) Resource Release w/o Dispatch	14	16	128
A7) Resource Release w/ Dispatch	36	38	280

TABLE II
EVALUATION OF **BASIC** TASK SWITCHING MICROBENCHMARKS WITHOUT STACK SWITCHES (EXECUTION TIME IN CLOCK CYCLES), COMPARING THE ORIGINAL SLOTH KERNEL TO SLEEPY SLOTH AND A COMMERCIAL OSEK IMPLEMENTATION.

Test Case	SLEEPY SLOTH	OSEK	Speed-Up
B1) Extended Task Activation w/ Dispatch	121	286	2.4
B2) Blocking w/ Dispatch	143	224	1.6
B3) Unblocking w/ Dispatch	120	205	1.7
B4) Event Mask Clearing	6	32	5.3
B5) Extended Task Termination w/ Dispatch	82	275	3.4
B6) Extended Task Chain w/ Dispatch	113	392	3.5

TABLE III
EVALUATION OF **EXTENDED** TASK SWITCHING MICROBENCHMARKS WITH STACK SWITCHES (EXECUTION TIME IN CLOCK CYCLES), COMPARING THE SLEEPY SLOTH KERNEL TO A COMMERCIAL OSEK IMPLEMENTATION.

On the other hand, both SLEEPY SLOTH and SLOTH outperform the commercial OSEK implementation, which uses a software scheduler and dispatcher for operation. The achieved speed-up is between 2.0 and 19, with higher speed-ups for those test cases that not only involve a scheduling decision but also dispatching a new task.

B. Extended Task System

To assess the performance of SLEEPY SLOTH’s extended task features, we configured an application that consists only of extended tasks. Hence, every task switch needs to perform a stack switch; the microbenchmark numbers are shown in Table III.

Although SLEEPY SLOTH uses the interrupt controller, whose run-to-completion model does not perfectly fit SLEEPY SLOTH’s blocking threads, it still outperforms the commercial OSEK implementation for all test cases. SLEEPY SLOTH’s extended task switches are considerably slower (82–143 cycles) than its basic task switches (14–67 cycles) due to the stack switches and involved decisions, but it is still considerably faster than the commercial OSEK, which has a software scheduler (205–392 cycles, resulting in a speed-up of 1.6 to 3.5).

C. Mixed Task System

One of SLEEPY SLOTH’s original goals and challenges was to be able to provide the flexibility of blocking tasks without influencing the task switch performance for basic tasks too much (see Section III). On application granularity, we have shown this by measuring a purely basic task system on SLEEPY SLOTH (see Section VI-A). To evaluate this property on task granularity, we configured a single application with two basic tasks and two extended tasks, and we measured different types of preemptions between those tasks (see Table IV).

As expected after analyzing the results from the basic-only and extended-only benchmarks, SLEEPY SLOTH is also faster than the software-based commercial OSEK in the mixed task case (speed-up between 1.3 and 9.7). The numbers for transitions between two basic tasks are on SLEEPY SLOTH’s low end (79, 29, and 98 cycles for C1, C5, and C8) and compare to the corresponding benchmarks in the basic-only version (60, 14, and 67 cycles for A2, A3, and A4). Thus, in a mixed task system, basic task scheduling in SLEEPY SLOTH is burdened by an overhead of 15 to 31 cycles, added to an already low base overhead of 14 to 67 cycles.

D. Evaluation Summary

The main goals in the design and implementation of SLEEPY SLOTH were to provide the flexibility of blocking tasks while relying on hardware as much as possible in order to

Test Case	Task Type Transition	Stack Switch	SLEEPY SLOTH	OSEK	Speed-Up
C1) Task Activation	Basic → Basic	w/o Stack Switch	79	281	3.6
C2) Task Activation	Basic → Extended	w/ Stack Switch	116	286	2.5
C3) Blocking	Extended → Basic	w/ Stack Switch	168	216	1.3
C4) Unblocking	Basic → Extended	w/ Stack Switch	118	205	1.7
C5) Task Termination	Basic → Basic	w/o Stack Switch	29	280	9.7
C6) Task Termination	Extended → Extended	w/ Stack Switch	94	344	3.7
C7) Task Termination	Extended → Basic	w/ Stack Switch	86	282	3.3
C8) Task Chain	Basic → Basic	w/o Stack Switch	98	393	4.0

TABLE IV
EVALUATION OF **MIXED**—THAT IS, BOTH EXTENDED AND BASIC—TASK SWITCHING MICROBENCHMARKS (EXECUTION TIME IN CLOCK CYCLES), COMPARING THE SLEEPY SLOTH KERNEL TO A COMMERCIAL OSEK IMPLEMENTATION.

provide good performance for extended tasks without harming performance for basic task scheduling (see Section III). The execution time measurements on the TriCore platform show that the SLEEPY SLOTH implementation is able to meet these goals.

First, no performance penalty at all is incurred for systems that only need basic run-to-completion tasks. In contrast, the software-based commercial implementation shows about the same overhead for switches between basic tasks as for switches between extended tasks.

Second, although the SLEEPY SLOTH implementation is not as simple as is SLOTH's, extended task scheduling is not slower than in the commercial implementation with a software scheduler. In fact, SLEEPY SLOTH is able to outperform the commercial OSEK by a factor of 1.6 to 3.5.

Third, in a mixed task system, the task switch overhead scales with the demand of the involved tasks. Task switches between basic tasks in a mixed SLEEPY SLOTH system are cheaper than task switches between extended tasks, which need additional stack switches.

The real-time application benefits from SLEEPY SLOTH by suffering lower system call latencies compared to a software scheduler kernel, positively affecting the response times it asserts itself to the user of the system. The actual performance gain depends on the actual application and the ratio of executed application code to executed system code. Additionally, since all tasks and ISRs run in the same priority space, the analysis of the system's real-time properties is facilitated (see also discussion in Section VII-A). Note that the numbers discussed in this section include all hardware-related preemption costs such as waiting for the bus arbitration (see also Section V-C); nevertheless, the SLEEPY SLOTH system still outperforms the software-based commercial kernel.

VII. DISCUSSION

SLEEPY SLOTH combines the flexibility of an off-the-shelf embedded kernel by providing blocking threads with the efficiency of a purely interrupt-driven system. In this section, we discuss the necessity for different control flow types in embedded systems, and we discuss the general applicability of the SLEEPY SLOTH approach.

A. Control Flows in Embedded Systems

As sketched in the introduction and as presented in Table I, the status quo concerning control flows in embedded systems is two worlds. On the one hand, an application programmer needs to decide for a control flow to be a *thread* if it needs to be provided with the ability to block its execution. On the other hand, a control flow needs to be an *ISR* if it can be activated asynchronously by a hardware periphery device. SLEEPY SLOTH threads, however, provide both of those properties in one universal abstraction, leading to several advantages for the real-time application.

For one, SLEEPY SLOTH threads can interact and synchronize freely using common synchronization and notification mechanisms, such as OSEK resources and events. In traditional systems, communication between threads and ISRs and synchronization of threads and ISRs is complicated or even impossible to achieve.

Additionally, since SLEEPY SLOTH threads run in one common priority space—the interrupt priority space—, the conditions for rate-monotonic priority inversion [3] are eliminated. In traditional kernels, it is impossible to have a thread run at a higher priority than an ISR (since interrupt priorities are implicitly higher than thread priorities), even though the application might demand it. SLEEPY SLOTH enables the application to freely distribute priorities among its control flows depending solely on its *semantic* requirements.

B. General Applicability of the Approach

The nature of both SLOTH and SLEEPY SLOTH implies that the actual *implementation* on a platform with a specific interrupt controller is highly hardware-dependent, since the kernels rely on efficient hardware mechanisms to perform the scheduling work for them. However, the matter of the fact is that the actual kernel code is very small and has a clear abstraction boundary that needs to be mapped to the hardware platform, resulting in a manageable porting effort. The fact that a kernel can reach new levels of efficiency by tailoring its implementation to the target hardware has been exploited lots of times before—for instance, when making microkernel inter-process communication more efficient [10].

Our SLEEPY SLOTH prototype implements the OSEK operating system specification to be able to compare it to other implementations without adapting the benchmark applications

(see also evaluation in Section VI). The approach to implement blocking threads as interrupt handlers with a tailored prologue, however, is applicable to any event-driven real-time operating system with static priorities. *Dynamic-priority* systems, on the other hand, which need to re-prioritize threads at run time, are not worthwhile implementing using the SLEEPY SLOTH approach, since dynamic re-configuration of interrupt sources and preempted interrupt handlers is usually very costly. Generally speaking, SLEEPY SLOTH is suitable to implement the most well-known fixed-priority scheduling algorithms such as rate-monotonic [12] and deadline-monotonic scheduling [13]. Advanced scheduling mechanisms such as the general priority ceiling protocol, the priority inheritance protocol, and aperiodic servers can still be implemented in SLEEPY SLOTH, since they only require *occasional* re-prioritization of control flows by re-configuring the corresponding interrupt sources.

Despite running all threads as ISRs, exception traps can also be accommodated in SLEEPY SLOTH. Traps cannot be masked and do not interfere with the interrupt priorities that SLEEPY SLOTH uses for its purposes. Thus, they can simply be executed in the context of the currently running thread or ISR; the trap return will then restore the context appropriately. Note that the SLEEPY SLOTH system calls themselves are *not* implemented as traps but as simple functions that can even be inlined by the compiler.

VIII. RELATED WORK

The approach to having basic run-to-completion tasks run as ISRs and to mapping task activations to IRQ source triggering was new when we first published SLOTH [5]. SLEEPY SLOTH extends this work by providing this functionality to more complex blocking tasks. We are not aware of any work on a comparable embedded kernel that runs on commodity hardware; here, we list work on kernels with scheduling enhanced by *customized* hardware and work on the thread–ISR boundary.

Control flow scheduling and dispatching is the core responsibility of any operating system kernel and thus important to be efficient. That is why ways to move software task scheduling into hardware have been researched, although all of the approaches—like Atalanta [23], cs2 [15], HW-RTOS [2], FASTCHART [11], and Silicon TRON [16]—use *customized* hardware synthesized on an FPGA or a similar component to achieve this. This way, those approaches are able to implement arbitrary control flow semantics, including blocking threads. The Responsive Multithreaded Processor (RMT) [25], [24] is customized to integrate real-time functions into the processing unit, including very fast task switching by providing eight hardware contexts plus 32 contexts in an on-chip context cache [17]. SLEEPY SLOTH, on the other hand, is designed to run on *commodity off-the-shelf hardware* with any modern interrupt controller to achieve its boost in performance. Note that some architectures provide hardware support for fast context switching; however, that support only targets fast *dispatching* but not *scheduling* of tasks as the SLEEPY SLOTH approach does.

Overcoming the strict distinction between OS-operated threads and hardware-operated interrupt handlers has also been the work of Kleiman et al. [8]. In their work on the Solaris kernel for desktops and servers, they investigated ways to enhance interrupt service routines to become threads under the control of the OS kernel—in order for them to be able to block. Lohmann et al. [14] showed that this concept is also feasible for embedded system kernels and made this kind of interrupt synchronization a configurable property of their CiAO system. Before those two systems, early microkernels and microkernel-like systems (including AX [22], L3 [9], and L4 [10]) started implementing interrupt handlers as user threads, having very small stubs send the corresponding threads an IPC message once an interrupt is triggered; this way, regular threads and ISR threads also run in a single priority space. However, during the execution of the interrupt handler stubs, those systems still exhibit rate-monotonic priority inversion by potentially delaying high-priority threads. Intel's early real-time operating systems, such as iRMX [20] and iDCX 51 [7], also included concepts of special interrupt tasks, which are scheduled and dispatched as high-priority OS tasks in the OS priority space. However, all of the discussed kernels merely enhance their interrupt handlers to threads (which implies the *overhead* for software-managed threads), whereas SLEEPY SLOTH goes the other way and makes threads run as ISRs, resulting in a significant performance gain.

IX. CONCLUSIONS

We have presented our SLEEPY SLOTH operating system design, which exploits standard interrupt hardware to implement an efficient universal thread abstraction that can be triggered by hardware and software events. SLEEPY SLOTH control flows are scheduled and dispatched with low latency by the interrupt controller; additionally, they can be blocked and resumed in their execution like threads. They share a single priority space, facilitating arbitrary interactions between hardware- and software-induced control flows and avoiding the real-time problem of rate-monotonic priority inversion.

As SLEEPY SLOTH uses the hardware interrupt system instead of software-implemented routines for scheduling and dispatching, the resulting performance boost is convincing. We have evaluated our approach by implementing the conformance classes BCC1 and ECC1 of the OSEK OS standard, which is omnipresent in the automotive industry. With respect to event latencies, SLEEPY SLOTH outperforms a leading commercial implementation of this standard by a factor of 1.3 to 19.

The SLEEPY SLOTH approach abolishes the artificial distinction between threads and ISRs: *threads can be interrupt handlers* and *interrupt handlers can be threads*. Developers of event-driven systems can forget about the differences, choosing priorities and semantics freely based solely on the requirements of the application.

X. Future Work

Having designed and implemented a universal thread abstraction that is scheduled and dispatched by the interrupt controller, we aim at accommodating future multi-core platforms as well. Since Sleepy Sloth is already running on an ARM Cortex-M3 microcontroller, we want to investigate ways how to run it on the open source Pandaboard [19], which features a dual-core ARM Cortex-A9 MPCore. Both the M3 and the A9 have the same interrupt controller, ARM's nested vectored interrupt controller (NVIC), which fulfills the requirements as stated in Section III-B. Since Sleepy Sloth is based on the OSEK OS standard [18], we will evaluate its successor standard, AUTOSAR OS, and in particular its multi-core specification [1] to be able to present a sophisticated multi-core Sloth design.

XI. Acknowledgments

We thank the RTSS 2011 reviewers for their valuable feedback and our shepherd Gernot Heiser for helping us improve this paper.

References

[1] AUTOSAR. Specification of multi-core OS architecture (version 1.0.0). Technical report, Automotive Open System Architecture GbR, November 2009.

[2] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*, pages 324–329, New York, NY, USA, 2006. ACM Press.

[3] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[4] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '06)*, pages 385–394, Washington, DC, USA, 2006. IEEE Computer Society Press.

[5] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)*, pages 204–213. IEEE Computer Society Press, December 2009.

[6] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *AP32009, TC17x6/TC17x7 – Safe Cancellation of Service Requests*, July 2008.

[7] Intel Corporation, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA. *iDCX 51 Distributed Control Executive User's Guide for Release 2*, April 1987. Available at http://www.alfirin.net/flamer/bitbus/dcx51.zip.

[8] Steve Kleiman and Joe Eykholt. Interrupts as threads. *ACM SIGOPS Operating Systems Review*, 29(2):21–26, April 1995.

[9] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM Press, 1993.

[10] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, ACM SIGOPS Operating Systems Review. ACM Press, December 1995.

[11] Lennart Lindh and Frank Stanischewski. FASTCHART – a fast time deterministic CPU and hardware based real-time-kernel. In *Proceedings of the 1991 Euromicro Workshop on Real-Time Systems*, pages 36–40, Jun 1991.

[12] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[14] Daniel Lohmann, Jochen Streicher, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Interrupt synchronization in the CiAO operating system. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, New York, NY, USA, 2007. ACM Press.

[15] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 869–875, New York, NY, USA, 2004. ACM Press.

[16] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *Proceedings of the 12th TRON Project International Symposium (TRON '95)*, pages 34–42, Nov 1995.

[17] Amos R. Omondi and Michael Horne. Performance of a context cache for a multithreaded pipeline. *Journal of Systems Architecture*, 45(4):305–322, 1998.

[18] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17.

[19] Pandaboard homepage. http://pandaboard.org/.

[20] RadiSys Corporation, 5445 NE Dawson Creek Drive, Hillsboro, OR 97124, USA. *Introducing the iRMX Operating Systems*, December 1999.

[21] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '09)*, pages 59–67, New York, NY, USA, 2009. ACM Press.

[22] Wolfgang Schröder. *A Family of UNIX-like Operating Systems — Use of Processes and the Message-Passing Concept in Structured Operating-System Design*. Dissertation, Technical University of Berlin, 1986. In German.

[23] Di-Shi Sun, Douglas M. Blough, and Vincent John Mooney III. Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications. Technical report, Georgia Institute of Technology, 2002.

[24] Nobuyuki Yamasaki. Responsive multithreaded processor for distributed real-time systems. *Journal of Robotics and Mechatronics*, 17(2), 2005.

[25] Nobuyuki Yamasaki, Ikuo Magaki, and Tsutomu Itou. Prioritized SMT architecture with IPC control method for real-time processing. In *Proceedings of the 13th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '07)*, pages 12–21, April 2007.

SLOTH ON TIME: Efficient Hardware-Based Scheduling for Time-Triggered RTOS*

Wanja Hofer, Daniel Danner, Rainer Müller,
Fabian Scheler, Wolfgang Schröder-Preikschat, Daniel Lohmann
Friedrich–Alexander University Erlangen–Nuremberg, Germany
E-Mail: {hofer,danner,raimue,scheler,wosch,lohmann}@cs.fau.de

***Abstract*—Traditional time-triggered operating systems are implemented by multiplexing a single hardware timer—the system timer—in software, having the kernel maintain dispatcher tables at run time. Our SLOTH ON TIME approach proposes to make use of multiple timer cells as available on modern microcontroller platforms to encapsulate dispatcher tables in the timer configuration, yielding low scheduling and dispatching latencies at run time. SLOTH ON TIME instruments available timer cells in different roles to implement time-triggered task activation, deadline monitoring, and time synchronization, amongst others.**

By comparing the SLOTH ON TIME kernel implementation to two commercial kernels, we show that our concept significantly reduces the overhead of time-triggered operating systems. The speed-ups in task dispatching that it achieves range up to a factor of 171 x, and its dispatch latencies go as low as 14 clock cycles. Additionally, we demonstrate that SLOTH ON TIME minimizes jitter and increases schedulability for its real-time applications, and that it avoids situations of priority inversion where traditional kernels fail by design.

I. INTRODUCTION AND MOTIVATION

In operating system engineering, the overhead induced by the kernel is a crucial property since operating system kernels do not provide a business value of their own. This is especially true in *embedded real-time systems*, where superfluous bytes in RAM and ROM as well as unnecessary event latencies can decide whether a kernel is used for the implementation of an embedded device or not. In previous work on the SLOTH approach, we have shown that by using commodity microcontroller hardware in a more sophisticated manner in the kernel, we can achieve lower footprints in RAM and ROM as well as very low system call overheads [7]. To achieve this, the SLOTH kernel maps run-to-completion tasks to interrupt handlers and lets the interrupt hardware schedule them, eliminating the need for a software task scheduler completely. Additionally, we have been able to show that implementing a full thread abstraction with blocking functionality in the SLEEPY SLOTH kernel still yields a significant performance boost over traditional, software-based embedded kernels [8].

However, both the SLOTH and the SLEEPY SLOTH kernels target *event-triggered* real-time systems with event-driven task dispatching. In this paper, we discuss how the SLOTH principle of making better use of hardware facilities in the implementation of embedded kernels can be applied to *time-triggered operating systems*. The resulting SLOTH ON TIME kernel uses the fundamental task dispatching mechanisms as introduced in the SLOTH kernel and makes better use of the central hardware part of any time-triggered kernel: the hardware timer facility.

This paper provides the following contributions:

- We develop a comprehensive design for mapping time-triggered tasks and kernel timing services to hardware timer cells and present SLOTH ON TIME, the first real-time kernel that utilizes *arrays* of hardware timers (see Section IV).
- By evaluating our SLOTH ON TIME kernel and comparing it to traditional time-triggered kernel implementations, we show that our design minimizes the number of interrupts and the kernel-induced overhead at run time, yielding lower latencies, reduced jitter, increased schedulability, and better energy efficiency for real-time applications (see Section V).
- We discuss the implications of the SLOTH ON TIME approach on the design of time-triggered kernels and time-triggered applications (see Section VI).

Before explaining the SLOTH ON TIME design and system, we first describe the time-triggered system model that we use in this paper in Section II and provide necessary background information on the original SLOTH and SLEEPY SLOTH kernels and the microcontroller hardware model that we use as a basis for our description in Section III.

II. SYSTEM MODEL FOR TIME-TRIGGERED TASK ACTIVATION

In this section, we describe the system model that we use for time-triggered task scheduling throughout this paper. Since our model is motivated by the kernel point of view on time-triggered tasks, we take the terminology and semantics from publicly available embedded operating system specifications. Those include the specification for the time-triggered OSEKtime kernel [13] and the AUTOSAR OS specification, which targets an event-triggered kernel that features a schedule table abstraction and timing protection facilities [2]. Both of those standards were developed by leading automotive manufacturers and suppliers based on their experiences with requirements on real-time kernels; this is why they are widely used in the automotive industry. Additionally, since the standards are also implemented by commercially available kernels, we are able to directly compare kernel properties between our SLOTH ON TIME operating system and those kernels by writing benchmarking applications against the respective OS interfaces.

*This work was partly supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-1, SCHR 603/9-1, and SFB/TR 89 (project C1).

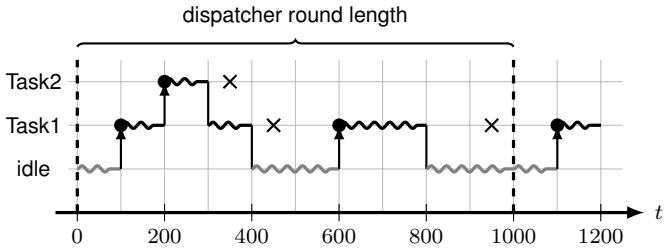


Fig. 1: The model for time-triggered activation and deadlines used in this paper, based on the OSEKtime specification [13]. In this example of a dispatcher table, task activations are depicted by circles, their deadlines by crosses. Later task activations preempt currently running tasks, yielding a stack-based execution pattern.

A. Time-Triggered OSEKtime

The central component of OSEKtime is a preplanned dispatcher table of fixed round length that cyclically activates tasks at given offsets (see example in Figure 1). An activated task always preempts the currently running task until its termination, resulting in a stack-based scheduling policy. Additional OSEKtime features include synchronization with a global time source and task deadline monitoring for detecting misbehaving tasks. The latter is done by checking whether a task is neither running nor preempted at given points in time within a dispatcher round (see crosses in Figure 1).

B. AUTOSAR OS Schedule Tables and Execution Budgets

In contrast to OSEKtime, AUTOSAR OS approaches time-triggered activations in a more complex way and offers seamless integration with the event-triggered model specified by the same system. Although the basic structure of statically defined dispatcher tables (called *schedule tables*) is the same as in OSEKtime, a time-triggered activation in AUTOSAR OS will not inevitably result in preempting any running task, but instead the activated task will be scheduled according to its own static priority and the priorities of other active tasks. In an AUTOSAR system, a distinction between time-triggered and event-triggered tasks does not exist, and both types of activations share the same priority space. Further extensions consist in the possibility to have multiple schedule tables run simultaneously and to have non-cyclic tables, which are executed only once.

AUTOSAR OS also specifies the ability to restrict the execution budget available to individual tasks. Although this is not related to time-triggered scheduling in particular, we will show that the mechanisms developed in SLOTH ON TIME are suitable for an efficient implementation of this feature as well.

III. BACKGROUND

In order to understand the concept and design principles behind SLOTH ON TIME, we first provide background information about the original event-triggered SLOTH kernel that SLOTH ON TIME is based on, and we describe the requirements on the underlying microcontroller hardware, introducing the abstract terminology used throughout the rest of the paper.

A. SLOTH *Revisited*

The main idea in the original event-triggered SLOTH kernel [7] is to have application tasks run as interrupt handlers internally in the system. Each task is statically mapped to a dedicated interrupt source of the underlying hardware platform at compile time; the IRQ source’s priority is configured to the task priority and the corresponding interrupt handler is set to the user-provided task function. SLOTH task system calls are implemented by modifying the state of the hardware IRQ controller: The activation of a task, for instance, is implemented by setting the pending bit of the corresponding IRQ source. This way, the interrupt controller automatically schedules and dispatches SLOTH tasks depending on the current system priority. By these means, SLOTH is able to achieve low overheads in its system calls, both in terms of execution latency and in terms of binary code size and lines of source code.

The original SLOTH kernel can only schedule and dispatch basic run-to-completion tasks as mandated by the OSEK BCC1 conformance class, in which the execution of control flows is strictly nested—which means that a task preempted by higher-priority tasks can only resume after those have run to completion. Thus, all tasks are executed on the same stack—the interrupt stack—which is used both for the execution of the current task and for storing the contexts of preempted tasks. The enhanced SLEEPY SLOTH kernel [8] additionally handles extended tasks that can block in their execution and resume at a later point in time (specified by OSEK’s ECC1 conformance class). SLEEPY SLOTH implements these extended tasks by providing a full task context including a stack of its own for each of them; additionally, a short task prologue is executed at the beginning of each task’s interrupt handler every time that a task is being dispatched. The prologue is responsible for switching to the corresponding task stack and then either initializes or restores the task context depending on whether the task was unblocked or whether it is being run for the first time.

B. Microcontroller Hardware Model and Requirements

With SLOTH ON TIME, we describe a hardware-centric and efficient way to design time-triggered services in a real-time kernel. Our concept makes use of timer arrays with multiple timer cells, which are available on many modern microcontroller platforms such as the Freescale MPC55xx and MPC56xx embedded PowerPC families (64 timer cells) or the Infineon TriCore TC1796 (256 timer cells)—the reference platform for SLOTH ON TIME. As shown in the following section on its design, SLOTH ON TIME requires one timer cell per task activation and deadline in a dispatcher round in its base implementation; the SLOTH approach is to leverage *available* hardware features (in this case, an abundance of timer cells) to improve non-functional properties of the kernel. For platforms with fewer timer cells than activation and deadline points, however, we describe a slightly less efficient alternative design that uses partial multiplexing in Section IV-F.

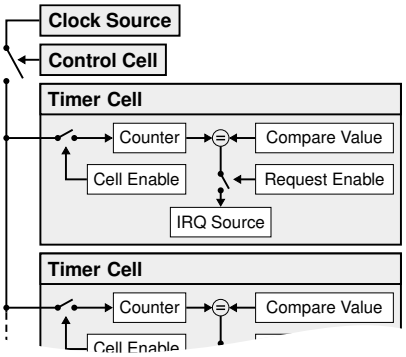


Fig. 2: The abstract model for available timer components on modern microcontroller platforms, introducing the terminology used in this paper.

If the hardware offers *hierarchical* timer cells for controlling lower-order cells, SLOTH ON TIME can optionally make use of that feature, too. Additionally, as in the original SLOTH kernel, we require the hardware platform to offer as many IRQ sources and IRQ priorities as there are tasks in the system; the platforms mentioned before offer plenty of those connected to their timer arrays.

In the rest of the paper, we use the following terminology for the timer array in use (see Figure 2). We call an individual timer of the array a *timer cell*, which features a *counter* register; it is driven by a connected clock signal, which increments or decrements the counter depending on the cell configuration. If the counter value matches the value of the *compare* register or has run down in decrement mode, it triggers the pending bit in the attached *IRQ source*, but only if the *request enable* bit is set. The whole cell can be deactivated by clearing the *cell enable* bit, which stops the internal counter.

IV. SLOTH ON TIME

The main design idea in SLOTH ON TIME is to map the application timing requirements to hardware timer arrays with multiple timer cells—pursuing efficient execution at run time. SLOTH ON TIME tailors those timer cells for different purposes within a time-triggered operating system, introducing different *roles* of timer cells (see overview in Figure 3)—including task activation cells, table control cells, deadline monitoring cells, execution budget cells, and time synchronization cells, as described in this section. Additionally, we show how time-triggered and event-triggered systems can be integrated in SLOTH ON TIME, and we highlight the design of multiplexed timer cells for hardware platforms with fewer available timer cells.

A. *Time-Triggered* SLOTH

Traditional time-triggered kernels implement task activations by instrumenting a single hardware timer, which then serves as the system timer. The system timer is programmed and reprogrammed by the kernel at run time whenever a scheduling decision has to be performed; the next system timer expiry point is usually looked up in a static table representing the application schedule.

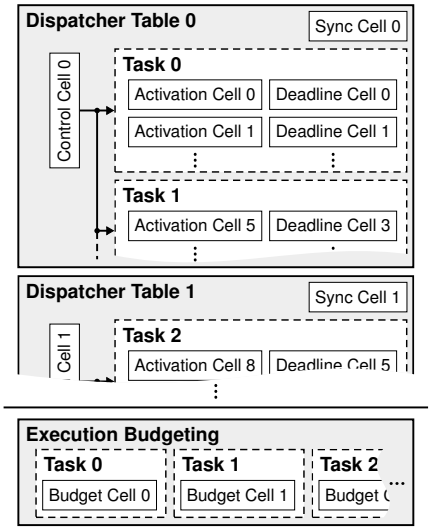


Fig. 3: The different roles that SLOTH ON TIME uses available hardware timer cells for. Some roles are only used in OSEKtime-like systems, others are only used in AUTOSAR-OS-like systems.

SLOTH ON TIME tries to avoid dynamic decisions—and, therefore, run time overhead—as far as possible by instrumenting multiple timer cells. This way, programming the timers can mostly be limited to the initialization phase; during the system's productive execution phase, the overhead for time-triggered task execution is kept to a minimum.

1) Static Design: SLOTH ON TIME not only comprises the actual time-triggered kernel, but also consists of a static analysis and generation component (see Figure 4). As its input, the analysis tool takes the task activation schedule as provided by the application programmer (see Artifact A in Figure 4) and the platform description of the timer hardware (Artifact B), and, in an intermediate step, outputs a mapping of the included expiry points to individual *activation cells* (Artifact C), which are subject to platform-specific mapping constraints due to the way the individual timer cells are interconnected[1]. The timer cells for SLOTH ON TIME to use are taken from a pool of cells marked as available by the application; this information is also provided by the application configuration. If the number of available timer cells is not sufficient, SLOTH ON TIME uses partial multiplexing, which we describe in Section IV-F.

In the next step, the calculated mapping is used to generate initialization code for the involved timer cells (Artifact D). The compare values for all cells are set to the length of the dispatcher round so that the cell generates an interrupt and resets the counter to zero after one full round has been completed. The initial counter value of a cell is set to the round length minus the expiry point offset in the dispatcher round. This way, once the cell is enabled, it generates its first interrupt after its offset is reached, and then each time a full dispatcher round has elapsed.

Additionally, code for starting the dispatcher is generated

[1]The mapping algorithm is work in progress; currently, the timer cells still have to be assigned manually to respect platform restrictions.

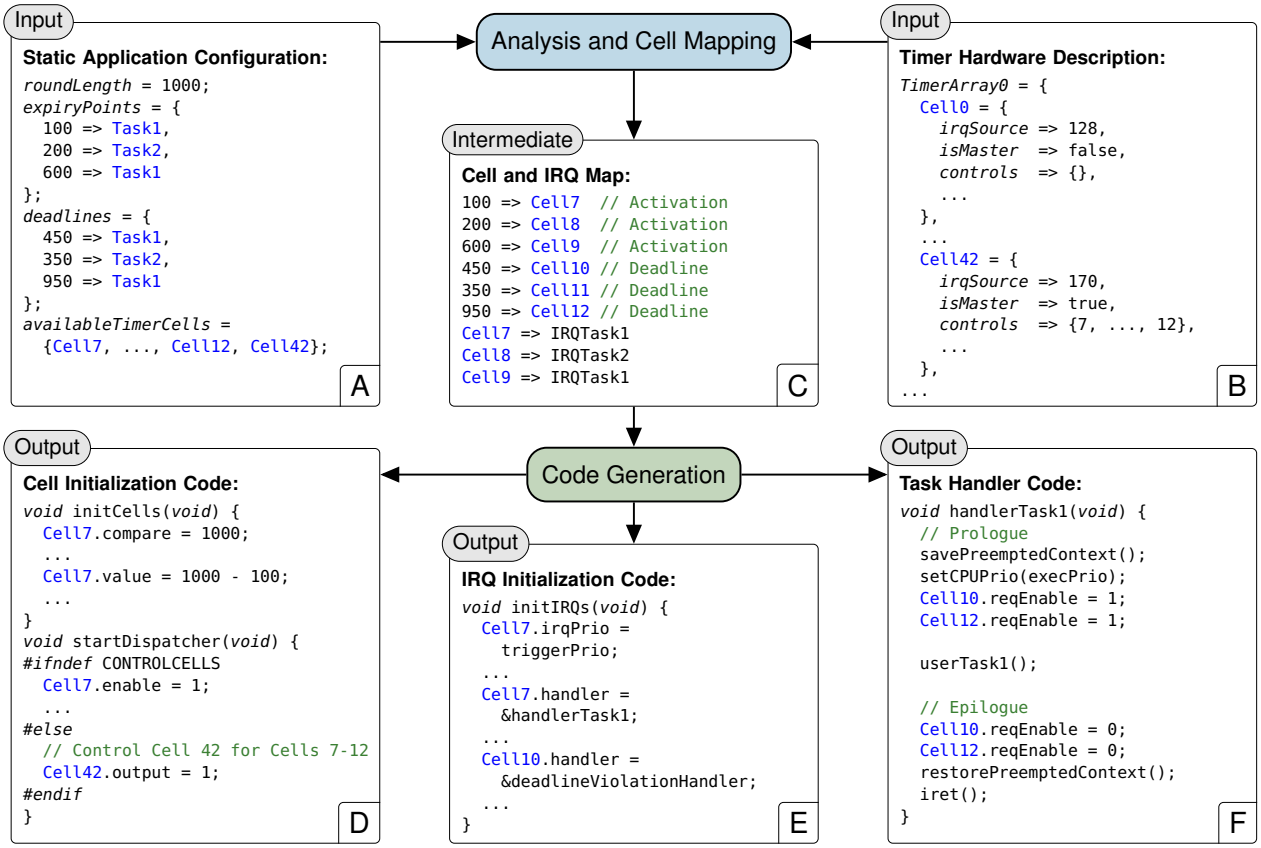


Fig. 4: Static analysis and generation in SLOTH ON TIME, producing the mapping of expiry points and deadlines to timer cells and IRQ sources, the corresponding timer and interrupt initialization code, and task handler code with prologues and epilogues. The example values correspond to the sample application schedule depicted in Figure 1.

that enables all involved activation cells consecutively (Artifact D). If the underlying hardware platform features hierarchically connected cascading timer cells, then a higher-order cell is used as a so-called *control cell* to switch all connected lower-order timer cells on or off simultaneously. If such a control cell is available, enabling it will enable all connected activation cells of a dispatcher round (see also the timer model in Figure 2). This mechanism enables completely accurate and atomic starting and stopping of dispatcher rounds by setting a single bit in the respective control cell (see also evaluation in Section V-B3).

Furthermore, since tasks are bound to interrupt handlers for automatic execution by the hardware, the interrupt system needs to be configured appropriately (Artifact E). This entails setting the IRQ priorities of the involved cells to the system trigger priority (see explanation in Section IV-A2) and registering the corresponding interrupt handler for the cell’s task.

2) Run Time Behavior: At run time, no expiry points and dispatcher tables need to be managed by SLOTH ON TIME, since all required information is encapsulated in the timer cells that are preconfigured during the system initialization. Once the dispatcher is started by enabling the control cell or the individual timer cells, the interrupts corresponding to task dispatches will automatically be triggered at the specified expiry points by the timer system. The hardware interrupt system will then interrupt the current execution, which will either be the system’s idle loop or a task about to be preempted, and dispatch the associated interrupt handler, which in SLOTH ON TIME basically corresponds to the task function as specified by the user, surrounded by a small wrapper.

The only functions that are not performed automatically by the hardware in SLOTH ON TIME are saving the full preempted task context when a new time-triggered task is dispatched and lowering the CPU priority from the interrupt *trigger priority* to the *execution priority* (see generated code in Artifact F in Figure 4). This lowering is needed to achieve the stack-based preemption behavior of tasks in the system, such as mandated by the OSEKtime specification [13], for instance (see also Figure 1). By configuring all interrupts to be triggered at a high trigger priority and lowering interrupt handler execution to a lower execution priority, every task can be preempted by any task that is triggered at a later point in time, yielding the desired stack-based behavior. Thus, a task activation with a later expiry point implicitly has a higher priority than any preceding task activation.

B. Deadline Monitoring

Deadlines to be monitored for violation are implemented in SLOTH ON TIME much in the same way that task activation expiry points are (see Figure 4). Every deadline specified in the application configuration is assigned to a *deadline*

cell (see also Figure 3), which is a timer cell configured to be triggered after the deadline offset, and then after one dispatcher round has elapsed (Artifacts C and D in Figure 4). The interrupt handler that is registered for such deadline cells is an exception handler for deadline violations that calls an application-provided handler to take action (Artifact E).

In contrast to traditional implementations, SLOTH ON TIME disables the interrupt requests for a deadline cell once the corresponding task has run to completion, and re-enables them once the task has started to run. This way, deadlines that are *not* violated do not lead to unnecessary IRQs, which would disturb the execution of other real-time tasks in the system (see also evaluation in Section V-B). In the system, this behavior is implemented by enhancing the generated prologues and epilogues of the time-triggered tasks; here, the request enable bits of the associated deadline cells are enabled and disabled, respectively (see generated Artifact F in Figure 4).

C. Combination of Time-Triggered and Event-Triggered Systems

In the OSEK and AUTOSAR automotive standards that we use as a basis for our investigations, there are two approaches to combine event-triggered elements with a time-triggered system as implemented by SLOTH ON TIME. The OSEK specifications describe the possibility of a mixed-mode system running an event-triggered OSEK system during the idle time of the time-triggered OSEKtime system running on top, whereas the AUTOSAR OS standard specifies a system with event-triggered tasks that can optionally be activated at certain points in time using the schedule table abstraction. In this section, we show how we implement both approaches in the SLOTH ON TIME system.

1) Mixed-Mode System: Since the original SLOTH kernel implements the event-triggered OSEK OS interface [14], whereas SLOTH ON TIME implements the time-triggered OSEKtime interface, we combine both systems by separating their priority spaces by means of configuration. By assigning all event-triggered tasks priorities that are lower than the time-triggered execution and trigger priorities, the event-triggered system is only executed when there are no time-triggered tasks running; it can be preempted by a time-triggered interrupt at any time. Additionally, we make sure that the event-triggered SLOTH kernel synchronization priority, which is used to synchronize access to kernel state against asynchronous task activations, is set to the highest priority of all event-triggered tasks but lower than the time-triggered priorities. Thus, the integration of both kinds of systems can easily be achieved without jeopardizing the timely execution of the time-triggered tasks.

2) Event-Triggered System with Time-Triggered Elements: In contrast to the mixed-mode approach, AUTOSAR OS defines an event-triggered operating system with static task priorities; its schedule table abstraction only provides means to *activate* the event-triggered tasks at certain points in time, which does not necessarily lead to *dispatching* them as in purely time-triggered systems (in case a higher-priority task is currently running). AUTOSAR tasks have application-configured and potentially distinct priorities; at run time, they can raise their execution priority by acquiring resources for synchronization or even block while waiting for an event.

The schedule table abstraction therefore does not strictly follow the time-triggered paradigm, but it is implemented in SLOTH ON TIME in a way that is very similar to the time-triggered dispatcher table. Instead of configuring the priorities of the IRQ sources attached to the timer system to the system trigger priority, however (see Artifact E in Figure 4), they are set to the priority of the task they activate. This way, the time-dependent activation of tasks is seamlessly integrated into the execution of the rest of the SLOTH system, since after the IRQ pending bit has been set, it does not matter whether this was due to a timer expiry or by synchronously issuing a task activation system call.

To fully implement AUTOSAR OS schedule tables, the SLOTH ON TIME timer facility is enhanced in three ways. First, AUTOSAR allows multiple schedule tables to be executed simultaneously and starting and stopping them at any time. Thus, SLOTH ON TIME introduces the corresponding system calls to enable and disable the control cell for the corresponding schedule table (see also Section IV-A1). Second, AUTOSAR defines non-repeating schedule tables, which encapsulate expiry points for a single dispatcher round, executed only once when that schedule table is started. SLOTH ON TIME implements this kind of schedule table by preconfiguring the corresponding timer cells to one-shot mode; this way, they do not need to be manually deactivated at the end of the schedule table. Third, schedule tables can be started with a global time offset specified dynamically at run time. In that case, SLOTH ON TIME reconfigures the statically configured timer cells for that schedule table to include the run time parameter in its offset calculation before starting it by enabling its control cell.

D. Execution Time Protection

In contrast to deadline monitoring, which is used in time-triggered systems like OSEKtime (see Section IV-B), AUTOSAR prescribes timing protection facilities using execution time budgeting for fault isolation. Each task is assigned a maximum execution budget per activation, which is decremented while that task is running, yielding an exception when the budget is exhausted. In its design, SLOTH ON TIME employs the same mechanisms used for expiry points and deadlines to implement those task budgets. It assigns one *budget cell* to each task to be monitored (see also Figure 3), initializes its counter with the execution time budget provided by the application configuration, and configures it to run down once started. The associated IRQ source is configured to execute a user-defined protection hook as an exception handler in case the budget cell timer should expire.

Furthermore, the dispatch points in the system are instrumented to pause, resume, and reset the budget timers appropriately. First, this entails enhancing the task prologues, which pause the budget timer of the preempted task and start the budget timer of the task that is about to run. Second, the task epilogues executed after task termination are added

instructions to reset the budget to the initial value configured for this task, as suggested by the AUTOSAR specification, and to resume the budget timer of the previously preempted task about to be restored.

This design allows for light-weight monitoring of task execution budgets at run time without the need to maintain and calculate using software counters; this information is implicitly encapsulated and automatically updated by the timer hardware in the counter registers.

E. Synchronization with a Global Time Base

Real-time systems are rarely deployed stand-alone but often act together as a distributed system. Therefore, in time-triggered systems, synchronization of the system nodes with a global time base needs to be maintained. This feature has to be supported by the time-triggered kernel by adjusting the execution of dispatcher rounds depending on the detected clock drift.

If support for synchronization is enabled, SLOTH ON TIME allocates a dedicated *sync cell* (see also Figure 3) and configures its offset to the point after the last deadline in a dispatcher round (e.g., 950 in the example schedule in Figure 1). This point has to be specified in the configuration by the application programmer and can be used to apply a limited amount of negative drift depending on the remaining length of the dispatcher round (50 in the example). Positive drifts are, of course, not restricted in this way.

The interrupt handler attached to the sync cell then checks at run time whether a drift occurred. If so, it simply modifies the counter values of all activation, deadline, and sync cells that belong to the dispatcher table, corrected by the drift value (see Figure 5). Since the cell counters are modified *sequentially*, the last counter is changed later than the first counter of a table. However, since the read–modify–write cycle of the counter registers always takes the same amount of time and the modification value is the same for all counters, in effect it does not matter when exactly the counter registers are modified. In case the synchronization handler is not able to reprogram all affected cell counters by the next task activation point in the schedule, it resumes execution at the next cell to be reprogrammed once it becomes active again in the next round.

F. Timer Cell Multiplexing

If the hardware platform does not have enough timer cells to allocate one cell per time-triggered event, SLOTH ON TIME also allows to partially fall back to multiplexing—allocating only one timer cell for each role (activation and deadline) *per task* and partly reconfiguring it at run time.

For multiplexed deadline monitoring, if the current deadline has not been violated, the task epilogue reconfigures the expiration point of the deadline cell to the *next* deadline instead of disabling it. The deltas between the deadline points are retrieved from a small offset array held in ROM.

For multiplexed activations of the same task, another offset array contains the deltas between the activation points of that task. The enhanced task prologue then reconfigures the compare value of the activation cell to the next activation point every time that task is dispatched.

V. EVALUATION

In order to assess the effects of our proposed timer-centric architecture on the non-functional properties of a time-triggered kernel, we have implemented SLOTH ON TIME with all the kernel features described in Section IV on the Infineon TriCore TC1796 microcontroller, which is widely used for control units in the automotive domain. Since from a *functional* point of view, SLOTH ON TIME implements the OSEKtime and AUTOSAR OS standards, we can directly compare our kernel to a commercial OSEKtime system and a commercial AUTOSAR OS system, both of which are available for the TC1796. This way, we can take benchmarking applications written against the respective OS interface and run them unaltered on both SLOTH ON TIME and the commercial kernels.

A. The Infineon TriCore TC1796 Microcontroller

The TC1796, which serves as a reference platform for SLOTH ON TIME, is a 32-bit microcontroller that features a RISC load/store architecture, a Harvard memory model, and 256 interrupt priority levels. The TC1796's timer system is called its general-purpose timer array and includes 256 timer cells, which can be configured to form a cascading hierarchy if needed. The cells can be routed to 92 different interrupt sources, whose requests can be configured in their priorities and the interrupt handlers that they trigger. The timer and interrupt configuration is performed by manipulating memory-mapped registers.

For the evaluation, we clocked the chip at 50 MHz (corresponding to a cycle time of 20 ns); however, we state the results of our latency and performance measurements in numbers of clock cycles to be frequency-independent. We performed all measurements from internal no-wait-state RAM (both for code and data), so caching effects did not apply. The actual measurements were carried out using a hardware trace unit by Lauterbach. All of the quantitative evaluation results were obtained by measuring the number of cycles spent between two given instructions (e.g., from the first instruction of the interrupt handler to the first user code instruction of the associated task) repeatedly for at least 1,000 times and then averaging these samples. In some situations, the distribution of the samples exhibits two distinct peaks of similar height, which are located exactly 4 cycles apart and presumably related to unstableness in measuring conditional jumps. Aside from this effect, the deviations from the average have shown to be negligible in all measurements of SLOTH ON TIME.

B. Qualitative Evaluation

While running our test applications on SLOTH ON TIME and both commercial kernels, we could observe several effects of our design on the *qualitative* execution properties by examining the execution traces from the hardware tracing unit.

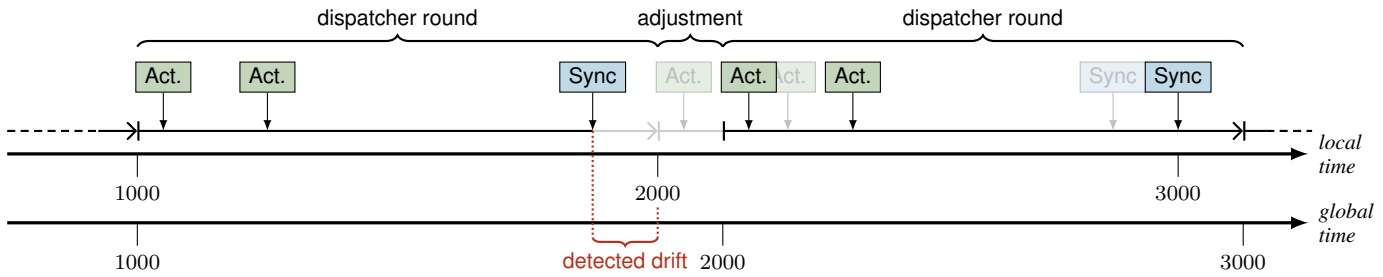
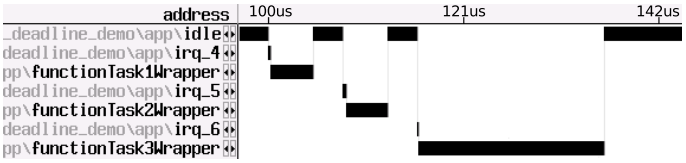
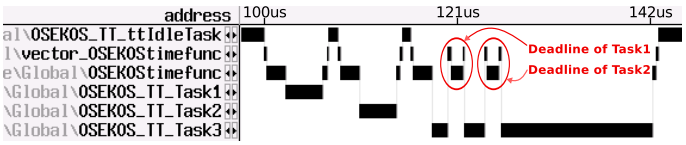


Fig. 5: Synchronization to a global time is implemented in SLOTH ON TIME by adjusting the current counter values of the cells involved in a dispatcher round one after the other, requiring only one read–modify–write cycle per cell.

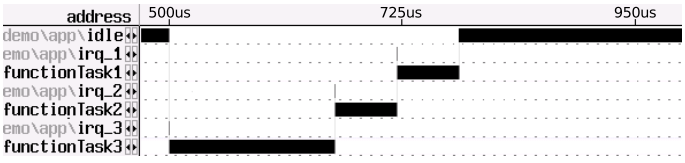


(a) SLOTH ON TIME

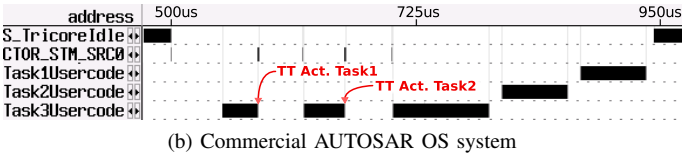


(b) Commercial OSEKtime system

Fig. 6: Comparison of execution traces of an OSEKtime application with two deadlines in **(a)** SLOTH ON TIME and **(b)** a commercial OSEKtime system. Non-violated deadlines of `Task1` and `Task2` interrupt the execution of `Task3` in the commercial system, but not in SLOTH ON TIME.



(a) SLOTH ON TIME



(b) Commercial AUTOSAR OS system

Fig. 7: Execution trace revealing rate-monotonic priority inversion in a commercial AUTOSAR OS system occurring on time-triggered activation of lower-priority tasks `Task1` and `Task2`. The trace of the same dispatcher table in SLOTH ON TIME shows no interruption of `Task3`.

1) Avoiding Unnecessary IRQs: For one, both commercial kernels exhibit unnecessary interrupts with unnecessary interrupt handlers executing, possibly interrupting and disturbing application control flows. All of those interrupts are not needed by the application semantics, and the SLOTH ON TIME design can avoid all of them in its running implementation.

Figure 6 shows the execution traces for an application with three tasks and three deadlines per dispatcher round, running on SLOTH ON TIME and the commercial OSEKtime implementation. The commercial OSEKtime issues an interrupt for every deadline to be monitored, since it then checks if the corresponding task is still running (see interruptions of `Task3` in Figure 6b). These interrupts take 95 clock cycles each, possibly interrupting application tasks for the violation check; this number multiplies by the number of deadlines stated in the application configuration to yield the overhead per dispatcher round. Those unnecessary IRQs in the commercial OSEKtime kernel are especially problematic since they also occur for *non-violated* deadlines—which are, after all, the normal case and not the exception.

SLOTH ON TIME can avoid those unnecessary IRQs completely (see continuous execution of `Task3` in Figure 6a). It uses dedicated deadline timer cells, which are turned off using a single memory-mapped store instruction when the task has run to completion in its task epilogue (see Section IV-B); this takes 10 clock cycles per deadline (see also Section V-C1). SLOTH ON TIME effectively trades the overhead introduced by interrupt handlers in the schedule as in traditional systems for overhead introduced when a task terminates; this trade-off decision has the advantage that it does not interrupt the execution of other tasks, facilitating real-time analyses on the schedule. Note that the commercial kernel *could* be implemented in a way similar to SLOTH ON TIME to avoid additional interrupts; however, the overhead for its software logic would probably exceed the overhead that SLOTH ON TIME introduces (compare 10 cycles per deadline in SLOTH ON TIME to 95 cycles per deadline check interrupt in the commercial kernel).

2) Avoiding Priority Inversion: Second, in the commercial AUTOSAR OS system, we could observe a certain kind of priority inversion, which occurs when a low-priority task is activated by the timer while a high-priority task is running (see gaps in the execution of `Task3` in Figure 7b). The high-priority task is interrupted by the timer interrupt, whose handler then checks which task is to be activated, and inserts this low-priority task into the ready queue. Thus, code is executed on behalf of a low-priority task while a high-priority task is running or ready to run; Leyva del Foyo et al. coined the term *rate-monotonic priority inversion* for that phenomenon [5]. The priority inversion interruptions exhibited by the commercial AUTOSAR OS kernel are really heavy-weight: The corresponding handlers execute for 2,075 clock cycles each. This can lead to serious

deviations between the statically calculated WCET for a task and its actual run time, which is potentially prolonged by several of those low-priority interrupts.

In SLOTH ON TIME, those interrupts do not occur at all since the corresponding timer cell activates the task not by executing code on the main CPU but by setting the pending bit of the associated IRQ source, configured with the task priority. Since the high-priority task that is currently running runs at high *interrupt* priority, the activation does not lead to an interruption until the high-priority task blocks or terminates (see continuous execution of `Task3` in Figure 7a).

Thus, the SLOTH approach of running tasks as interrupt service routines in combination with the SLOTH ON TIME concept of using dedicated timer cells for time-dependent activations minimizes the number of interrupts in the system, facilitating real-time analyses. Interrupts only occur if an action needs to be taken by the kernel on behalf of the application. Traditional kernels with a single system timer cannot avoid the described kind of priority inversion since they *have to* put the activated task into the ready queue (even if it has a low priority) and reconfigure the timer to the next expiry point; this is an inherent design issue that SLOTH ON TIME overcomes. In SLOTH ON TIME, those problems are avoided by design; the timer hardware runs concurrently to the main CPU and activates a task by setting the corresponding IRQ pending bit without having to interrupt the CPU.

Furthermore, in traditional systems, the timer interrupt handler needs to be synchronized with the kernel since it accesses the kernel ready queue; this leads to jitter in the task dispatch times since the timer interrupt handler might additionally be delayed by a task executing a system call. The SLOTH ON TIME approach eliminates that jitter source since explicit kernel synchronization is not necessary—the ready queue is implemented implicitly in the hardware IRQ state and does not need to be synchronized in software.

3) Preciseness: We also investigated the preciseness of task dispatch times as specified by the static schedule and the drift between several consecutive dispatcher rounds. Both in our SLOTH ON TIME kernel and the two commercial kernels, we could not observe any drift since all of them rely on hardware timers and static execution overhead in their timer interrupt handlers.

Additionally, we could show that by using control cells as proposed in Section IV-A1, all activation cells of the dispatcher round or schedule table can be started simultaneously. This way, the additional overhead introduced by starting all timer cells in sequential machine instructions does not need to be respected in the offset calculation for the individual activation cells.

C. Quantitative Evaluation

Since non-functional properties such as kernel execution times, latencies, and memory footprint are crucial to real-time kernels, we also took comprehensive measurements to be able to state the *quantitative* effects of our SLOTH ON TIME design on these important properties.

```
<handlerTask2>:
    mov %d0,2944
    mtcr $psw,%d0  // enable global address registers
    isync  // synchronize previous instruction
    st.a [+%a9]-4,%a8  // save preempted task ID on stack
    mov.a %a8,2  // set new task ID
    st.t <GPTA0_LTCCTR11>,3,1  // enable deadline cell
    bisr 2  // set exec prio 2, save context, enable IRQs
    call userTask2  // enter user code
    disable  // suspend IRQs for synchronization
    st.t <GPTA0_LTCCTR11>,3,0  // disable deadline cell
    rslcx  // restore context
    ld.a %a8,[%a9+]  // restore preempted task ID from stack
    rfe  // return from interrupt handler (re-enables IRQs)
```

Fig. 8: Compiled time-triggered task interrupt handler in SLOTH ON TIME on the TC1796 for `Task2` with one deadline.

1) OSEKtime Evaluation: Since SLOTH ON TIME encapsulates the expiry points of a dispatcher round in its timer cell configuration and traditional implementations need a look-up table to multiplex the system timer at run time, we expect differences in the overhead of time-triggered task dispatch and termination in SLOTH ON TIME and the commercial OSEKtime kernel. The top of Table I shows our measurements, which confirm that our approach yields very low latencies at run time compared to traditional implementations like the commercial OSEKtime, yielding speed-up numbers of 8.6 and 2.7 for task dispatch and termination, respectively. This reduced overhead in SLOTH ON TIME leads to additional slack time in a dispatcher round, which can be used to include additional application functionality (compare the idle times in Figure 6).

Note that the number of 14 clock cycles for the time-triggered task dispatch includes *all* costs between the preemption of the running task or the idle loop to the first user instruction in the dispatched task (see assembly instructions in Figure 8). Thus, the number reflects the complete SLOTH ON TIME prologue wrapper, which itself entails saving the context of the preempted task on the stack; since the system is strictly stack-based, all tasks run on the same stack, so the stack pointer does not have to be switched. The overhead numbers of 60–74 cycles for a task activation in an event-triggered SLOTH system (presented in [7]) exactly correspond to the 14 cycles for the context save prologue plus the activation system call issued by the calling task. Since SLOTH ON TIME activations are time-triggered, the overhead for the system call is not applicable, yielding the very low total number of 14 cycles. Enabling activation cell multiplexing (see Section IV-F) adds 18 cycles to the activation overhead for any task that benefits from this feature due to multiple activations in a single dispatcher round. Tasks activated only once per dispatcher round are not affected by this and retain the usual overhead.

If deadline monitoring is used in the application, both overhead numbers in SLOTH ON TIME increase by about 10 cycles for every deadline associated with a given task (see bottom of Table I). This stems from the fact that SLOTH ON TIME activates and deactivates the deadline cells for that task, which compiles to a single memory-mapped store instruction per cell (see also Figure 8); due to memory bus synchronization,

	SLOTH ON TIME	OSEKtime	Speed-Up
Time-triggered (TT) dispatch	14	120	8.6
Terminate	14	38	2.7
TT dispatch w/ 1 deadline	26	120	4.6
TT dispatch w/ 2 deadlines	34	120	3.5
Terminate w/ 1 deadline	24	38	1.6
Terminate w/ 2 deadlines	34	38	1.1

TABLE I: Run time overhead of time-triggered task dispatch and termination with deadline monitoring enabled, comparing SLOTH ON TIME with a commercial OSEKtime implementation (in number of clock cycles).

this instruction needs 10 clock cycles. If multiplexing of deadline cells as described in Section IV-F is used instead, no additional overhead is incurred during dispatch, but an increase of 18 cycles is measured for the task termination, representing the cost of maintaining the state of the offset array and reconfiguring the deadline cell. However, this increased overhead does not increase further with additional deadlines to be monitored for the same task; starting with two deadlines per task, multiplexing yields a performance advantage. This advantage is traded for a slight increase in memory footprint for the offset array and the associated index variable.

The dispatch overhead in the commercial OSEKtime system remains the same independent of the number of deadlines; however, as discussed in Section V-B1, it introduces additional IRQs to a dispatcher round, executing for 95 cycles per deadline.

In the mixed-system case, when an event-triggered OSEK system runs in the idle time of the time-triggered OSEKtime system, the commercial implementation exhibits the same latency as in the time-triggered-only case (120 cycles). SLOTH's latency depends on the conformance class of the underlying event-triggered system: If the application only includes basic run-to-completion tasks (class BCC1), then its latencies remain the same as in the time-triggered-only case (14 cycles for task activation and 14 cycles for task termination). If it also includes extended tasks that can block at run time (class ECC1), then the latencies rise the same way as described in SLEEPY SLOTH [8]. Since extended tasks have to run on stacks of their own because they potentially block, an additional stack switch is needed when a time-triggered task preempts an extended task of the underlying event-triggered system. With the raised latency being 28 cycles, even in that case SLOTH is still considerably faster than the commercial kernel (speed-up of 4.3).

2) AUTOSAR OS Evaluation: We also evaluated the latencies of event-triggered systems featuring time-triggered task activation; the measurements use the AUTOSAR interface of both SLOTH ON TIME and the commercial AUTOSAR OS kernel.

First, we show the measurement numbers for the two relevant system calls: `StartScheduleTableRel()`, which starts a schedule table dispatcher round relative to the current time, and `StopScheduleTable()`, which stops the execution of further expiry points of a given table. The results are shown in Table II. Due to the preconfiguration of corresponding timer cells during the initialization, the system call latencies in SLOTH ON TIME compared to the commercial implementation reach speed-up numbers of 10.2 and 37.6 for starting and stopping a schedule table at run time, respectively.

	SLOTH ON TIME	AUTOSAR OS	Speed-Up
StartScheduleTableRel()	108	1,104	10.2
StopScheduleTable()	20	752	37.6

TABLE II: Overhead of time-triggered system services in event-triggered AUTOSAR OS systems, comparing SLOTH ON TIME with a commercial implementation of the AUTOSAR standard (in number of clock cycles).

Second, we measured the latencies for activating a task at certain points in time as specified by an AUTOSAR schedule table; the results are shown in Table III. In SLOTH ON TIME, activating a task using preconfigured timer cells and user task functions connected to an interrupt source totals to between 14 and 77 clock cycles, depending on the types of the involved tasks. If full stack switches are needed (since extended tasks are able to block), the latency is higher compared to when only basic tasks are involved, which run to completion and can therefore run on the same stack. Again, those numbers reflect the execution of the whole wrapper prologue, measuring the time from the timer interrupt to the first user task instruction; thus, they include the whole context switch, including the stack switch if an extended task is involved. The commercial AUTOSAR system needs 2,344 to 2,400 clock cycles to achieve this, resulting in speed-up numbers for SLOTH ON TIME between 31.2 and 171.4. Dispatches resulting from task termination amount to 14 to 88 cycles, again depending on the involved task types. The commercial AUTOSAR implementation takes 382 to 532 clock cycles for those test cases, yielding speed-up numbers of 6.0 to 38.0 for SLOTH ON TIME.

3) Memory Footprint: Since SLOTH ON TIME is highly configurable, its memory footprint depends on factors such as the selected set of features and the configured number of tasks, activation points, deadlines, et cetera. As a ballpark figure for the memory usage of SLOTH ON TIME, we created a minimal time-triggered application with one task and one deadline but no actual user code and measured a total of 8 bytes of RAM usage and 1,480 bytes of ROM (of which 624 bytes are claimed by the platform start-up code). In comparison, equivalent setups in the commercial implementations allocate 52 bytes of RAM and 2,600 bytes of ROM (OSEKtime) and 900 bytes of RAM and 34,514 bytes of ROM (AUTOSAR OS).

D. Overall Benefit for the Application

By using SLOTH ON TIME instead of one of the commercial kernels, a time-triggered application will be executed more predictably, since unnecessary interrupts and priority inversion can be avoided in SLOTH ON TIME. The gained slack time per dispatcher round depends on the degree that the application makes use of the operating system—on its number of activation points, deadline monitoring points, and executed system calls.

		SLOTH ON TIME	AUTOSAR OS	Speed-Up
Time-triggered task activation with dispatch	idle loop to basic task	14	2,344	167.4
Time-triggered task activation with dispatch	basic task to basic task	14	2,400	171.4
Time-triggered task activation with dispatch	extended task to extended task	77	2,400	31.2
Task termination with dispatch	basic task to idle loop	14	382	27.3
Task termination with dispatch	basic task to basic task	14	532	38.0
Task termination with dispatch	extended task to extended task	88	532	6.0

TABLE III: Latencies of time-triggered task activation and dispatching in event-triggered AUTOSAR OS systems, comparing SLOTH ON TIME with a commercial AUTOSAR OS implementation (in number of clock cycles).

However, since SLOTH ON TIME has a lower overhead in *all* microbenchmarks, the application will *always* experience a benefit.

For OSEKtime systems, the number of additionally available clock cycles per dispatcher round is 130 per task activation point without a deadline, plus 114 per task activation with one or more associated deadlines, plus 95 per monitored deadline. For AUTOSAR OS systems, the total benefit in clock cycles is at least 2,767 per task activation, plus 996 per schedule table start system call, plus 732 per schedule table stop system call, plus the SLOTH and SLEEPY SLOTH benefit for the regular event-triggered operation (see [7] and [8]). The gained slack time allows the application to include additional functionality.

VI. DISCUSSION

In this section, we discuss the general applicability of our SLOTH ON TIME approach and the impact it has on applications running on top of the kernel.

A. Applicability

The applicability of the proposed SLOTH ON TIME design depends on the timer architecture of the underlying hardware platform. Due to the instrumentation of timer cells in different role types, an appropriate number of timer cells that are not otherwise used by the application needs to be available for the kernel to use, specified in the configuration (see Artifact A in Figure 4). Many modern microcontrollers, especially those that are used in control systems, offer plenty of configurable timers—like the Freescale MPC55xx and MPC56xx embedded PowerPC families and the Infineon TriCore TC1796, the reference platform for SLOTH ON TIME.

Since timer cells and connected interrupt sources are usually not freely configurable, the mapping of scheduling points to timer cells can be challenging for the developer of the hardware model (see Artifact B in Figure 4). On the TC1796, for instance, restrictions apply that make it necessary to use two adjacent cells per activation; additionally, four cells are connected to a single interrupt source. Thus, on that platform, a second activation of the same task in a dispatcher round can be accommodated with minimal additional hardware resources. More than two activations will be subject to a trade-off decision, probably favoring a multiplexing implementation if cells become scarce (see Section IV-F).

In theory, SLOTH ON TIME competes with the application for the timer cells, which may limit their availability for the kernel. In practice, however, timer arrays are only used for control algorithms that bear latency and activation rate requirements so tight that traditional RTOS cannot fulfill them; by using the timer hardware directly, the application also becomes less portable. SLOTH ON TIME, on the other hand, offers very low latencies, but hides its implementation beneath a platform-independent OSEKtime API and configuration, shielding the developer from porting the application from one hardware timer API to another. We are convinced that, given an RTOS that offers hardware-comparable latencies for task activations such as SLOTH ON TIME, application developers would happily migrate from using timer arrays directly to using time-triggered task abstractions.

By using platform-specific timer hardware extensively, the SLOTH ON TIME kernel itself is less portable than a traditional time-triggered kernel with software multiplexing. Our reference implementation runs on the Infineon TriCore TC1796; from our experiences in porting the event-triggered SLOTH and SLEEPY SLOTH kernels, however, we can state that the additional porting effort can be contained by using a clear internal abstraction boundary.

Since multi-core processors are used mainly for consolidation purposes in the automotive market, the AUTOSAR standard recently introduced hard task partitioning for multi-core applications. Schedule tables, which encapsulate task activations, are therefore also bound to specific cores; thus, the SLOTH ON TIME approach can be applied to each schedule table separately by statically initializing the task interrupt sources to route interrupt requests to the configured core.

B. Impact on Schedulability, Predictability, and Efficiency

The benefits of improved latency and system call performance introduced by the SLOTH ON TIME concept have a positive impact on the schedulability of tasks in the application. As directly perceivable by comparing the idle times in the execution traces in SLOTH ON TIME and the commercial kernels (see Figures 6 and 7), the increased slack time can be used to include additional application functionality by either extending existing tasks or by introducing additional time-triggered tasks. In application scenarios with highly loaded schedules, an implementation using traditional kernels might not even be possible, whereas the reduced overhead in SLOTH ON TIME might make it feasible.

Schedules with activation points that are very close together in time will cause problems in traditional kernels, since the software scheduler will delay the second activation through its overhead for the first task activation. By activating and

dispatching in hardware, the minimal overhead caused by SLOTH ON TIME can accommodate close activation points—as they occur when scheduling tasks with high activation frequencies, for instance. Taking into account the dispatching overheads caused by the kernels, SLOTH ON TIME supports a maximum dispatch frequency of 1.7 MHz of a single minimal task, whereas the commercial AUTOSAR kernel only supports 17 kHz, for example.

The fact that SLOTH ON TIME has very few data structures in the software part of the kernel not only reduces its footprint in RAM, but also in the platform's data cache. This reduced kernel-induced cache load increases application performance by letting it execute out of the cache more often, and, more importantly, reduces caching effects *caused* by the kernel—thereby increasing the predictability of the application. This facilitates the development of the real-time schedule with tightened WCETs.

Additionally, the reduced kernel-induced load in SLOTH ON TIME systems positively influences the energy consumption of embedded devices. Since most of those systems spend the majority of their time in sleep mode, the lower overhead introduced by the operating system has a significant impact on energy efficiency—and, therefore, battery life, which is crucial in *mobile* embedded systems.

VII. RELATED WORK

The idea of hardware-based and hardware-assisted real-time scheduling is not new. Existing approaches, like Atalanta [17], cs2 [11], HW-RTOS [3], FASTCHART [9], and Silicon TRON [12], but also the work presented in [16], [4], however, focus on *event-triggered* real-time systems and employ *customized* hardware synthesized on an FPGA or a similar component. SLOTH ON TIME, in contrast, employs *commodity* hardware to implement scheduling with focus on *time-triggered* systems; the seamless integration of mixed-mode systems is possible by employing the techniques presented in our previous papers [7], [8].

The fact that only little work so far has focused on hardware assistance for time-triggered schedulers might be rooted in the generally simple and straight-forward software implementations of such schedulers [10]. On the other hand, aiming for efficient software-based timer abstractions has a long tradition in the operating systems community, including concepts such as timer wheels [18], soft timers [1], and adaptive timers [15]. These concepts, however, are all based on the assumptions that 1) hardware timers are sparse and that 2) they are costly to reprogram [6]. The first assumption is no longer valid with current 32-bit microcontroller platforms; the second still is, but SLOTH ON TIME can avoid the reprogramming costs by using dedicated timer cells for each task and deadline.

VIII. CONCLUSION

We have presented our SLOTH ON TIME RTOS design, which exploits standard timer array hardware available on modern microcontrollers to efficiently offload schedules in a time-triggered or mixed-mode real-time system to the hardware. With our design, tasks are scheduled and dispatched with low latency by the platform's timer and interrupt controller. SLOTH ON TIME instruments the available timer cells not only for task activation, but also for schedule table control, deadline monitoring, and time synchronization; thereby it entirely prevents issues of rate-monotonic priority inversion.

The resulting performance boost is convincing: We have evaluated our approach by implementing the OSEKtime OS standard and the AUTOSAR OS schedule table facility, both of which are omnipresent in the automotive industry. With a dispatch latency of 14 cycles, SLOTH ON TIME outperforms leading commercial implementations of these standards by a factor of up to 171 x. Our results show that it is time (sic!) to exploit the capabilities of modern microcontrollers in time-triggered real-time kernels.

REFERENCES

[1] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM TOCS*, 18(3):197–228, 2000.

[2] AUTOSAR. Specification of operating system (version 4.0.0). Technical report, Automotive Open System Architecture GbR, 2009. http://autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf.

[3] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In *GLSVLSI '06*, pages 324–329, 2006.

[4] Uwe Dannowski and Hermann Härtig. Policing offloaded. In *RTAS '00*, pages 218–228, 2000.

[5] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *RTAS '06*, pages 14–23, 2006.

[6] Antônio Augusto Fröhlich, Giovani Gracioli, and João Felipe Santos. Periodic timers revisited: The real-time embedded system perspective. *Computers & Electrical Engineering*, 37(3):365–375, 2011.

[7] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *RTSS '09*, pages 204–213, 2009.

[8] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Sleepy Sloth: Threads as interrupts as threads. In *RTSS '11*, pages 67–77, 2011.

[9] Lennart Lindh and Frank Stanischewski. FASTCHART – A fast time deterministic CPU and hardware based real-time-kernel. In *Euromicro Workshop on Real-Time Systems*, pages 36–40, 1991.

[10] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[11] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In *SAC '04*, pages 869–875, 2004.

[12] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *12th TRON Project International Symposium (TRON '95)*, pages 34–42, 1995.

[13] OSEK/VDX Group. Time triggered operating system specification 1.0. Technical report, OSEK/VDX Group, 2001. http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf.

[14] 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.

[15] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 seconds is not enough! A study of operating system timer usage. In *EuroSys '08*, pages 205–218, 2008.

[16] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *LCTES '05*, pages 50–58, 2005.

[17] Di-Shi Sun, Douglas M. Blough, and Vincent John Mooney III. Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications. Technical report, Georgia Institute of Technology, 2002.

[18] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *SOSP '87*, pages 25–38, 1987.

Feature Consistency in Compile-Time–Configurable System Software *

Facing the Linux 10,000 Feature Problem

Reinhard Tartler, Daniel Lohmann, Julio Sincero, Wolfgang Schröder-Preikschat
Friedrich–Alexander University Erlangen–Nuremberg
{tartler, lohmann, sincero, wosch}@cs.fau.de

Abstract

Much system software can be configured at compile time to tailor it with respect to a broad range of supported hardware architectures and application domains. A good example is the Linux kernel, which provides more than 10,000 configurable features, growing rapidly.

From the maintenance point of view, compile-time configurability imposes big challenges. The configuration model (the selectable features and their constraints as presented to the user) and the configurability that is actually implemented in the code have to be kept in sync, which, if performed manually, is a tedious and error-prone task. In the case of Linux, this has led to numerous defects in the source code, many of which are actual bugs.

We suggest an approach to automatically check for configurability-related implementation defects in large-scale configurable system software. The configurability is extracted from its various implementation sources and examined for inconsistencies, which manifest in seemingly conditional code that is in fact unconditional. We evaluate our approach with the latest version of Linux, for which our tool detects 1,776 configurability defects, which manifest as dead/superfluous source code and bugs. Our findings have led to numerous source-code improvements and bug fixes in Linux: 123 patches (49 merged) fix 364 defects, 147 of which have been confirmed by the corresponding Linux developers and 20 as fixing a new bug.

* This work was partly supported by the German Research Foundation (DFG) under grant no. SCHR 603/7-1 and SFB/TR 89.

EuroSys '11, April 10–13, 2011, Salzburg, Austria.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.2.9 [*Management*]: Software configuration management

General Terms Algorithms, Design, Experimentation, Management, Languages

Keywords Configurability, Maintenance, Linux, Static Analysis, VAMOS

1. Introduction

> *I know of no feature that is always needed. When we say that two functions are almost always used together, we should remember that "almost" is a euphemism for "not".* DAVID L. PARNAS [1979]

Serving no user value on its own, system software has always been "caught between a rock and a hard place". As a link between hardware and applications, system software is faced with the requirement for variability to meet the specific demands of both. This is particularly true for operating systems, which ideally should be tailorable for domains ranging from small, resource-constrained embedded systems over network appliances and interactive workstations up to mainframe servers. As a result, many operating systems are provided as a software family [Parnas 1979]; they can (and have to) be configured at compile time to derive a concrete operating-system variant.

Configurability as a system property includes two separated – but related – aspects: *implementation* and *configuration*. Kernel developers implement configurability in the code; in most cases they do this by means of conditional compilation and the C preprocessor [Spinellis 2008], despite all the disadvantages with respect to understandability and maintainability ("`#ifdef` hell") this approach is known for [Liebig 2010, Spencer 1992]. Users configure the operating system to derive a concrete variant that fits their purposes. In simple cases they have to do this by (un-)commenting `#define` directives in some global `configure.h` file; however, many operating systems today come with an interactive configuration tool. Based on an internal model of features and constraints,

this tool guides the user through the configuration process by a hierarchical / topic-oriented view on the available features. In fact, it performs implicit consistency checks with respect to the selected features, so that the outcome is always a valid configuration that represents a viable variant. In today's operating systems, this extra guidance is crucial because of the sheer enormity of available features: eCos, for instance, provides more than 700 features, which are configured with (and checked by) eCosConfig [Massa 2002]; the Linux kernel is configured with KConfig and provides more than 10,000 (!) features. This is a lot of variability – and, as we show in this paper, the source of many bugs that could easily be avoided by better tool support.

Our Contributions

This article builds upon previous work. In [Sincero 2010], we have introduced the extraction of a source-code variability model from C Preprocessor (CPP)-based software, which represents a building block for this work. A short summary of this approach is presented in Section 3.2.3. In this paper, we extend that work by incorporating other sources of variability and automatically (cross-) checking them for configurability-related implementation defects in large-scale configurable system software. We evaluate our approach with the latest version of Linux. In summary, we claim the following contributions:

1. It is the first work that shows the problem with the increasing configurability in system software that causes serious maintenance issues. (Section 2.2)
2. It is the first work that checks for configurability-related implementation defects under the consideration of both symbolic *and* logic integrity. (Section 3.1)
3. It presents an algorithm to effectively slice very large configuration models, which are commonly found in system software. This greatly assists our crosschecking approach. (Section 3.2.2)
4. It presents a practical and scalable tool chain that has detected 1,776 configurability-related defects and bugs in Linux 2.6.35; for 121 of these defects (among them 22 confirmed new bugs) our fixes have already been merged into the mainline tree. (Section 4)

In the following, we first analyze the problem in further detail before presenting our approach in Section 3. We evaluate and discuss our approach in Section 4 and Section 5, respectively, and discuss related work in Section 6. The problem of configurability-related defects will be introduced in the context of Linux, which will also be the case study used throughout this paper. Our findings and suggestions, however, also apply to other compile-time configurable system software.

2. Problem Analysis

Linux today provides more than 10,000 configurable features, which is a lot of variability with respect to hardware platforms and application domains. The possibility to leave out functionality that is not needed (such as x86 PAE support in an Atom-based embedded system) and to choose between alternatives for those features that are needed (such as the default IO scheduler to use) is an important factor for its ongoing success in so many different application and hardware domains.

2.1 Configurability in Linux

The enormous configurability of the Linux kernel demands dedicated means to ensure the validity of the resulting Linux variants. Most features are not self-contained; instead, their possible inclusion is constrained by the presence or absence of other features, which in turn impose constraints on further features, and so on. In Linux, variant validity is taken care of by the KConfig tool chain, which is depicted in Figure 1:

❶ Linux employs the KConfig language to specify its configurable features together with their constraints. In version 2.6.35 a total of 761 *Kconfig files* with 110,005 lines of code define 11,283 features plus dependencies. We call the thereby specified variability the Linux **configuration space**.

The following KConfig lines, for instance, describe the (optional) Linux feature to include support for hot CPU plugging in an enterprise server:

```
config HOTPLUG_CPU
  bool "Support for hot-pluggable CPUs"
  depends on SMP && HOTPLUG
   && SYS_SUPPORTS_HOTPLUG_CPU
```

The HOTPLUG_CPU feature *depends on* general support for hot-pluggable hardware and must not be selected in a single-processor system.

❷ The KConfig configuration tool implicitly enforces all feature constraints during the interactive feature selection process. The outcome is, by construction, the description of a valid Linux **configuration variant**.

Technically, the output is a C header file (autoconf.h) and a Makefile (auto.make) that define a CONFIG_<*FEATURE*> preprocessor macro and make variable for every selected KConfig feature:

```
#define CONFIG_HOTPLUG_CPU 1
#define CONFIG_SMP 1
```

It's a convention that *all* and *only* KConfig flags are prefixed with CONFIG_.

❸ Features are implemented in the Linux source base. Whereas some coarse-grained features are enforced by including or excluding whole compilation units in the build process, the majority of features are enforced within the source files by means of the conditional compilation with

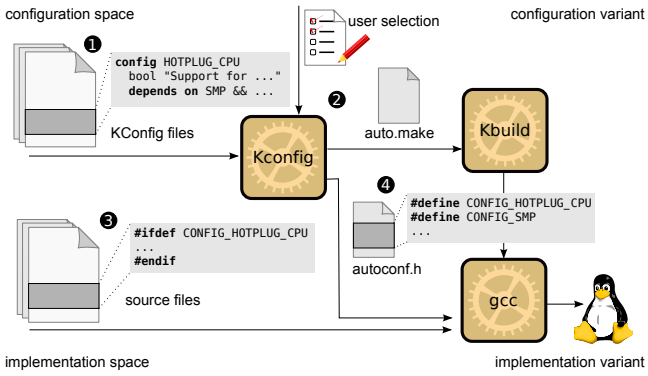


Figure 1. Linux build process (simplified).

the C preprocessor. A total of 27,166 source files contain 82,116 `#ifdef` blocks. We call the thereby implemented variability the Linux **implementation space**.

❹ The KBUILD utility drives the actual variant compilation and linking process by evaluating `auto.make` and embedding the configuration variant definition `autoconf.h` into every compilation unit via GCC's "forced include"[1] mechanism. The result of this process is a concrete Linux **implementation variant**.

2.2 The Issue

Overall, the configurability of Linux is defined by two separated, but related models: The configuration space defines the *intended* variability, whereas the implementation space defines the *implemented* variability of Linux. Given the size of both spaces – 110 kloc for the configuration space and 12 mloc for the implementation space in Linux 2.6.35 –, it is not hard to imagine that this is prone to inconsistencies, which manifest as configurability defects, many of which are bugs. We have identified two types of integrity issues, namely *symbolic* and *logic*, which we introduce in the following by examples from Linux:

Consider the following change, which corrects a simple feature misnaming (detected by our tool and confirmed as a bug) in the file `kernel/smp.c`[2]:

```
diff --git a/kernel/smp.c b/kernel/smp.c
--- a/kernel/smp.c
+++ b/kernel/smp.c

 -#ifdef CONFIG_CPU_HOTPLUG
 +#ifdef CONFIG_HOTPLUG_CPU
```

Patch 1. Fix for a symbolic defect

The issue, which was present in Linux 2.6.30, is an example of a **symbolic integrity violation**; the implementation space references a feature that does not exist in the configuration spaces, so the actual implementation of the `HOTPLUG_CPU` feature is incomplete. This bug remained undetected in the kernel code base for more than six months. We cannot claim credit for detecting this particular bug (it had been reported to the respective developer just before we submitted our patch); however, we have found 116 similar defects caused by symbolic integrity violation that have been confirmed as *new*.

A symbolic integrity violation indicates a configuration–implementation space mismatch with respect to a feature *identifier*. However, consistency issues also occur at the level of feature *constraints*. Consider the following fix, which fixes what we call a **logic integrity violation**:

```
diff --git a/arch/x86/include/asm/mmzone_32.h
           b/arch/x86/include/asm/mmzone_32.h
--- a/arch/x86/include/asm/mmzone_32.h
+++ b/arch/x86/include/asm/mmzone_32.h
@@ -61,11 +61,7 @@ extern s8 physnode_map[];

 static inline int pfn_to_nid(unsigned long pfn)
 {
-#ifdef CONFIG_NUMA
   return((int) physnode_map[(pfn)
            / PAGES_PER_ELEMENT]);
-#else
-  return 0;
-#endif
 }

 /*
```

Patch 2. Fix for a logical defect

The patch itself does not look too complicated – the particularities of the issue it fixes stem from the context: In the source, the affected `pfn_to_nid()` function is nested within a larger code block whose presence condition is `#ifdef CONFIG_DISCONTIGMEM`. According to the KCONFIG model, however, the `DISCONTIGMEM` feature *depends on* the `NUMA` feature, which means that it also implies the selection of `NUMA` in any valid configuration. As a consequence, the `#ifdef CONFIG_NUMA` is superfluous; the `#else` branch is dead and both are removed by the patch. The patch has been confirmed as fixing a *new* defect by the respective Linux developers and is currently processed upstream for final acceptance into mainline Linux.

Compared to symbolic integrity violations, logic integrity violations are generally much more difficult to analyze and fix. So far we have fixed 38 logic integrity violations that have been confirmed as new defects.

Note that Patch 2 does not fix a real bug – it only improves the source-code quality of Linux by removing some dead code and superfluous `#ifdef` statements. Some readers might consider this as "less relevant cosmetical improvement"; however, such "cruft" (especially if it contributes to "`#ifdef`

[1] implemented by the `-include` command-line switch

[2] Shown in unified diff format. Lines starting with -/+ are being removed/added

version	features	#ifdef blocks	source files
2.6.12 (2005)	5338	57078	15219
2.6.20	7059	62873	18513
2.6.25	8394	67972	20609
2.6.30	9570	79154	23960
2.6.35 (2010)	11223	84150	28598
relative growth (5 years)	**110%**	**47%**	**88%**

Table 1. Growth of configurability in Linux

hell") causes long-term maintenance costs and impedes the general accessibility of the source.

2.3 Problem Summary

Overall, we find 1,316 symbolic + 460 logic integrity violations in Linux 2.6.35 – numbers that speak for themselves. The situation becomes more severe every day, given how quickly Linux is growing: Within the last five years, the number of configuration-conditional blocks in the source (`#if` blocks that test for some KCONFIG item) has grown by around fifty percent, the number of features (KCONFIG items) and source files have practically doubled (Table 1).

We think that configurability as a system property has to be seen as a significant (and so far underestimated) cause of software defects in its own respect.

3. The Approach

As pointed out in Section 2.2, many configurability-related defects are caused by inconsistencies that result from the fact that configurability is defined by two (technically separated, but conceptually related) models: the configuration space and the implementation space. The general idea of our approach is to extract all configurability-related information from both models into a common representation (a propositional formula), which is then used to cross-check the variability exposed within and across both models in order to find inconsistencies. We call these inconsistencies *configurability defects:*

> A **configurability defect** (short: **defect**) is a configuration-conditional **item** that is either **dead** (never included) or **undead** (always included) under the precondition that its **parent** (enclosing item) is included.

Examples for *items* in Linux are: KCONFIG options, build rules, and (most prominent) `#ifdef` blocks. The `CONFIG_NUMA` example discussed in Section 2.2 (see Figure 2) bears two *defects* in this respect: Block_2 is *undead* and Block_3 is *dead*. Defects can be further classified as:

Confirmed – a defect that has been confirmed as *unintentional* by the corresponding developers. If the defect has an effect on the binary code of at least one Linux implementation variant, we call it a **bug**.

Rule violation – a defect that, even though it breaks a generally accepted development rule, has been confirmed as *intentional* by the corresponding developers.

Patch 1 discussed in Section 2.2 fixes a *bug*, Patch 2 a *confirmed defect*. In the case of Linux a rule violation is usually the use of the `CONFIG_` prefix for preprocessor flags that are not (yet) defined by KCONFIG. We will discuss the source of rule violations more thoroughly in Section 5.1.

3.1 Challenges in Analyzing Configurability Consistency – "What's wrong with GREP?"

Since version 2.6.23, Linux has included the (AWK/GREP-based) script `checkkconfigsymbols.sh`. This script is supposed to be used by maintainers to check for referential integrity between the KCONFIG model and the source code before committing their changes.

However, for Linux 2.6.30 this script reports 760 issues, among them the `CONFIG_CPU_HOTPLUG` issue discussed in Section 2.2, which remained in the kernel for more than six months. Apparently, kernel maintainers do *not* use this script systematically. While we can only speculate why this is the case, we have identified a number of shortcomings:

Accuracy. The output is disturbed by many **false positives**, defect reports that are not valid, but caused by some `CONFIG_` macros being mentioned in a historical comment. We consider this as a constant annoyance that hinders the frequent employment of the script.

Performance. The script can only be applied on the complete source tree. On reasonable modern hardware (Intel quadcore with 2.83 GHz) it takes over 7 minutes until the output begins. We consider this as too long and too inflexible for integration into the daily incremental build process.

Coverage. Despite its verbosity, the script misses many *valid* defects. **False negatives** are caused, on the one hand, by logic integrity issues, like the `CONFIG_NUMA` example from Figure 2, as logic integrity is not covered at all. However, even many referential integrity issues are not detected – the script does not deal well with KCONFIG's tristate options (which are commonly used for loadable kernel modules). We consider this as a constant source of doubt with respect to the script's output.

The lack of accuracy causes a lot of noise in the output. This, and the fact that the script cannot be used during incremental builds, renders the script barely usable. Most of these shortcomings come from the fact that it does not actually parse and analyze the expressed variability, but just employs regular expressions to cross-match `CONFIG_` identifiers. We conclude that the naïve GREP-based approach is (too) limited in this respect and that this problem has not been considered seriously in the past.

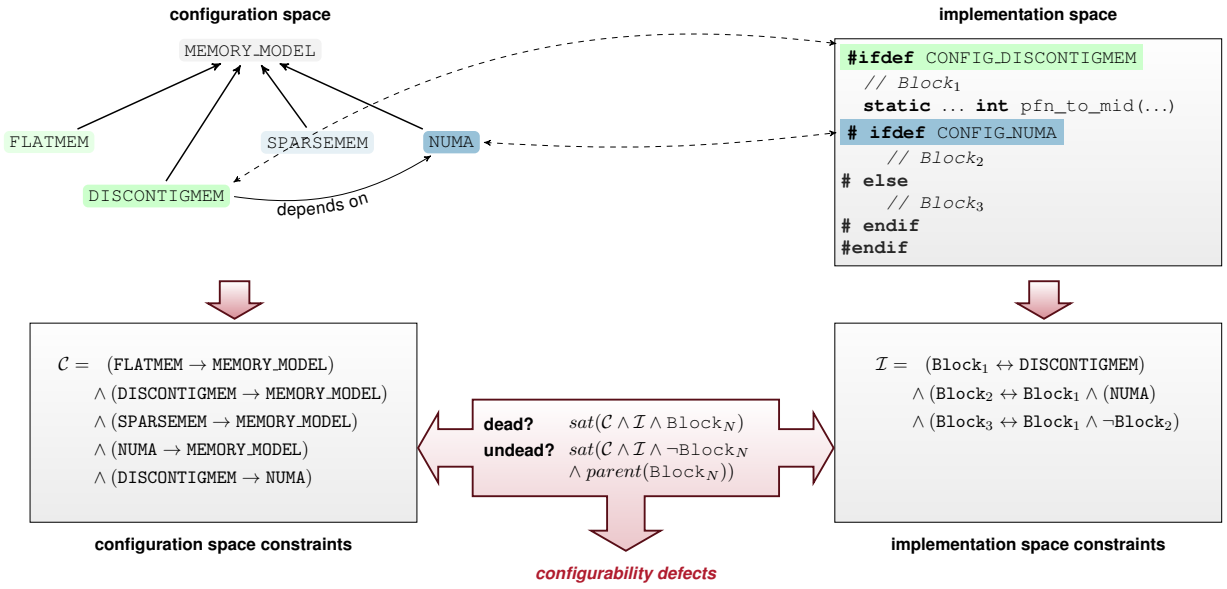


Figure 2. Our approach at a glance: The variability constraints defined by both spaces are extracted separately into propositional formulas, which are then examined against each other to find inconsistencies we call *configurability defects*.

3.2 Our Solution

Essential for the analysis of configurability problems is a common representation of the variability that spreads over different software artifacts. The idea is to individually convert each variability source (e.g., source files, KCONFIG, etc.) to a common representation in form of a sub-model and then combine these sub-models into a model that contains the whole variability of the software project. This makes it possible to analyze each sub-model as well their combination in order to reveal inconsistencies across sub-models.

Most of the constructs that model the variability both in the configuration and implementation spaces can be directly translated to propositional logic; therefore, propositional logic is our abstraction means of choice. As a consequence, the detection of configuration problems boils down to a satisfiability problem.

Linux (and many other systems) keep their configuration space ($\mathcal{C}$) and their implementation space ($\mathcal{I}$) separated. The variability model ($\mathcal{V}$) can be represented by the following boolean formula:

$$\mathcal{V} = \mathcal{C} \wedge \mathcal{I} \tag{1}$$

$\mathcal{V} \mapsto \{0, 1\}$ is a boolean formula over all features of the system; $\mathcal{C}$ and $\mathcal{I}$ are the boolean formulas representing the constraints of the configuration and implementation spaces, respectively. Properly capturing and translating the variability of different artifacts into the formulas $\mathcal{C}$ and $\mathcal{I}$ is crucial for building the complete variability model $\mathcal{V}$. Once the model $\mathcal{V}$ is built we use it to search for defects.

With this model, we validate the implementation for configurability defects, that is, we check if the conditions for the presence of the block ($\texttt{Block}_N$) are fulfillable in the model $\mathcal{V}$. For example, consider Figure 2: The formula shown for dead blocks is satisfiable for Block_1 and Block_2, but not for Block_3. Therefore, Block_3 is considered to be *dead*; similarly the formula for undead blocks indicates that Block_2 is *undead*.

3.2.1 Challenges

In order to implement the solution sketch described above in practice for real-world large-scale system software, we face the following challenges:

Performance. As we aim at dealing with huge code bases, we have to guarantee that our tools finish in a reasonable amount of time. More importantly, we also aim at supporting programmers at development time when only a few files are of interest. Therefore, we consider the efficient check for variability consistency during incremental builds essential.

Flexibility. Projects that handle thousands of features will eventually contain *desired* inconsistencies with respect to their variability. Gradual addition or removal of features and large refactorings are examples of efforts that may lead to such inconsistent states within the lifetime of a project. Also, evolving projects may change their requirements regarding their variability descriptions. Therefore, a tool that checks for configuration problems should be flexible enough to incorporate information about desired issues in order to deliver precise and useful results; it should also minimize the number of false positives and false negatives.

Require: $\mathcal{S}$ initialized with an initial set of items
1: $\mathcal{R} = \mathcal{S}$
2: **while** $\mathcal{S} \neq \emptyset$ **do**
3: $item = \mathcal{S}.pop()$
4: $\mathcal{PC} = presenceCondition(item)$
5: **for all** i such that $i \in \mathcal{PC}$ **do**
6: **if** $i \notin \mathcal{R}$ **then**
7: $\mathcal{S}.push(i)$
8: $\mathcal{R}.push(i)$
9: **end if**
10: **end for**
11: **end while**
12: **return** $\mathcal{R}$

Figure 3. Algorithm for configuration model slicing

In order to achieve both *performance* and *flexibility*, the implementation of our approach needs to take the particularities of the software project into account. Moreover, the precision of the configurability extraction mechanism has direct a impact on the rate of false positive and false negative reports. As many projects have developed their own, custom tools and languages to describe configuration variability, the configurability extraction needs to be tightly tailored.

In the following sections, we describe how we have approached these challenges to achieve good performance, flexibility, and, at the same time, a low number of false positives and false negatives.

3.2.2 Configuration Space

There are several strategies to convert configuration space models into boolean formulas [Benavides 2005, Czarnecki 2007]. However, due to the size of real models – the KCONFIG model contains more than 10,000 features –, the resulting boolean formulas become very complex. The search for a solution to problems that use such formulas may become intractable.

Therefore, we have devised an algorithm that implements *model slicing* for KCONFIG. This allows us to generate submodels from the original model that are smaller than the complete model. To illustrate, suppose we want to check if a specific block of the source code can be enabled by any valid user configuration. This is expressed by the satisfiability of the formula $\mathcal{V} \wedge \texttt{Block}_N$. With a full model, the term $\mathcal{V}$ would contain all user-visible features as logical variables; for the Linux kernel it would have more than 10,000 variables. Nevertheless, not all features influence the solution for this specific problem. The key challenge is to find a sufficient – and preferably minimal – subset of features that can possibly influence the selection of the code block under analysis.

Our slicing algorithm for this purpose is depicted in Figure 3. The goal is to find the set of configuration items that can possibly affect the selection of one or more given initial items. (In our tool, which we will present in Section 4.1, this initial set of items will be taken from the `#ifdef` expressions.)

The basic idea is to check the presence conditions of each item for additional relevant items. Both *direct* and *indirect* dependencies from the initial set of features are thus taken into account such that the resulting set contains all features that can influence the features in the initial set.

In the first step (Line 1) the resulting set $\mathcal{R}$ is initialized with the list of input features. Then, the algorithm iterates until the working stack $\mathcal{S}$ is empty. In each iteration (Lines 2–11), a feature is taken from the stack and its *presence condition* is calculated through the function $presenceCondition(feature)$, which returns a boolean formula of the form $feature \rightarrow \varphi$. This formula represents the condition under which the feature can be enabled; φ is a boolean formula over the available features. Then, all features that appear in φ and have not already been processed (Line 6), are added to the working stack $\mathcal{S}$ and the result set $\mathcal{R}$. This algorithm always terminates; in the worst case, it will return all features and the slice will be exactly like the original model.

To implement our algorithm for Linux, we also have to implement the function $presenceCondition()$ that takes the semantic details of the KCONFIG language into account. In a nutshell, the KCONFIG language supports the definitions of five types of features. Moreover, the features can have a number of attributes like prompts, direct and reverse dependencies, and default values. The presence condition of a feature is the set of conditions that must be met, so that either the user can select it or a default value is set automatically. Consider the following feature defined in the KCONFIG language:

```
config DISCONTIGMEM
        def_bool y
        depends on (!SELECT_MEMORY_MODEL &&
                    ARCH_DISCONTIGMEM_ENABLE) ||
                    DISCONTIGMEM_MANUAL
```

The presence condition for the feature `DISCONTIGMEM` is simply the selection of the feature itself and the expression of the `depends on` option. If a feature has several definitions of prompts and defaults, the feature implies the disjunction of the condition of each option that control its selection. The formal semantics of the KCONFIG language has been studied elsewhere [Berger 2010, Zengler 2010]; such formalisms describe in detail how to correctly derive the presence conditions.

3.2.3 Implementation Space

Many techniques [Baxter 2001, Hu 2000] have been proposed to translate the `CPP` semantics to boolean formulas. However, for our approach, we need to consider the language features of `CPP` that implement conditional compilation only. Therefore we devised an algorithm [Sincero 2010] that is tailored in this respect in order to be precise and have good performance. In short, our algorithm generates a boolean formula that describes a source file by means of its *conditional com-*

pilation structures; it therefore examines the CPP directives `#ifdef, #ifndef, #if, #elif, #else`, which are the constructs responsible for conditional compilation. As result, we receive a formula that describes the presence conditions for each conditional block. It thereby includes all flags (features) that appear in any conditional compilation expression as a logical variable. We build the presence condition $\mathcal{PC}$ of the conditional block b_i as follows:

$$\mathcal{PC}(b_i) = expr(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i) \quad (2)$$

Let b_i be a conditional block. In order for this block to be selected, it is required that its expression $expr(b_i)$ evaluates to true, in `#elif` cascades none of its predecessors are selected $noPredecessors(b_i)$, and for nested blocks, its enclosing `#ifdef` block $parent(b_i)$ is selected. If all these conditions are met, then CPP will necessarily select this block. Additionally, also the reverse is true: if the CPP selects the block, all these presence conditions need to hold. This results in a biimplication: $b_i \leftrightarrow \mathcal{PC}(b_i)$. Therefore, the formula for a file with several blocks can be built as follows:

$$\mathcal{F}_u(\vec{f}, \vec{b}) = \bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i) \quad (3)$$

where $\vec{f}$ is a vector containing all flags present in the file, and $\vec{b}$ is a vector containing a variable for each block of the file. An example is shown on the right hand side of Figure 2: in the upper part we show the source code, and in the lower part we show the generated formula by our algorithm; note that in this example $\mathcal{F}_u(\texttt{[DISCONTIGMEM, NUMA]}, \texttt{[Block}_1\texttt{, Block}_2\texttt{, Block}_3\texttt{]}) = \mathcal{PC}(\texttt{Block}_1) \wedge \mathcal{PC}(\texttt{Block}_2) \wedge \mathcal{PC}(\texttt{Block}_3) = \mathcal{I}$

3.2.4 Crosschecking Among Variability Spaces

Our approach converts the different representations of variability to a common format so that we can check for inconsistencies, the configurability *defects*. Defects appear in two ways, either as **dead**, that is, unselectable blocks, or **undead**, that is, always present blocks. Both kinds of defects indicate code that is only seemingly conditional. They can be found within single models as presented in the previous two sections in isolation as well as across multiple models.

Within a single model we have **implementation-only** defects, which represent code blocks that cannot be selected regardless of the systems' selected features; the structure of the source file itself contains contradictions that impede the selection of a block. This can be determined by checking the satisfiability of the formula $sat(b_i \leftrightarrow \mathcal{PC}(b_i))$. **Configuration-only** defects represents features that are present in the configuration-space model but do not appear in any valid configuration of the model, which means that the presence condition of the feature is not satisfiable. We can check for such defects by solving: $sat(feature \rightarrow presenceCondition(feature))$.

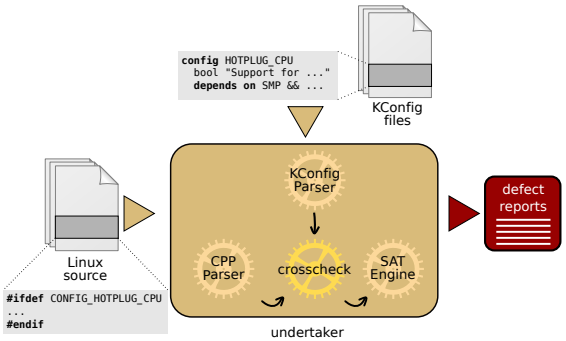


Figure 4. Principle of Operation

Across multiple models we have **configuration-implementation** defects, which occur when the rules from the configuration space contradict rules from the implementation space. We check for such defects by solving $sat((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V})$. **Referential** defects are caused by a *missing feature* (m) that appears in either the configuration or the implementation space *only*. That is, $sat((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V} \wedge \neg(m_1 \vee \cdots \vee m_n))$ is unsatisfiable.

Implementation-only defects have already been addressed in [Sincero 2010]; this paper focuses on the detection of configuration-implementation and referential defects in Linux. The defect analysis can be done using the *dead* and *undead* formulas as shown in the center of Figure 2.

We categorize all identified defects as either *logic* or *symbolic*. Logic defects are those that can only be found by determining the satisfiability of a complex boolean formula. Symbolic defects belong to a sub-group of referential defects where the expression of the analyzed block $exp(b_i)$ is an atomic formula.

4. Evaluation

In order to evaluate our approach, we have developed a prototype tool for Linux and a workflow to submit our results to the kernel developers. We started submitting our first patches in February 2010, at which time Linux version 2.6.33 has just been released. Most of our patches entered the mainline kernel tree during the merge window of version 2.6.36. In the following, we describe our tool and summarize the results.

4.1 Implementation for Linux

We named our tool UNDERTAKER, because its task is to identify (and eventually bury) dead and undead CPP-Blocks. Its basic principle of operation is depicted in Figure 4: The different sources of variability are parsed and transformed into propositional formulas. For CPP parsing, we use the BOOST::WAVE[3] parsing library; for proper parsing of the `Kconfig` files, we have hooked up in the original Linux KCONFIG implementation. The outcome of these parsers is fed into the crosscheck-

[3] `http://www.boost.org`

ing engine as described in Section 3.2.4 and solved using the PICOSAT[4] package. The tool itself is published as Free Software and available on our website.[5]

Our tool scans each `.c` and `.h` file in the source tree individually. Unlike the script `checkkonfigsymbols.sh` as discussed in Section 3.1, this allows developers to focus on the part of the source code they are currently working on and to get instant results for incremental changes. The results come as *defect reports* per file: For each file all configurability-related CPP blocks are analyzed for satisfiability, which yields the defect types described in the previous section. For instance, the report produced for the *configuration-implementation defect* from Figure 2 looks like this:

```
Found Kconfig related DEAD in arch/parisc/include/asm/mmzone.h,
line 40: Block B6 is unselectable, check the SAT formula.
```

Based on this information, the developer now revisits the KCONFIG files. The basis for the report is a formula that is falsified by our SAT solver. For this particular example the following formula was created:

```
1  #B6:arch/parisc/include/asm/mmzone.h:40:1:logic:undead
2  B2 &
3  !B6 &
4  (B0 <-> !_PARISC_MMZONE_H) &
5  (B2 <-> B0 & CONFIG_DISCONTIGMEM) &
6  (B4 <-> B2 & !CONFIG_64BIT) &
7  (B6 <-> B2 & !B4) &
8  (B9 <-> B0 & !B2) &
9  (CONFIG_64BIT -> CONFIG_PA8X00) &
10 (CONFIG_ARCH_DISCONTIGMEM_ENABLE -> CONFIG_64BIT) &
11 (CONFIG_ARCH_SELECT_MEMORY_MODEL -> CONFIG_64BIT) &
12 (CONFIG_CHOICE_11 -> CONFIG_SELECT_MEMORY_MODEL) &
13 (CONFIG_DISCONTIGMEM -> !CONFIG_SELECT_MEMORY_MODEL &
       CONFIG_ARCH_DISCONTIGMEM_ENABLE | CONFIG_DISCONTIGMEM_MANUAL) &
14 (CONFIG_DISCONTIGMEM_MANUAL -> CONFIG_CHOICE_11 &
       CONFIG_ARCH_DISCONTIGMEM_ENABLE) &
15 (CONFIG_PA8X00 -> CONFIG_CHOICE_7) &
16 (CONFIG_SELECT_MEMORY_MODEL -> CONFIG_EXPERIMENTAL |
       CONFIG_ARCH_SELECT_MEMORY_MODEL)
```

This formula can be deciphered easily by examining its parts individually. The first line shows an “executive summary” of the defect; here, Block B6, which starts in Line 40 in the file `arch/parisc/include/asm/mmzone.h`, inhibits a *logical* configuration defect in form of a block that cannot be unselected (“undead”). Lines 4 to 8 show the *presence conditions* of the corresponding blocks (cf. Section 3.2.3 and [Sincero 2010]); they all start with a block variable and by construction cannot cause the formula to be unsatisfiable. From the structure of the formula, we see that Block B2 is the enclosing block. Lines 9ff. contain the extracted implications from KCONFIG (cf. Section 3.2.2). In this case, it turns out that the KCONFIG implications from Line 9 to 16 show a *transitive* dependency from the KCONFIG item `CONFIG_DISCONTIGMEM` (cf. Block B2, Line 5) to the item `CONFIG_64BIT` (cf. Block B4, Line 6). This means that the KCONFIG selection has no impact on the evaluation of the `#ifdef` expression and the code can thus be simplified. We have therefore proposed the following patch to the Linux developers[6]:

```
1  diff --git a/arch/parisc/include/asm/mmzone.h b/arch/parisc/include/
       asm/mmzone.h
2  --- a/arch/parisc/include/asm/mmzone.h
3  +++ b/arch/parisc/include/asm/mmzone.h
4  @@ -35,6 +35,1 @@ extern struct node_map_data node_data[];
5
6  -#ifndef CONFIG_64BIT
7  #define pfn_is_io(pfn) ((pfn & (0xf0000000UL >> PAGE_SHIFT)) == (0
       xf0000000UL >> PAGE_SHIFT))
8  -#else
9  -/* io can be 0xf0f0f0f0f0xxxxxx or 0xffffffff0000000 */
10 -#define pfn_is_io(pfn) ((pfn & (0xf000000000000000UL >> PAGE_SHIFT))
       == (0xf000000000000000UL >> PAGE_SHIFT))
11 -#endif
```

Please note that this is one of the more complicated examples. Most of the defects reports have in fact only a few lines and are much easier to comprehend.

Results. Table 2 (upper half) summarizes the defects that UNDERTAKER finds in Linux 2.6.35, differentiated by subsystem. When counting defects in Linux, some extra care has to be taken with respect to architectures: Linux employs a separate KCONFIG-model per architecture that may also declare architecture-specific features. Hence, we need to run our defect analysis over every architecture and intersect the results. This prevents us from counting, for example, MIPS-specific blocks of the code as *dead* when compiling for x86. An exception of this rule is the code below `arch/`, which is architecture-specific by definition and checked against the configuration model of the respective architecture only.

Most of the 1,776 defects are found in `arch/` and `drivers/`, which together account for more than 75 percent of the configurability-related `#ifdef`-blocks. For these subsystems, we find more than three defects per hundred `#ifdef`-blocks, whereas for all other subsystems this ratio is below one percent (`net/` below two percent). These numbers support the common observation (e.g., [Engler 2001]) that “most bugs can be found in driver code”, which apparently also holds for configurability-related defects. They also indicate that the problems induced by “`#ifdef`-hell” grow more than linearly, which we consider as a serious issue for the increasing configurability of Linux and other system software.

Even though the majority of defects (74%) are caused by symbolic integrity issues, we also find 460 logic integrity violations, which would be a lot harder to detect by “developer brainpower”.

Performance. We have evaluated the performance of our tool with Linux 2.6.35. A full analysis of this kernel processes 27,166 source files with 82,116 configurability-conditional code blocks. This represents the information from the implementation space. The configuration space provides 761 KCONFIG files defining 11,283 features.

A *full* analysis that produces the results as shown in Table 2 takes around 15 minutes on a modern Intel quadcore with 2.83 GHz and 8 GB RAM. However, the implementation

[4] `http://fmv.jku.at/picosat/`

[5] `http://vamos.informatik.uni-erlangen.de/trac/undertaker/`

[6] `http://lkml.org/lkml/2010/5/12/202`

subsystem	#ifdefs	logic	symbolic	total
arch/	33757	345	581	926
drivers/	32695	88	648	736
fs/	3000	4	13	17
include/	7241	6	11	17
kernel/	1412	7	2	9
mm/	555	0	1	1
net/	2731	1	49	50
sound/	3246	5	10	15
virt/	53	0	0	0
other subsystems	601	4	1	5
∑	85291	460	1316	1776
fix proposed		150 (1)	214 (22)	364 (23)
confirmed defect		38 (1)	116 (20)	154 (21)
confirmed rule-violation		88 (0)	21 (2)	109 (2)
pending		24 (0)	77 (0)	101 (0)

Table 2. Upper half: #ifdef blocks and defects per subsystem in Linux version 2.6.35; Lower half: acceptance state of defects (bugs) for which we have submitted a patch

Figure 5. Processing time for 27,166 Linux source files

still leaves a lot of room for optimization: Around 70 percent of the consumed CPU time is *system* time, which is mostly caused by the fact that we fork() the SAT solver for every single #ifdef block.

Despite this optimization potential, the runtime of UNDERTAKER is already appropriate to be integrated into (much more common) incremental Linux builds. Figure 5 depicts the file-based runtime for the Linux source base: Thanks to our slicing algorithm, 94 percent of all source files are analyzed in less than half a second; less than one percent of the source files take more than five seconds and only four files take between 20 and 30 seconds. The upper bound (29.1 seconds) is caused by kernel/sysctl.c, which handles a very high number of features; changes to this file often require a complete rebuild of the kernel anyway. For an incremental build that affects about a dozen files, UNDERTAKER typically finishes in less than six seconds.

4.2 Evaluation of Findings

To evaluate the quality of our findings, we have given our defect reports to two undergraduate students to analyze them, propose a change, and submit the patch upstream to the responsible kernel maintainers. Figure 6 depicts the whole workflow.

The first step is *defect analysis*: The students have to look up the source-code position for which the defect is reported and understand its particularities, which in the case of logical defects (as in the CONFIG_NUMA example presented in Figure 2) might also involve analyzing KCONFIG dependencies and further parts of the source code. This information is then used to develop a *patch* that fixes the defect.

Based on the response to a submitted patch, we *improve and resubmit* and finally classify it (and the defects it fixes) in two categories: *accept (confirmed defect)* and *reject (confirmed rule violation)*. The latter means that the responsible developers consider the defect for some reason as *intended*; we will discuss this further in Section 5.1. As a matter of pragmatics, these defects are added into a local whitelist to filter them out in future runs.

In the period of February to July 2010, the students so far have submitted 123 patches. The submitted patches focus on the arch/ and driver/ subsystems and fix 364 out of 1,776 identified defects (20%). 23 (6%) of the analyzed and fixed defects were classified as bugs. If we extrapolate this defect/bug ratio to the remaining defects, we can expect to find another 80+ configurability-related bugs in the Linux kernel.

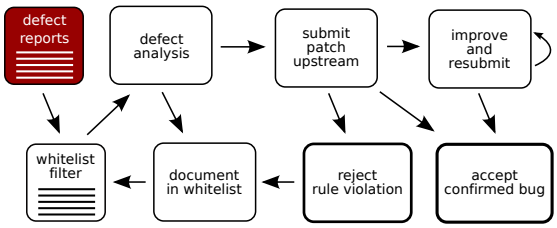


Figure 6. Based on the analysis of the defect reports, a patch is manually created and submitted to kernel developers. Based on the acceptance, we classify the defects that are fixed by our patch either as *confirmed rule violation* or *confirmed defect*.

Reaction of Kernel Maintainers. Table 3 lists the state of the submitted patches in detail; the corresponding defects are listed in Table 2 (lower half). In general, we see that our patches are well received: 87 out of 123 (71%) have been answered; more than 70 percent of them within less than one day, some even within minutes (Figure 7). We take this as indication that many of our patches are easy to verify and in fact appreciated.

Contribution to Linux. Table 3 also classifies the submitted patches as *critical* and *noncritical*, respectively. Critical patches fix *bugs*, that is, configurability defects that have an impact on the binary code. We did not investigate in detail the run-time observable effects of the 23 identified bugs. However, what can be seen from Table 3 is that the responsible developers consider them as worth fixing: 16 out of 17 (94%) of our critical patches have been answered; 9 have already been merged into Linus Torvalds' master git tree for Linux 2.6.36.

The majority of our patches fixes defects that affect the source code only, such as the examples shown in Section 2.2. However, even for these noncritical patches 57 out of 106 (54%) have already reached acknowledged state or better.

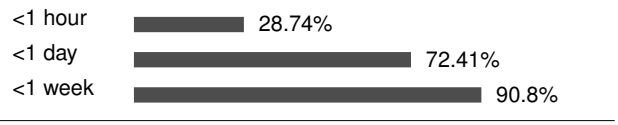


Figure 7. Response time of 87 answered patches

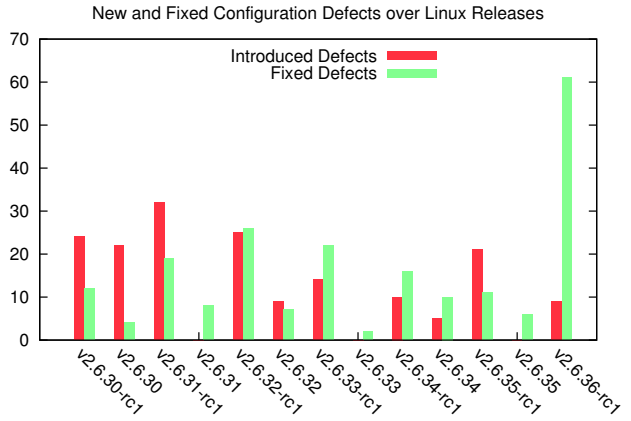


Figure 8. Evolution of defect blocks over various Kernel versions. Most of our work was merged after the release of Linux version 2.6.35.

These patches clean up the kernel sources by removing 5,129 lines of configurability-related dead code and superfluous `#ifdef` statements (“cruft”). We consider this as a strong indicator that the Linux community is aware of the negative effects of configurability on the source-code quality and welcomes attempts to improve the situation.

Figure 8 depicts the impact of our work on a larger scale. To build this figure, we ran our tool on previous kernel versions and calculated the number of configurability defects that were *fixed* and *introduced* with each release. Most of our patches entered the mainline kernel tree during the merge window of version 2.6.36. Given that the patch submissions of two students have already made such a measurable impact, we expect that a consequent application of our approach, ideally directly by developers that work on new or existing code, could significantly reduce the problem of configurability-related consistency issues in Linux.

5. Discussion

Our findings have yielded a notable number of configurability defects in Linux. In the following, we discuss some potential causes for the introduction of defects and rule violations, threats to validity, and the broader applicability of our approach.

5.1 Interpretation of the Feedback

About 57 of the 123 submitted patches were accepted without further comments. We take this as indication that experts can easily verify the correctness of our submissions. Because of the distributed development of the Linux kernel, drawing the line between acknowledged and accepted (i.e., patches that have been merged for the next release), is challenging. We therefore count the 87 patches for which we received comments by Linux maintainers that maintain a public branch on the internet or are otherwise recognized in the Linux community as a confirmation that we identified a valid defect.

patch status	critical	noncritical	$\sum$
submitted	17	106	123
unanswered	1	35	36
ruleviolation	1	14	15
acknowledged	1	14	15
accepted	5	3	8
mainline	9	40	49

Table 3. *Critical* patches do have an effect on the resulting binaries (kernel and runtime-loadable modules). Noncritical patches remove text from the source code only.

Causes for Defects. We have not yet analyzed the causes for defects systematically; doing this (e.g., using HERODOTOS [Palix 2010]) remains a topic for further research. However, we can already name some common causes, for which we need to consider how changes get integrated into Linux:

Logical defects are often caused by *copy and paste* (which confirms a similar observation in [Engler 2001]). Apparently code is often copied *together* with an enclosing `#ifdef`–`#else` block into a new context, where either the `#ifdef` or the #else branch is always taken (i.e., undead) and the counterpart is dead.

The most common source for symbolic defects is *spelling mistakes*, such as the `CONFIG_HOTPLUG` example in Patch 1. Another source for this kind of defects is *incomplete merges* of ongoing developments, such as architecture-specific code that is maintained by respective developer teams who maintain separate development trees and only submit hand-selected *patch series* for inclusion into the mainline. Obviously, this hand selection does not consider configurability-based defects – despite the recommendations in the patch submission guidelines:[7]

```
6: Any new or modified CONFIG options don't muck up the config menu.
7: All new Kconfig options have help text.
8: Has been carefully reviewed with respect to relevant Kconfig
   combinations.  This is very hard to get right with testing --
   brainpower pays off here.
```

Our approach provides a systematic, tool-based approach for this demanded checking of KCONFIG combinations.

Reasons for Rule Violations. On the other hand, we count 15 patches that were rejected by Linux maintainers. For all these patches, the respective maintainers confirmed the defects as valid (in one case even a bug!), but *nevertheless* prefer to keep them in the code. Reasons for this (besides *carelessness* and *responsibility uncertainties*) include:

[7] `Documentation/SubmitChecklist` in the Linux source.

Documentation. Even though all changes to the Linux source code are kept in the version control system (GIT), some maintainers have expressed their preference to keep outdated or unsupported feature implementations in the code in order to serve as a reference or template (e.g., to ease the porting of driver code to a newer hardware platform).

Out-of-tree development. In a number of cases, we find configurability-related items that are referenced from code in private development trees only. Keeping these symbolic defects in the kernel seems to be considered helpful for future code submission and review.

While it is debatable if all of the above are good reasons or not, of course we have to accept the maintainers preferences. The whitelist approach provides a pragmatic way to make such preferences explicit – so that they are no longer reported as defects, but can be addressed later if desired.

5.2 Threats to Validity

Accuracy. A strong feature of our approach is the accuracy with which configurability defects can be found. In our approach, false positives are conditional blocks that are falsely reported as unselectable. This means that there is a KCONFIG selection for which the code is *seen* by the compiler. By design, our approach operates *exact* and avoids this kind of error. Since by construction we avoid false positives (sans implementation bugs), the major threat to validity is the rate of false negatives, that is, the rate of the remaining, unidentified issues.

In fact, we have found for 2 (confirmed) defects explicit `#error` statements in the source that provoke compilation errors in case an *invalid* set of features has been selected. In our experiment, we classified these defects as confirmed rule violations. On top of that, we can find 28 similar `#error` statements in Linux 2.6.35. This indicates some distrust of developers in the variability declarations in KCONFIG, which our tool helps to mitigate by checking the effective constraints accurately.

Coverage. So far we do not consider the (discouraged) 509 `#undef` and `#define CONFIG_` statements that we find in the code. However, these statements could possibly lead to incomplete results for only at most 4.51 percent of the 11,283 KCONFIG items.

Another restriction of the current implementation is that we do not yet analyze nonpropositional expressions in `#ifdef` statements, like comparisons against the integral value of some `CONFIG_` flag. This affects about 2% out of 82,116 `#ifdef` blocks. We are currently looking into improving our implementation to reduce this number even further by rewriting the extracted constraints and process them using a satisfiability modulo theories (SMT) or constraint solving problem (CSP) engine.

An important, yet not considered source of feature constraints is the build system (makefiles). 91 percent of the Linux source files are feature-dependent, that is, they are not compiled at all when the respective feature has not been selected. Incorporation of these additional constraints into our approach is straight-forward: they can simply be added as further conjunctions to the variability model. These additional constraints could possibly restrict the variability even further, and thereby lead to false negatives.

Subtle semantic details and anachronisms of the KCONFIG language and implementation [Berger 2010, Zengler 2010] made our engineering difficult and contributed to the number of false negatives. At the time we conducted the experiment in Section 4, our implementation did not completely support the KCONFIG features *default value* and *select*. Meanwhile, we have fixed these issues in the undertaker, which increases the raw number of defects from 1,776 to 2,972.

In no case our approach resulted in a change that proposes to remove blocks that are used in production. However, in one case[8] we stumbled across old code that is useful with some additional debug-only patches that have never been merged. It turned out that the patches in question are no longer necessary in favor of the new tracing infrastructure. Our patch therefore has contributed to the removal of otherwise useless and potentially confusing code.

Despite all potential sources of false negatives: Compared to the 760 issues reported by the GREP-based approach (including many false positives!, see Section 3.1), our tool already finds more than twice as many defects. As our approach prevents false positives, this has to be considered as a *lower bound* for the number of configurability defects in Linux!

5.3 General Applicability of the Approach

Linux is the most configurable piece of software we are aware of, which made it a natural target to evaluate accuracy and scalability of our approach. However, the approach can be implemented for other software families as well, given there is some way to extract feature identifiers and feature constraints from all sources of variability. This is probably always the case for the implementation space (code), which is generally configured by `CPP` or some similar preprocessor. Extracting the variability from the configuration space is straight-forward, too, as long as features and constraints are described by some semi-formal model (such as KCONFIG). The configurability of eCos, for instance, is described in the configuration description language (CDL) [Massa 2002], whose expressiveness is comparable to KCONFIG.

KCONFIG itself is employed by more and more software families besides Linux. Examples include OpenWRT[9] or

[8] `http://kerneltrap.org/mailarchive/linux-ext4/2010/2/8/6762333/thread`

[9] `http://www.openwrt.org`

BusyBox.[10] For these software families our approach could be implemented with minimal effort.

However, even if the system software is configured by a simple configure script (such as FreeBSD), it would still be possible to extract feature identifiers and, hence, use our approach to detect symbolic configurability defects – which in the case of Linux account for 74 percent of all defects. Feature constraints, on the other hand, are more difficult to extract from configure files, as they are commonly given as human-readable comments only. A possible solution might be to employ techniques of natural language processing to automatically infer the constraints from the comments, similar to the approach suggested in [Tan 2007].

In a more general sense, our approach could be combined with other tools to make them *configurability aware*. For instance, modifications on in-kernel APIs and other larger refactorings are commonly done tool assisted (e.g., Padioleau [2008]). However, refactoring tools are generally not aware of code liveness and suggest changes in dead code. Our approach contributes to avoiding such useless work.

5.4 Variability-Aware Languages

The high relevance of static configurability for system software gives rise to the question if we are in need of better programming languages. Ideally, the language and compiler would directly support configurability (implementation and configuration), so that symbolic and semantic integrity issues can be prevented upfront by means of type-systems or at least be checked for at compile-time.

With respect to implementation of configurability it is generally accepted that `CPP` might not be the right tool for the job [Liebig 2010, Spencer 1992]. Many approaches have been suggested for a better separation of concerns in configurable (system) software, including, but not limited to: object-orientation [Campbell 1993], component models [Fassino 2002, Reid 2000], aspect-oriented programming (AOP) [Coady 2003, Lohmann 2009], or feature-oriented programming (FOP) [Batory 2004]. However, in the systems community we tend to be reluctant to adopt new programming paradigms, mostly because we fear unacceptable runtime overheads and immature tools. C++ was ruled out of the Linux kernel for exactly these reasons.[11] The authors certainly disagree here (in previous work with embedded operating systems we could show that C++ class composition [Beuche 1999] and AOP [Lohmann 2006] provide excellent means to implement overhead-free, fine-grained static configurability). Nevertheless, we have to accept `CPP` as the still de-facto standard for implementing static configurability in system software [Liebig 2010, Spinellis 2008].

With respect to modeling configurability, feature modeling and other approaches from the product line engineering domain [Czarnecki 2000, Pohl 2005] provide languages and tooling to describe the variability of software systems, including systematic consistency checks. KCONFIG for Linux or CDL for eCos fit in here. However, what is generally missing is the bridge between the modeled and the implemented configurability. Hence tools like the UNDERTAKER remain necessary.

[10] `http://www.busybox.net`

[11] *Trust me – writing kernel code in C++ is a BLOODY STUPID IDEA* LINUS TORVALDS [2004], `http://www.tux.org/lkml/#s15-3`

6. Related Work

Automated bug detection by examining the source code has a long tradition in the systems community. Many approaches have been suggested to extract rules, invariants, specifications, or even misleading source-code comments from the source code or execution traces [Engler 2001, Ernst 2000, Kremenek 2006, Li 2005, Tan 2007]. Basically, all of these approaches extract some *internal model* about what the code *should* look like/behave and then match this model against the reality to find defects that are potential bugs. For instance, iComment [Tan 2007] employs means of natural language processing to find inconsistencies between the programmer's intentions expressed in source-code comments and the actual implementation; Kremenek [2006] and colleagues use logic and probability to automatically infer specifications that can be checked by static bug-finding tools. However, none of the existing approaches takes *configurability* into account when inferring the internal model. In fact, the existing tools are more or less *configurability agnostic* – they either ignore configuration-conditional parts completely, fall back to simple heuristics, or have to be executed on preprocessed source code. Thereby, important information is lost. Our analysis framework could be combined with these approaches to make them configurability-aware and to systematically improve their coverage with respect to the (extremely high) number of Linux variants. However, we also think that configurability has to be understood as a significant source of bugs in its own respect. Our approach does just that.

A reason for the fact that existing source-code analysis tools ignore configurability (more or less) might be that conditionally-compiled code tends to be hard to analyze in real-world settings. Many approaches for analyzing conditional-compilation usage have been suggested, usually based on symbolic execution. However, even the most powerful symbolic execution techniques (such as KLEE [Cadar 2008]) would currently not scale to the size of the Linux kernel. Hence, several authors proposed to apply transformation systems to symbolically simplify `CPP` code with respect to configurability aspects [Baxter 2001, Hu 2000]. Our approach is technically similar in the sense that we also analyze only the configurability-related subset of `CPP`. However, by "simulating" the mechanics of the `CPP` using propositional formulas [Sincero 2010], we can more easily integrate (and check against) other sources of configurability, such as the configuration-space model.

So far we have submitted 123 patches to the Linux community, which is a reasonably high number to confirm many observations of Guo [2009]: Patches for actively-maintained files are *a lot* more likely to receive responses. It really is worth the effort to figure out who is the principal maintainer (which is not always obvious) and to ensure that patches are easy reviewable and easy to integrate.

7. Summary and Conclusions

> *`#ifdef`'s sprinkled all over the place are neither an incentive for kernel developers to delve into the code nor are they suitable for long-term maintenance.*[12]

To cope with a broad range of application and hardware settings, system software has to be highly configurable. Linux 2.6.35, as a prominent example, offers 11,283 configurable features (KCONFIG items), which are implemented at compile time by 82,116 conditional blocks (`#ifdef`, `#elif`, ...) in the source code. The number of features has more than doubled within the last five years! From the maintenance point of view, this imposes big challenges, as the configuration model (the selectable features and their constraints) and the configurability that is *actually* implemented in the code have to be kept in sync. In the case of Linux, this has led to numerous inconsistencies, which manifest as dead `#ifdef`-blocks and bugs.

We have suggested an approach for automatic consistency checks for compile-time configurable software. Our implementation for Linux has yielded 1,776 configurability issues. Based on these findings, we so far have proposed 123 patches (49 merged, 8 accepted, 15 acknowledged) that fix 364 of these issues (among them 20 confirmed new bugs) and improve the Linux source-code quality by removing 5,129 lines of unnecessary `#ifdef`-code. The performance of our tool chain is good enough to be integrated into the regular Linux build process, which offers the chance for the Linux community to prevent configurability-related inconsistencies from the very beginning. We are currently finalizing out tools in this respect to submit them upstream.

The lesson to learn from this paper is that configurability has to be seen as a significant (and so far underestimated) cause of software defects in its own respect. Our work is meant as a call for attention on these problems – as well as a first attempt to improve on the situation.

Acknowledgments

We would like to thank our students Christian Dietrich and Christoph Egger for their enduring and admiring work on the implementation and evaluation. We thank Julia Lawall for inspiring conversations and her helpful comments on an early version of this work. Many thanks go to the anonymous reviewers for their constructive comments that helped to improve this paper a lot as well as to our shepherd Dawson Engler.

[12] Linux maintainer Thomas Gleixner in his ECRTS '10 keynote *"Realtime Linux: academia v. reality"*. `http://lwn.net/Articles/397422`

References

[Batory 2004] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society Press, 2004.

[Baxter 2001] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE '01)*, page 281, Washington, DC, USA, 2001. IEEE Computer Society Press. ISBN 0-7695-1303-4.

[Benavides 2005] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAISE '05)*, volume 3520, pages 491–503, Heidelberg, Germany, 2005. Springer-Verlag.

[Berger 2010] Thorsten Berger and Steven She. Formal semantics of the CDL language. Technical note, University of Leipzig, 2010.

[Beuche 1999] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[Cadar 2008] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th Symposium on Operating System Design and Implementation (OSDI '08)*, Berkeley, CA, USA, 2008. USENIX Association.

[Campbell 1993] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9), 1993.

[Coady 2003] Yvonne Coady and Gregor Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 50–59, Boston, MA, USA, March 2003. ACM Press.

[Czarnecki 2000] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000. ISBN 0-20-13097-77.

[Czarnecki 2007] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 23–34. IEEE Computer Society Press, Sept. 2007.

[Engler 2001] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[Ernst 2000] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE ’00)*, pages 449–458, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-206-9.

[Fassino 2002] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. THINK: A software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 73–86. USENIX Association, June 2002.

[Guo 2009] Philip J. Guo and Dawson Engler. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the 2009 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 2009. USENIX Association. ISBN 978-1-931971-68-3.

[Hu 2000] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *Proceedings of the 16th IEEE International Conference on Software Maintainance (ICSM’00)*, page 196, Washington, DC, USA, 2000. IEEE Computer Society Press. ISBN 0-7695-0753-0.

[Kremenek 2006] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *7th Symposium on Operating System Design and Implementation (OSDI ’06)*, pages 161–176, Berkeley, CA, USA, 2006. USENIX Association.

[Li 2005] Zhenmin Li and Yuanyuan Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference and the 13th ACM Symposium on the Foundations of Software Engineering (ESEC/FSE ’00)*, pages 306–315, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-014-0.

[Liebig 2010] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE ’10)*, New York, NY, USA, 2010. ACM Press.

[Lohmann 2009] 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 *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association. ISBN 978-1-931971-68-3.

[Lohmann 2006] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys ’06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

[Massa 2002] Anthony Massa. *Embedded Software Development with eCos*. New Riders, 2002. ISBN 978-0130354730.

[Padioleau 2008] Yoann Padioleau, Julia L. Lawall, Gilles Muller, and René Rydhof Hansen. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys ’08)*, Glasgow, Scotland, March 2008.

[Palix 2010] Nicolas Palix, Julia Lawall, and Gilles Muller. Tracking code patterns over multiple software versions with Herodotos. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD ’10)*, pages 169–180, New York, NY, USA, 2010. ACM Press. ISBN 978-1-60558-958-9.

[Parnas 1979] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[Pohl 2005] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005. ISBN 978-3540243724.

[Reid 2000] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *4th Symposium on Operating System Design and Implementation (OSDI ’00)*, pages 347–360, Berkeley, CA, USA, October 2000. USENIX Association.

[Sincero 2010] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE ’10)*, New York, NY, USA, 2010. ACM Press.

[Spencer 1992] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[Spinellis 2008] Diomidis Spinellis. A tale of four kernels. In Wilhem Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE ’08)*, pages 381–390, New York, NY, USA, May 2008. ACM Press. ISBN 987-1-60558-079-1.

[Tan 2007] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP ’07)*, pages 145–158, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-591-5.

[Zengler 2010] Christoph Zengler and Wolfgang Küchlin. Encoding the Linux kernel configuration in propositional logic. In Lothar Hotz and Alois Haselböck, editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*, pages 51–56, 2010.

A Robust Approach for Variability Extraction from the Linux Build System

Christian Dietrich Reinhard Tartler
Wolfgang Schröder-Preikschat Daniel Lohmann
{dietrich, tartler, wosch, lohmann}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

With more than 11,000 optional and alternative features, the Linux kernel is a highly configurable piece of software. Linux is generally perceived as a textbook example for preprocessor-based product derivation, but more than 65 percent of all features are actually handled by the build system. Hence, variability-aware static analysis tools have to take the build system into account.

However, extracting variability information from the build system is difficult due to the declarative and turing-complete MAKE language. Existing approaches based on text processing do not cover this challenges and tend to be tailored to a specific Linux version and architecture. This renders them practically unusable as a basis for variability-aware tool support – Linux is a moving target!

We describe a *robust* approach for extracting implementation variability from the Linux build system. Instead of extracting the variability information by a text-based analysis of all build scripts, our approach exploits the build system itself to produce this information. As our results show, our approach is robust and works for all versions and architectures from the (git-)history of Linux.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; D.2.9 [**Management**]: Software configuration management

General Terms

Design, Experimentation, Management

Keywords

Configurability, Maintenance, Linux, Build Systems, Kbuild, Static Analysis, VAMOS

1. INTRODUCTION

System-software product lines usually employ compile-time configuration as a simple and widely used technique for tailoring with respect to a broad range of supported hardware architectures and application domains. A prominent example is the Linux kernel.

SPLC '12, September 02 – 07 2012, Salvador, Brazil

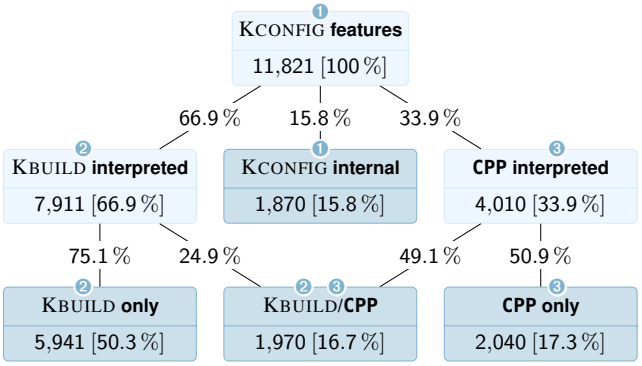


Figure 1: Statistic how the features, declared in KCONFIG, are referenced by source-code and Makefiles in Linux v3.2

The Linux KCONFIG feature model provides more than 11,000 configurable features in Linux v3.2. The thereby described *intended* variability is implemented by 28,000 source files containing 84,000 `#ifdef`-blocks.

In previous work, we could show that intended and actually implemented variability (i.e., the KCONFIG feature model and the variability points in the code) do not necessarily match. However, many configurability-related defects, such as dead `#ifdef`-code, and bugs, can be found upfront by better tool support [29]. This eventually has led to (accepted) fixes for twenty new bugs and the removal of 5,000 superfluous lines of `#ifdef`-code in Linux v2.6.36. However, these numbers are just the tip of an iceberg. The lesson to be learned from this is: Variability has to be understood, analyzed, and tested as a system property in its own respect. For a system-software product line at the size of Linux, this requires profound and robust tool support.

1.1 The Role of the Build System

A crucial building block for variability-aware static checking tools are reliable extractors that transform the *actually implemented* variability from their various sources into a formal model. Existing studies (including our own) have mostly focused on the C Preprocessor (`CPP`) as a means to implement features in Linux [13, 14, 25, 28, 29]; however, in Linux, variability is mostly implemented in a more coarse-grained manner (Figure 1): Only a third (33.9%) of all features do affect the work of the `CPP`, that is, have an effect on the sub-file level. On the other hand, two third (66.9%) of all features are referenced in the build system (KBUILD). These features have an effect on the selection of whole files into the build process. Hence, we need robust tools to extract the implementation variability from the Linux build system.

1.2 Related Work in a Nutshell

Approaches to extract implementation variability from KBUILD have previously been published by Berger et al. [4] and Nadi and Holt [18]. A common characteristic of both approaches is that they rely on *text processing* of makefiles, that is, they employ parsing (Berger et al.) or clever regular expressions (Nadi and Holt) to extract the presence implications for Linux source files from the build scripts. However, the underlying MAKE language is a declarative and turing-complete language; its advanced features, such as the `$(eval)`, `$(shell)`, or `$(wildcard)` functions, make it notoriously difficult to analyze. If these features are used, a text-processing–based approach quickly hits its limits, since the enclosed fragments may be for instance arbitrary shell command.

Even worse from a practical point of view is, however, that the existing approaches are brittle with respect to evolutionary changes in the KBUILD system itself: To achieve good results, they have to provide explicit support for many corner cases of KBUILD analysis, which effectively tailors them for a specific Linux version and architecture. While this might be perfectly acceptable if the goal is to analyze a certain Linux version, it renders them as practically unusable as a basis for general variability-aware tool support – Linux is a moving target.

1.3 About this Paper

The contribution of this paper is a *robust* approach for extracting implementation variability from the Linux KBUILD system. Instead of text processing, our approach exploits the build system itself to produce this information. Thereby, our approach is not only simple to implement, but also robust with respect to evolutionary changes and the usage of advanced MAKE features. Our evaluation results show that our approach works for all versions and architectures from the (git-)history of Linux and reliably extracts presence conditions for more than 93% percent of all source code files. In two applications, we show that the presented implementation significantly improves our previous results on configuration defects [29] and configuration coverage (CC) [28].

The context of this work is the VAMOS [1] project, funded by the German Research Council (DFG). The goal is to provide practical tools for analysis and management of variability in system software. So far the produced tool hav produced over 100 patches that have been integrated into the Linux mainline kernel.

The remainder of this paper is structured as following: In Section 2, we introduce the background and technical context and analyze the challenges in build-system analysis. This is followed by the description of the basics of our approach in Section 3. Then, we analyze the results in Section 4, followed by two applications in Section 5. After discussing the results in Section 6 and an overview over further related work in Section 7, the paper concludes with Section 8.

2. VARIABILITY IN LINUX

The scattered nature of variability and variability implementation in Linux makes holistic reasoning challenging. In practice, the analysis of the different models, languages and representations of variability requires very specialized and sophisticated extraction tools. A solid understanding of how the Linux build system KBUILD and the configuration tool KCONFIG play together is instrumental to correctly relate variability implementations from different extraction tools. This subsection analyzes the mechanics of KBUILD and identifies the challenges for an automated extraction of variability.

[1] **V**ariability **M**anagement in **O**perating **S**ystems

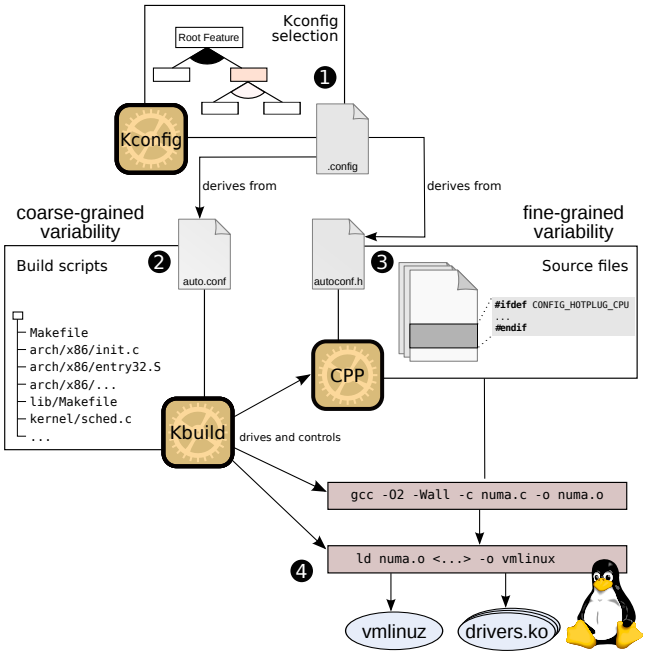


Figure 2: Overview of the technical realization of software variability in Linux. The coarse-grained variability implemented in makefile dominates fine-grained variability in CPP code.

2.1 Levels of Variability

In a nutshell, static configurability is specified and implemented in Linux top-down on three major levels, for which Figure 1 illustrates their quantitative relevance:

❶ The configuration system (KCONFIG) defines the available features and their constraints (*intended variability*) and provides an interface to specify and manage a concrete (product) **configuration**.

❷ The build system (KBUILD) implements *coarse-grained variability* in the code by inclusion and exclusion of complete translation units in the build process. The produced **build products** include object files, the bootable kernel image and loadable kernel modules (LKMs).

❸ The CPP implements *fine-grained variability* by inclusion or exclusion of `#ifdef`-blocks within the files selected by KBUILD.

Figure 2 describes the Linux toolchain that drives the compilation process. At the top, the Linux feature model defines the (intentional) **product line variability** [16, 20]. Here, the user selects a concrete product configuration with the KCONFIG tool and saves his selection to a file named `.config`. The Linux build system KBUILD transforms the thereby encoded feature selection into two further representations: An `auto.conf` file in MAKE-syntax and an `autoconf.h` file in CPP syntax (Figure 3). Technically, these representations control the (extensional) **software variability** [16, 27] in makefiles and C source code during the compilation process.

For the CPP representation, an additional normalization step is applied for tristate features: Many features, especially device drivers, can be configured as *compiled into kernel*, *compiled as loadable kernel module* or *disabled*. To ease the use in `#ifdef` statements, KCONFIG maps this to boolean flags by inserting an additional CPP variable with the `_MODULE`, suffix into `autoconf.h` for each tristate feature (Figure 3).

(a) KCONFIG output: .config

```
SMP=n
PM=y
APM=m
```

(b) MAKE representation: auto.conf

```
CONFIG_SMP     := n
CONFIG_PM      := y
CONFIG_APM     := m
```

(c) CPP representation: autoconf.h

```
#undef  CONFIG_SMP
#define CONFIG_PM           1
#undef  CONFIG_APM
#define CONFIG_APM_MODULE   1
```

Figure 3: Representation of a feature selection

In Step ❷, the MAKE representation of the current feature selection is then used by KBUILD to implement the coarse-grained variability on a per-file basis. All thereby included translation units are passed to the compiler, which in turn uses the CPP representation during preprocessing (Step ❸) to implement the *fine-grained* configurability. The invocation of the compiler and linker is, however, controlled by KBUILD,[2] which again uses the MAKE representation of the current feature selection to construct compiler and linker options used in Step ❹ for creating the build goals: The `vmlinuz` kernel image and the library of loadable driver objects.

2.2 Variability Implementation in Kbuild

As detailed in the previous section, KBUILD gets a file `auto.conf` that describes all selected features and their values in MAKE syntax. KBUILD then resolves which file implements what feature, determines the set of translation units that are relevant for a given configuration selection, and invokes the compiler for each translation unit with potentially configuration-dependent settings and compilation flags. Internally, KBUILD employs GNU MAKE [26] to control the actual build process; in Linux v3.2 the mapping from features to translation units is encoded in 1,568 makefiles that are spread across the source tree. However, Linux makefiles look quite different from typical text-book makefiles as they employ KBUILD-specific idioms to implement Linux-specific (variability) requirements, such as [11]:

- **Optional features:** Many features, such as drivers, are present (or absent) by deciding about the inclusion of their respective implementation files.
- **Tristate features:** Linux allows most drivers to be compiled either statically into the kernel or as LKM.
- **Loose coupling:** The decision about what set of files is used for a given configuration can be specified at various levels of granularity (such as disabling a complete subsystem by not descending a subdirectory).

In the following, we provide further details on these idioms, as they are relevant for this paper.

2.2.1 *Optional and Tristate Features*

In all makefile fragments, we can find two variables that collect selected and unselected object files: The make variable `obj-y` contains the list of all files that are to be statically compiled into the kernel. Similarly, the variable `obj-m` collects all object files that will be compiled as LKM. Object files in the make variable `obj-n` are not considered for compilation. The suffixes {`y`,`m`,`n`} are added by the expansion of variables from `auto.conf` (Figure 3).[3] This pattern for managing variability with KBUILD is best illustrated by a concrete example:

```
1  obj-y              += fork.o
2  obj-$(CONFIG_SMP)  += spinlock.o
3  obj-$(CONFIG_APM)  += apm.o
```

In line 1, the target `fork.o` is unconditionally added to the list `obj-y`, which instructs KBUILD to compile and link the file directly into the kernel. In line 2, the variable `CONFIG_SMP`, which is taken from the KCONFIG selection, controls the compilation of the target `spinlock.o`. The variable derives from the feature `SMP`, which is declared as `boolean`. Therefore, `spinlock.o` cannot be compiled as LKM. When the feature selection from Figure 3 (b) is applied, `CONFIG_SMP` has the value `n`, `spinlock.o` is added to `obj-n` and therefore not compiled. In line 3 the file `apm.o` is handled in a similar way to `spinlock.o`. Because the enabling feature `APM` is declared as *tristate*, it might take value `m`. With the feature selection from Figure 3 (b), `APM` has the value `m`, therefore `apm.o` is added to `obj-m` and compiled as LKM.

Note that instead of mentioning the source files, the makefile rules reference only the resulting build products. The mapping to source files is implemented by *implicit rules* (for details, cf. [26, Chapter 10]). This mapping has to be considered for any kind of makefile variability analysis.

2.2.2 *Loose Coupling*

Programmers specify in KBUILD makefiles the conditions that lead to the inclusion of source files in the compilation process. As shown above, this commonly happens by mentioning the respective build products in the special targets `obj-y` and `obj-m`. This works for the majority of cases, where a feature is implemented by a single implementation file. However, in order to control complete subsystems, which generally consist of several implementation files, the programmer can also include subdirectories:

```
obj-$(CONFIG_PM)  += power/
```

This line adds the subdirectory `power` conditionally, depending on the selection of the feature `PM` (power management). For each listed subdirectory, its containing Makefile is evaluated during the build process. This allows a more coarse-grained control of source file compilation with KCONFIG configuration options. As we will show later in this paper, the inclusion of most source files in Linux is controlled by enabling a single configuration option.

2.3 Challenges in Build-System Analysis

While the selection process described in Section 2.2 is conceptually simple, an automated analysis is challenging because of engineering reasons. Since KBUILD is implemented with the MAKE tool, the kernel developer has many possibilities to express constraints. Not only is MAKE a full-blown programming language that supports a wide range of operations, including string modifications, conditionals, and meta-programming, it also allows the execution of arbitrary further programs ("shell escapes"). The Linux coding guidelines do not pose any restrictions on what MAKE features should be used

[2]The exact mechanisms are fairly technical and have already been discussed elsewhere (e.g., [11, 17]).

[3]The idea of this pattern dates back to 1997 and was proposed by Micheal Elizabeth Castain under the working title "Dancing Makefiles" (`https://lkml.org/lkml/1997/1/29/1`). It was globally integrated into the kernel makefiles by Linus Torvalds shortly before the release of Linux v2.4.

in KBUILD. This subsection presents a few selected examples of constructs that are present in the build system of Linux and are far more expressive than the standard constructs.

The following example is taken from `arch/x86/kvm/Makefile` and uses the function `addprefix`:

```
obj-$(CONFIG_KVM_ASYNC_PF)  += \
  $(addprefix ../../../virt/kvm/, async_pf.o)
```

The `addprefix` function takes an arbitrary amount of arguments, prepends its first argument to the remaining ones, and returns them. In this case using `addprefix` is not really necessary, because there is only one additional argument and the whole expression is equal to `../../../virt/kvm/async_pf.o`. Nevertheless, this case requires special handling with a text-processing–based approach.

In KBUILD, programmers also use generative programming techniques and loop constructs, like in this excerpt taken from `arch/ia64/kernel/Makefile`:

```
ASM_PARAVIRT_OBJS = ivt.o entry.o fsys.o
define paravirtualized_native
AFLAGS_$(1) += -D__IA64_ASM_PARAVIRTUALIZED_NATIVE
[...]
extra-y += pvchk-$(1)
endef
$(foreach obj,$(ASM_PARAVIRT_OBJS),$(eval $(call
     paravirtualized_native,$(obj))))
```

Here, a list of implementation files (`ivt.S`, `entry.S` and `fsys.S`) not only need to be included, but also require special compilation flags. In this example, the macro `paravirtualized_native` is evaluated for all three implementation files by the MAKE tool at compilation-time. Again, for a text-processing–based approach, this corner case is challenging to implement in a general manner.

Even worse is the `shell` function, which makes it possible to spawn an arbitrary external program to let it control (parts of) the compilation process.

The text-processing–based approaches [4, 18] both fail on the examples shown above. Luckily – and this comes to their rescue – these MAKE language features are currently not used very frequently in KBUILD. However, they *are* used[4] and their usage is not discouraged by Linux coding guidelines. On the longer term, this implies a danger regarding the robustness of text processing as a means to extract variability information from the Linux build system. In the following, we therefore devise a pragmatic approach that, conceptually and practically, is robust with respect to these challenges.

3. EXPERIMENTAL PROBING FOR BUILD-SYSTEM VARIABILITY

In order to enable variability analyses, such as consistency checks with the KCONFIG feature model [29] or variability aware static analysis with existing tools [28], the results of the variability extractors may require a normalization step. Literature proposes propositional formulas as lingua franca for combining the different sources of variability (e.g., [4, 13, 15, 18]). Similar to [4, 18] , we extract propositional formulas that model the behavior of KBUILD, similar as we did in previous work for the `CPP` [24].

The set of files that KBUILD produces during the compilation process depends on selection of features done by KCONFIG. The basic idea of our approach is to (partially) execute KBUILD with different feature selections and observe the behavioral changes. This allows to correlate variability points in the feature model with the produced build products.

The presence implication of a source file is determined by the feature selections that include the file in the compilation process. Therefore, in order to extract the presence implication for a specific source file, all feature selections that enable this file need to be recorded. Our approach exploits this observation and determines for each file all selections that include the file during the compilation process.

Instead of parsing the makefile, our approach is based on "clever probing": Basically, we "ask" the build system for each feature which files it *would* built. The basic idea is to investigate a feature selection S_{base}, which uses the set F_{base} during the compilation process. Now we add one additional feature f_1 to it. The new feature selection $S_1 := S_{\text{base}} + \{f_1\}$ now compiles the set of files F_1. For every file that is in F_1 but not in F_{base} we have found a feature selection that enables this particular file.

3.1 Subdirectories

As discussed in Section 2.2.2, not necessarily all subdirectories in the Linux source tree are traversed at compilation time. Subdirectories are therefore not only used to organize files for the programmer, but also for implementing build-system variability. We address this in our approach by treating subdirectories that appear in the file sets F_{n+1} in a special way: For each subdirectory we determine the condition under which the compilation process traverses it. If the condition is non-trivial, then it is taken as precondition (the "base expression") to all presence condition of its included files. After processing all files in the file set F_n, each of the included subdirectories is processed recursively.

3.2 From Feature Selections to File Sets

Our approach relies on the following primitive operation to find the file set and all considered subdirectories that are associated to a feature selection:

$$list : \text{Selection} \longmapsto (\text{FileSet},\ \text{SubDirs}) \qquad (1)$$

This primitive is essential for any build system that implements variability. There are several options how this can be implemented for a given build system. As a last resort, the mapping could be extracted from build traces of a real build process [cf. 2]. However, in order to avoid unnecessary compilation steps, an efficient extraction of this mapping is essential.

For KBUILD, our implementation traverses the source tree in the same way the regular compilation process. Hereby, MAKE collects all selected files and the visited subdirectories into lists (technically MAKE variables), which are used internally to drive the compilation. We make use of these implementation internals and therefore exploit the built-in KBUILD functionality to ensure an accurate operation of the *list* primitive. The full implementation is available for download from the VAMOS website [30].

As an additional optimization, our implementation ignores logical constraints that stem from KCONFIG declarations, which allows us to reduce the number of necessary probing steps. This optimization would not have been possible to implement using build traces, which (successfully) compiles and links only valid configurations.

3.3 Base Selection and Added Features

The algorithm starts with the empty selection $S_\emptyset$ as starting point for the recursion, which serves as base point for the file set and subdirectory differences. The empty selection contains no selected feature at all; it is therefore not a valid configuration according to the KCONFIG model. This base file set only includes files that

[4] In Linux v3.2, we count for `shell`: 127, `foreach`: 16, `eval`: 3, and `addprefix`: 88 occurrences.

are included in every configuration. One example of such a file is `kernel/fork.c`, which is essential for the process creation and therefore needed in every configuration.

$$(F_{\text{base}}, D_{\text{base}}) = \mathit{list}(S_\emptyset) \tag{2}$$

The files F_{base} selected by $S_\emptyset$ are unconditionally compiled into the kernel. In Linux v3.2 `arch-x86`, our implementation detects 334 such unconditional files. $S_\emptyset$ also selects the subdirectories D_{base}, which are the starting point for the build system during the source tree traversal. The presented approach uses F_{base} and D_{base} in the same manner as starting point.

In the process of adding single features to the base selection, it is necessary to know which variables have to be considered. We exploit the fact that the Linux source tree is organized hierarchically: Each conditional subdirectory carries, in addition to the base selection, a base directory d_{base}. All features referenced in the makefile of a base directory are added to the list of features to probe.

$$\mathit{features\_in\_dir} : \text{Directory} \longmapsto \text{FeatureSet} \tag{3}$$

For KBUILD, the *features_in_dir* function is straight-forward to implement with regular expressions that extract all referenced variables in the Makefile that start with `CONFIG_`. This is also some sort of text processing, but in contrast to the competing approaches [4, 18], we just extract the feature identifiers and not their (context-dependent) semantics. Therefore, the *features_in_dir* function also detects referential KCONFIG ↔ KBUILD defects, similar as described by Nadi and Holt in [18]. By excluding undeclared configuration variables from the `FeatureSet`, we reduce the number of necessary probing steps.

3.4 Build-System Probing

```
1: function KBuildProbe
2:     vDirs ← empty set                          ▷ set of visited dirs
3:     filePC ← empty map [ File → list [ Selection ]]
4:     (F_base, D_base) = list(S_∅)
5:     for all d_base in D_base do
6:         KBuildProbeRecursion(d_base, S_∅, F_base)
7:     end for
8:     for all (file, selections) in filePC do
9:         toPC(file, selections)
10:    end for
11: end function
```

Figure 4: Starting-Point for the Build-System Probing

Figure 4 shows the recursion step over the source tree for probing the file presence implications. The recursion is done only once for each directory. In line 2, a set of already visited directories is initialized. The resulting selections for each file is stored in filePC, which holds a list of selections for each file (line 3). For each directory that is considered by the empty selection $S_\emptyset$, we start the recursion in line 6 to dig into the source tree beginning at the directory. After all file sets have been calculated, the presence implications for all source files are calculated by the helper function *toPC* in line 9.

In Figure 5, the recursion step, which is executed for every subdirectory that may be considered by KBUILD, is shown. The function *KBuildProbeRecursion* takes three arguments: The first argument d_{base} is the directory this function call should focus on. S_{base} contains the features that are necessary to visit d_{base} in the first place. The third argument is the file set associated with S_{base}. Our implementation avoids unnecessary recalculation of F_{base} by caching the result.

```
1: function KBuildProbeRecursion(d_base, S_base, F_base)
2:     if d_base ∈ vDirs then                     ▷ already visited
3:         return
4:     end if
5:     vDirs ← vDirs ∪ {d_base}                   ▷ mark as visited
6:     features ← features_in_dir(d_base)
7:     for all f in features do
8:         S_new ← S_base ∪ {f}                   ▷ add one feature
9:         (F_new, D_new) ← list(S_new)
10:        for all file in (F_new − F_base) do
11:            filePC[file].append(S_new)         ▷ new files found
12:        end for
13:        for all dir in (D_new − D_base) do
14:            KBuildProbeRecursion(dir, S_new, F_new)
15:        end for
16:    end for
17: end function
```

Figure 5: Recursion step in the Build-System Probing.

Lines 2 to 5 ensure that each directory is only visited once and the recursion terminates in finite steps. The function *features_in_dir* is called in line 6 to determine all features that are used in the directory's makefile. These features will be probed together with the base selection against this Makefile. A new feature selection S_{new} is created (line 8) as an extention to S_{base} for each of these features. For this feature selection, all considered files and subdirectories are collected by a call to *list* in line 9. The difference between the new file set and the old file set are all files that are additionally enabled by the current feature. The feature selection is added in line 11 to all additionally enabled files. Similar to this, we recurse into the file system hierarchy for each newly detected directory in line 14. The newly detected directory is used as base directory and S_{new}, with the associated file set, as base selection. The conversion from the feature selections to the presence implications for a file is straightforward:

$$\mathit{toPC}(\text{File}, \text{Selection}) = \text{File} \rightarrow \bigvee_{S \in \text{Sels}} \left(\bigwedge_{f \in S} f \right) \tag{4}$$

Each selection is a conjunction of the set features that must be enabled in order satisfy the developer-specified KBUILD constraint to compile the file. Multiple selections occur when there are multiple rules that require the source file to be compiled. In case of multiple selections, all selections are disjuncted, because any of these disjunction leads to the inclusion of the file in the compilation process.

The resulting propositional formula can by simplified, for instance by removing selections that are a full subset of another selection.

4. EVALUATION

In the following, we evaluate our approach and compare it to the existing approaches. We start with a general description and compare the three respective implementations, which is followed by analyses regarding run time, robustness and coverage.

4.1 Implementation Overview

The GOLEM tool

We have implemented the algorithms from Section 3 into the GOLEM tool which is part of our VAMOS [30] toolchain [28, 29]. The implementation encompasses about 1,000 lines of Python code.

The KBUILD specific probing primitives are implemented in two additional "front-end" makefiles (about 120 lines of MAKE code), so that not a single line in Linux had to be changed for the analysis. The tools are freely available on the project website.

KBUILDMINER

KBUILDMINER by Berger and She [3] has been presented on a poster at SPLC '10 [5] and is further detailed in a technical report [4]: A fuzzy parser transforms KBUILD makefiles into an abstract syntax tree (AST), which is then transformed into presence conditions. The implementation consists of about 1,400 lines of Scala code and 450 lines of Java code. The tool, as well as a result set for Linux v2.6.33.3, have been downloaded from [3]. Because this tool requires manual modification of existing makefiles (the technical report states that for Linux v2.6.28.6, 28 makefiles were adapted manually [4]), it is not easily possible to apply it to arbitrary versions of Linux.

The UNDERTAKER Extension by Nadi

Nadi and Holt [18] have implemented their KBUILD extractor independently from us. Similar to our GOLEM tool, this extractor calculates logical constraints that our UNDERTAKER tool [29] can use directly. Their implementation employs pattern matching in Linux makefiles to identify variability in KBUILD. It consists of about 750 lines of Java code. While not (yet) publicly available, the authors have kindly provided us with the version that has been used in [18].

4.2 Runtime

All parsing-based approaches are (persumably) much faster than the GOLEM implementation presented in this paper. For KBUILDMINER [4], no run-time data is available. The parser by Nadi and Holt [18] processes a architecture in under 30 seconds. The current GOLEM implemenation takes approximately 90 minutes per architecture. The obvious bottleneck is the run-time and the amount of probing steps, which have been described in Figure 4. For Linux v3.2 `arch-x86`, the *list* operation takes about a second (depending on the selected features and filesystem cache state) and was executed 7,073 times.

However, the *list* function does neither modify the analyzed source tree, nor exhibit other side effects. We therefore see a great potential in improving the performance by running several probing steps in parallel. For practical applications, the large runtime overhead has little big impact on the usability of the approach, because for many applications, such as the applications in Section 5, the variability extraction has to be done only once per version and architecture.

4.3 Robustness

As Linux is a moving target, variability identification and extraction approaches need to be both conceptually as well as implementation-wise robust. In order to evaluate the property of robustness for future versions of Linux, we test on a wide-ranged number of Linux versions have been retrieved from the git history. We choose five Linux releases with one year distance that cover 4 years of the Linux development (2008-2012). In order to keep the results for the various implementations comparable, we refrain from analyzing earlier versions than Linux v2.6.25, because the `arch-x86` architecture was introduced in v2.6.24 by merging the 32bit and 64bit variants, which were previously maintained separately. Table 1 summarizes the results of this analysis.

In general we found it challenging to apply the parsing-based approaches to Linux versions for which they have not been tailored to.

Table 1: Direct quantitative comparsion over Linux versions over the last 5 years. The Kernel versions are roughly equidistant over the time and include all version for which dataset are available for KBUILDMINER and the Nadi Parser.

All source files for v2.6.25 (w/o `#included` files)	6,826	(127)
Files hit by KBUILDMINER	*data not available*	
Files hit by GOLEM	6,274	(93.7%)
Files hit by Nadi parser	*tool crashes*	
All source files for v2.6.28.6 (w/o `#included` files)	7,665	(153)
Files hit by KBUILDMINER	7,243	(96.4%)
Files hit by GOLEM tool	7,032	(93.6%)
Files hit by Nadi parser	*tool crashes*	
All source files for v2.6.33.3 (w/o `#included` files)	9,827	(261)
Files hit by KBUILDMINER	9,090	(95%)
Files hit by GOLEM	9,079	(94.9%)
Files hit by Nadi parser	7,154	(74.8%)
All source files for v2.6.37 (w/o `#included` files)	10,958	(292)
Files hit by KBUILDMINER	*data not available*	
Files hit by GOLEM	10,145	(95.1%)
Files hit by Nadi parser	7,916	(74.2%)
All source files for v3.2 (w/o `#included` files)	11,862	(276)
Files hit by KBUILDMINER	*data not available*	
Files hit by GOLEM	11,050	(95.4%)
Files hit by Nadi parser	8,592	(74.2%)

For the fuzzy-parsing approach presented by Berger et al. [4], there are only data sets for Linux version v2.6.28.6 [4] and v2.6.33.3 [3] available. For all other versions we were unable to produce any results, because of the necessary (but undocumented) changes of the Linux makefiles. These modifications include the disabling parts of `arch/x86/Makefile` in a way that break a regular compilation. The technical report leaves it open what effects these changes have on the extracted logical constraints.

The parsing approach presented by Nadi and Holt [18] does not require any modifications to existing Makefiles. We were able to produce presence implications for two additional versions. Unfortunately, the tool crashes with an endless recursion and a stack overflow on Linux v2.6.28.6 and earlier, so that no logical constraints could be obtained.

The presented approach and implementation in this article produces presence implications on all selected versions without requiring any source code modification or version specific adaptations. Also, the extraction process for the 22 other architectures in Linux v3.2 did not require any further modification.

As shown in this section, both parsing-based approaches have difficulties to achieve a robust operation on a wide range of versions. Since the Linux build system is still in active development and difficulties like those described in Section 2.3 may appear with every new version, every new introduced MAKE idiom requires manual (and thus error-prone) additional engineering in order to keep up with the Linux development. In contrast to to that, our approach works in a robust manner with stable results for each version without any further adaptations.

4.4 Coverage

This subsection compares the results of the three KBUILD variability extractors quantitatively. We do this by analyzing for how many source files the respective approach produces a logical formula as metric for their coverage in the Linux v2.6.33.3 source tree for `arch-x86`. We choose this source tree because it is the most recent version of Linux for which results of all tools are available.

For that version, KBUILD handles a total of 9,827 source files. As pointed out by Nadi and Holt [17], 276 of these source files (2.8%) are referenced by `#include`-statements in other implementation

Table 2: Configuration Defect Analysis Results with Linux v3.2

Configuration Defects without file constraints	
Code defects	1835
Referential defects	415
Logical defects	83
Total:	Σ 2333
Configuration Defects with file constraints	
Code defects	1835
Referential defects	439
Logical defects	299
Total:	Σ 2573

source files rather than KBUILD rules in KBUILD.

The UNDERTAKER extension by Nadi and Holt [18] approach identifies presence implications for 7,154 out of all source files (74.8%). For 2,412 source files, no logical implication was found. A quick analysis of the data indicates that deficiencies in the mapping from build products to source files (cf. Section 2.2.1) are part of the problem for this relatively high number.

An analysis of the data provided for KBUILDMINER [3] on the tool's website for `arch-x86` shows that the tool produces presence implications for 9,090 out of all source files (95%) on Linux v2.6.33.3, `arch-x86`. This data is consistent to the technical report [4], which states a coverage of 94 percent on Linux v2.6.28.6, `arch-x86`.

The current implementation of our GOLEM tool calculates presence implications for 9,079 out of the 9,566 source files on Linux v2.6.33.3 (94.9%) on `arch-x86`.

5. APPLICATIONS

As part of the VAMOS project [30], we aim at providing (Linux) developers tool support for managing and maintaining variability. This goal includes finding configuration defects [29] and making existing tools for static analysis variability-aware [28]. The remainder of this section demonstrates the improvements of considering the build system in these tools.

5.1 Configuration Defect Analysis

In earlier work [29], we have discussed and analyzed configuration-derived defects in the variability implementation on an earlier version of Linux v2.6.35. Such defects are inconsistencies in the variability implementation, such as `#ifdef` blocks that either cannot be selected under any configuration selection (a *dead* block), or there is provably no configuration that deselects a `CPP` block (an *undead* block). Our UNDERTAKER tool creates for each `#ifdef` block a set of propositional formulas and checks their satisfiability with a SAT Checker. The first formula includes only the constraints that are found in the structure of the `CPP` statements [cf. 24]. If this formula is unsatisfiable, then the block is classified as a *code defect*. If it is satisfiable, logical constraints that derive from the KCONFIG feature model are added as further conjunctions to the formula. If the enriched formula is unsatisfiable, the UNDERTAKER tool classifies the `CPP` block as a *logical defect*. This formula may (still) contain configuration variables that are not declared in the configuration model for this architecture (e.g., `CONFIG_ARM` is not present on `arch-x86`, etc.). The third formula therefore adds contraints to set such absent variables to false, and checks for satisfiability again. If this enriched formula is now unsatisfiable, then the UNDERTAKER tool classifies the `CPP` block as *referential defect*.

For this kind of analysis, our tools, which (now) include the extracted variability from KBUILD, do not only need to be robust regarding the Linux version, but also the analyzed architecture. A more detailed explanation of this experiment can be found in [29]. That work has yielded 1,776 configurability issues, for which 123 patches has been proposed (49 merged, 8 accepted, 15 acknowledged), which in total have fixed 364 of these issues (among them 20 confirmed new bugs).

Table 3: CC-Analysis Results with Linux v3.2, `arch-x86`

Analyzed files	10,383
Number of variation points (files + `#ifdef` blocks)	25,369
1. Comparison with 'allyesconfig'	
Number of compiler (tool) invocations	10,383
Rate of skipped invocations	18.5%
Configuration Coverage	67.2%
2. Expansion without file constraints	
Number of partial configurations	14,169
Rate of skipped tool invocations (partial configurations)	83.6%
Configuration Coverage	37.4%
3. Expansion with file constraints	
Number of partial configurations	12,388
Rate of skipped tool invocations (partial configurations)	18.2%
Configuration Coverage	78.6%

Table 2 compares the impact of the inclusion of the extracted source file constraints by our GOLEM tool on the results produced by the approach as presented in [29]. In this experiment, source file constraints from all 23 architectures in Linux v3.2 have been used to enrich the variability models. Every defect is tested against each architecture individually (where applicable) and classified as such.

In this work, we define as **variation point** every `CPP` block and source file that KCONFIG allows to include or exclude in the resulting build products. This simplifaction is valid, because the coarse-grained selection of source files by MAKE could also be implemented by `CPP` by introducing additional `#ifdef` blocks that contain the whole file.

We did not find any *dead* source files, that is, files that will never be compiled due to the constraints from KBUILD. We can therefore confirm that the contributions of Nadi and Holt [18] have fixed all these "dead files". Nevertheless, by considering KBUILD-derived constraints, the UNDERTAKER tool detects 216 additional (+260.2%) logical defects in `#ifdef`-blocks. The number of configuration defects increases by 10.3 percent. This shows that the source-file constraints have an considerable improvement on the results.

5.2 Configuration Coverage

This subsection investigates the effects of the extracted source-file constraints on the configuration coverage (CC) [28]: We define CC as the fraction of selected variation points (`#ifdef`-blocks and source files as defined in Section 2.1) divided by all possible variation points. However, one has to be careful with calculating the "possible" variation points on a specific architecture, because architecture-specific drivers or `#ifdef` blocks that test for a specific other architecture must not be counted. In order to get a fair comparison, we use our UNDERTAKER tool to detect such unselectable variation points in the 11,862 source files considered by KBUILD on `arch-x86` and exclude them from all results in this subsection.

We calculate a set of configurations which, when combined (i.e., compile each configuration individually), maximize the CC. This allows "traditional" tools for static analysis to uncover additional defects that are hidden in seldomly selected `#ifdef`-blocks. Table 3 summarizes the results. Since the analyzed source files only reference a subset of all available KCONFIG features, the produced configuration are "incomplete" in the sense that they define only referenced features. Such a *partial configuration* sets only variation

points from the extracted software variability [27] of a given source file. The remaining, unreferenced features need to be set in a way that they do not conflict in order to obtain a concrete product configuration, upon which traditional tools for static analysis can be employed. We use the KCONFIG tool to *expand* such partial to *full* configurations.

For comparison purposes, we first calculate the CC for the KCONFIG provided configuration preset `allyesconfig`. Interestingly, `allyesconfig` is way off from a "full" configuration, as 1,917 (18.5%) of all source files for `arch-x86` are **not** compiled. This, and the fact that every file with `#else` and `#elif` statements require more than one configuration to select all lines of code, account for the missing 32.8% CC.

In previous work [28], we have calculated partial configurations on all source files, and applied the KCONFIG infrastructure to expand each partial configuration to a full configuration. In this work, we consider both, KCONFIG-controlled `#ifdef` blocks (i.e., `#ifdef` blocks with a logical expression that contains at least one reference to a variable that starts with `CONFIG_`), as well as the inclusion of a source file into the compilation process, as a variation point. Therefore, the numbers of the calculated CC are hard to compare to those in our previous work [28].

Table 3 shows that the number of calculated configurations is not much higher than the number of analyzed source files (about 19.3% more configurations than source files). This number is surprisingly low because most files in Linux do not contain `#ifdef` blocks, but are controlled by at most a single MAKE variable (cf. Section 2.2). This means the majority of files in Linux require only a single configuration to achieve full CC.

For each partial configuration, we check if the respective expanded configuration would actually let KBUILD include the file in the build process. Because of uncovered source file constraints in the GOLEM implementation and incompleteness of our KCONFIG variability model, this is not always the case. We do not count variation points of a partial configuration that does not include its corresponding file, because this configuration does not practically cover any variation point.

When calculating the CC without considering source file constraints (the second experiment in Table 3), we notice a coverage of only 9,492 out of 25,369 (37.4%) possible variation points. The reason for this alarmingly low rate is that 11,844 out of 14,169 (83.6%) variation points have not been considered, because the calculated configuration did not compile the source file for which it has been calculated.

When calculating the CC with considering the file constraints (the third experiment in Table 3), we observe a CC of 19,938 out of 25,369 (78.6%) variation points. The reason for this improvement is that the rate of skipped configurations decreases dramatically to 16.4 percent. This number is still considerable. Since each skipped configuration provably contains skipped variation points, we expect that additional engineering (cf. Section 6.1) will considerably increase the CC even further. Additionally, a first analysis of the calculated partial configurations shows that the quality of the expansion process still leaves room for improvement: In many expanded configurations, we observe omitted and wrongly set features. Improving the expansion process would therefore improve the achieved CC as well.

Because of the skipped partial configurations and the deficiencies in the expansion process, the improvement of the calculated CC has to be seen as lower bound that can be greatly improved by more precise MAKE and KCONFIG models, and better expansion of partial configurations. We are currently working on improving these results.

6. DISCUSSION

As demonstrated by the two applications in the previous section, the implementation of our approach greatly assists variability-aware analyses. This subsection discusses the limitations and in what way the results can be transferred to other systems.

6.1 Benefits and Limitations of the Approach

Compared to parsing-based approaches for extracting variability from the build system [e.g., 4, 13, 18] our approach of build-system probing exhibits a number of unique characteristics. While existing parsing-based approaches suffer from technical implementation challenges that require manual (and error-prone) engineering for the many corner-cases, our approach handles complicated makefile constructions as presented in Section 2.3 and shell escapes (i.e., invocation of external tools in the build system) error-free. It is also much harder, as presented in Section 4.3, for a parsing based approach to keep pace with the Linux development, whereas our approach works predictably for a wide range of Linux versions and architectures.

However, we also make a number assumptions on the build system, which may impact the results of our approach:

1. We exploit the observation that the file presence implications in KBUILD correspond to the hierarchical organiztion of directories along subsystems. If a feature is a prequisite for enabling files in a subdirectories, then this constraint applies for each file in that directory.

2. We assume that in a subdirectory, each file is only dependant on single features and not by a conjunction of two or more features.

3. In KBUILD, a feature always selects additional sources files for compilation. In no case the selection of a feature causes a source file to be removed from compilation process. This is a rather uncommon feature for MAKE based systems but more commonly found in systems that employ delta-oriented programming (DOP) [22].

As shown in Section 4, the current implementation produces presence implications for 95.4% of all source files in Linux on `arch-x86`. An investigation of the remaining 4.6% source files reveales that the majority of files violate assumption #2. The violation of this assumption is best explained with an example:

```
1  my-obj-$(CONFIG_FB_MATROX_G)  += matroxfb_crtc2.o
2  obj-$(CONFIG_FB_MATROX)       += $(my-obj-y)
```

Here the the file `matroxfb_crtc2.o` is only built if both features `FB_MATROX_G` and `FB_MATROX` are enabled at the same time. The helper function *features_in_dir* fails to detect that those two features have a connection. Therefore both features are tested independently and the build product `matroxfb_crtc2.o` does not show up in the output of *list*.

In the future, we intend to cover these cases by employing some simple heuristics (e.g., with data from the KCONFIG model) in the helper function *features_in_dir* to probe for more than a single configuration variable at the same time without increasing the number of necessary probing steps excessively. We expect this to improve the resulting logical constraints both the quantitatively and qualitatively even further.

Depending on how the extracted build-system constraints are employed, the higher runtime, compared to other approaches, might be a limitation of the approach. However many applications require the KBUILD constraints to be calculated exactly once and reuse

them in analyses that take much longer compared to the extraction process. This applies to both applications that have been presented in Section 5.

6.2 Generalizability

In contrast to the parsing-based approaches, which rely heavily the idiomatic style in which KBUILD makes use of the MAKE language, we avoid this dependency by treating the build system as a black box. Only two primitives, *list* and *features_in_dir*, have to be reimplemented for other build systems. This thin connection to the internal structures is the main reason for the robustness of the probing based approach with respect to the presented application on a wide range of Linux versions and architectures.

In order to show the portability of our approach, we have implemented the necessary adaptations for two further software projects: The build system of BUSYBOX [7], a toolbox of UNIX-tools for embedded systems, and the build system of FIASCO [12], a L4-like micro kernel. Both ports took less than 100 additional lines of code and were straight-forward to implement. We are convinced that the assumptions made on KBUILD in Section 6.1 also apply to other build systems.

6.3 Comparison of the Calculated Source File Constraints

For a qualitative evaluation of the extracted presence implications, we compare the output of our GOLEM tool to the results of Berger et al. [4] and Nadi and Holt [18]. For all the files that have a presence implication in our model, the presence implication from the other models is checked for semantic equivalence by using a SAT Checker.

$$\phi_{M_1}(f) \leftrightarrow \phi_{M_2}(f) \qquad f \in files(M_1) \cap files(M_2)$$

This equivalence check is done by instrumenting the SAT checker to prove that the bi-implication of the presence implications is a tautology and therefore have always the same implication. We use this check to compare the GOLEM model to the models of Nadi and Holt and Berger et al.

For the much smaller model of Nadi and Holt, 15 percent of the 7,082 common files have an equivalent presence implication and 81.9 percent have a presence implication that implies the GOLEM presence implication. We conclude that this model is mostly subsumed by the GOLEM model.

The comparison of the GOLEM model with the model from Berger et al. shows that out of 8,885 common files, 99.6 percent fulfill this bi-implication. This pratical equivalence shows that both tools are similarly mature.

7. RELATED WORK

The analysis of variability in Linux is a hot topic in the Software Engineering (SWE) and Software Product Line (SPL) community. Zengler and Küchlin [31] show an attempt to derive formal semantics of KCONFIG. She et al. [23] reverse-engineer the KCONFIG variability declaration in order to reconstruct a feature model. In [10] we have shown and quantified that the fine-grained variability implementation by CPP is dominated by a more coarse-grained management in KBUILD. We therefore think that KBUILD variability extractors, such as KBUILDMINER [3], the Nadi parser [18] or the GOLEM tool presented in this article, are a necessary complement for holistic variability analysis.

Berger et al. [6] investigate the configuration languages and tools KCONFIG and configuration description language (CDL). While the work shows that variability-management tools are employed successfully in open-source operating systems, it covers only the feature specification and modeling.

Adams et al. [2] demonstrate that analysis, visualization and in essence, re-engineering of the Linux build system is feasible. Their framework Makao [1] infers modularity in KBUILD by analyzing build traces. However, the amount of variation points that we identify in KBUILD with this article indicates that the full re-engineering of build-system variability remains an unsolved problem.

Kästner et al. [13] propose a technique coined "variability aware parsing", which essentially integrates the CPP variability into tools for variability aware type-checking. Mainly because of implementation challenges, TypeChef focuses on `arch-x86` and requires assistance in form of additional constraints by tools like KBUILDMINER [3]. Even with this, the approach is restricted to CPP based variability—the build-system–derived variability remains out of scope.

Palix et al. [19] try to reproduce a ten year old analysis on Linux by Chou et al. [8] in order to investigate the evolutionary development of Linux across the last decade. As the old experiment misses to state the exact configuration that was used, the environment could only be approximated. Hereby, the paper indirectly discusses CC in the sense that the selected configuration can (and does) affect the results of static analysis tools considerably. We take this anecdote as call for further integration of configuration consistency checks and CC into static analysis tools.

Inside the software verification community, Post and Sinz [21] introduce a technique coined "configuration lifting", which translates the variability expressed in KCONFIG, KBUILD and CPP into C source code. The generated C files encode the variability of the original source, the makefiles, and the feature model, and is verified with the CBMC tool by Clarke, Kroening, and Lerda [9]. While "configuration lifting" has similar goals, it remains unclear if that approach scales to the size of Linux.

8. SUMMARY AND CONCLUSION

To cope with a broad range of application and hardware settings, system software has to be highly configurable. Linux v3.2, as a prominent example, offers 11,000 configurable features. The implementation of this huge amount of static variability is implemented by `#ifdef`-blocks in the source code, but especially by the Linux make system. From the maintenance point of view, this imposes big challenges, as the feature model and the configurability that is *actually* implemented in the code have to be kept in sync. This calls for tool support.

A major hurdle for acceptance by the Linux developers is that such tools have to work reliably on the latest development version of Linux. Robustness against evolutionary changes in Linux, which includes both C code and the build system, is a strong requirement. In this paper, we have presented such a robust approach for extracting variability from the Linux build system that extracts logical constraints for 95.4% of all source files in Linux v3.2 on the `x86` architecture. Unlike existing approaches, our approach does not try to analyze the makefiles, but exploits the build system itself to infer the effects of selected features on the set of compiled files. Instead of manual and error-prone engineering that tailors the variability extractor to a specific version or architecture of Linux, our approach requires only two basic and straightforward to implement primitives. This thin interface to the build system allows a straight-forward to implement adaptation of the approach to other software projects, which has been demonstrated for BUSYBOX [7] and FIASCO [12].

9. ACKNOWLEDGMENTS

We whish to thank our anonymous reviewers for their helpful suggestions. Special thanks got to Sarah Nadi and Thorsten Berger for

providing access to their tools and data and their helpful comments on a draft of this paper.

This work was supported by the German Research Council (DFG) under grants no SCHR 603/7-2 and LO 1719/2-2.

References

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. “Design recovery and maintenance of build systems”. In: *Proceedings of the 23st IEEE International Conference on Software Maintainance (ICSM’07)*. IEEE Computer Society Press, 2007. DOI: 10.1109/ICSM.2007.4362624.

[2] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. “The Evolution of the Linux Build System”. In: *Electronic Communications of the EASST* (2007).

[3] Thorsten Berger and Steven She. *Google Code Project: various variability extraction and analysis tools*. URL: http://code.google.com/p/variability/ (visited on 02/16/2012).

[4] Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wasowski. *Feature-to-Code Mapping in Two Large Product Lines*. Technical report. University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[5] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. “Feature-to-code mapping in two large product lines”. In: *Proceedings of the 14th Software Product Line Conference (SPLC ’10)*. Volume 6287. Lecture Notes in Computer Science. Poster session. Springer-Verlag, 2010.

[6] Thorsten Berger, Steven She, Rafael Lotufo, and Andrzej Wasowski und Krzysztof Czarnecki. “Variability Modeling in the Real: A Perspective from the Operating Systems Domain”. In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE ’10)*. ACM Press, 2010. DOI: 10.1145/1858996.1859010.

[7] *BusyBox Project Homepage*. URL: http://www.busybox.net/ (visited on 05/11/2012).

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An empirical study of operating systems errors”. In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01)*. ACM Press, 2001. DOI: 10.1145/502034.502042.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Volume 2988. Lecture Notes in Computer Science. Springer-Verlag, 2004. DOI: 10.1007/978-3-540-24730-2_15.

[10] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Understanding Linux Feature Distribution”. In: *Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software (AOSD-MISS ’12)*. ACM Press, 2012. DOI: 10.1145/2162024.2162030.

[11] Kai Germaschewski and Sam Ravnborg. “Kernel configuration and building in Linux 2.5”. In: *Proceedings of the Linux Symposium*. 2003.

[12] Michael Hohmuth. *The Fiasco kernel: System architecture*. Technical report. TU Dresden, 1998.

[13] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. “Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation”. In: *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’11)*. ACM Press, 2011. DOI: 10.1145/2048066.2048128.

[14] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. “An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines”. In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE ’10)*. ACM Press, 2010. DOI: 10.1145/1806799.1806819.

[15] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. “SAT-based analysis of feature models is easy”. In: *Proceedings of the 13th Software Product Line Conference (SPLC ’09)*. Carnegie Mellon University, 2009.

[16] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. “Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis”. In: *Proceedings of the 15th IEEE Conference on Requirements Engineering (RE ’07)*. IEEE Computer Society, 2007. DOI: 10.1109/RE.2007.61.

[17] Sarah Nadi and Richard C. Holt. “Make it or Break it: Mining Anomalies from Linux Kbuild”. In: *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE ’11)*. 2011. DOI: 10.1109/WCRE.2011.46.

[18] Sarah Nadi and Richard C. Holt. “Mining Kbuild to Detect Variability Anomalies in Linux”. In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR ’12)*. To appear. IEEE Computer Society Press, 2012.

[19] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. “Faults in Linux: Ten years later”. In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’11)*. ACM Press, 2011. DOI: 10.1145/1950365.1950401.

[20] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[21] Hendrik Post and Carsten Sinz. “Configuration Lifting: Verification meets Software Configuration”. In: *Proceedings of the 23th IEEE/ACM International Conference on Automated Software Engineering (ASE ’08)*. IEEE Computer Society, 2008. DOI: 10.1109/ASE.2008.45.

[22] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. “Delta-oriented programming of software product lines”. In: *Proceedings of the 14th Software Product Line Conference (SPLC ’10)*. Volume 6287. Lecture Notes in Computer Science. Springer-Verlag, 2010. DOI: 10.1007/978-3-642-15579-6_6.

[23] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. “Reverse Engineering Feature Models”. In: *Proceedings of the 33nd International Conference on Software Engineering (ICSE ’11)*. ACM Press, 2011. DOI: 10.1145/1985793.1985856.

[24] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “Efficient Extraction and Analysis of Preprocessor-Based Variability”. In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE ’10)*. ACM Press, 2010. DOI: 10.1145/1868294.1868300.

[25] Diomidis Spinellis. “A Tale of Four Kernels”. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE ’08)*. ACM Press, 2008. DOI: 10.1145/1368088.1368140.

[26] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU make manual. A Program for Directing Recompilation*. Free Software Foundation. GNU Press, 2010.

[27] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. “A Taxonomy of Variability Realization Techniques”. In: *Software - Practice and Experience* 35.8 (2006). DOI: 10.1002/spe.v35:8.

[28] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. “Configuration Coverage in the Analysis of Large-Scale System Software”. In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS ’11)*. ACM Press, 2011. DOI: 10.1145/2039239.2039242.

[29] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys ’11)*. ACM Press, 2011. DOI: 10.1145/1966445.1966451.

[30] *VAMOS - Variability Management in Operating Systems*. FAU Erlangen-Nuremberg, 2012. URL: http://www4.informatik.uni-erlangen.de/Research/VAMOS/.

[31] Christoph Zengler and Wolfgang Küchlin. “Encoding the Linux Kernel Configuration in Propositional Logic”. In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*. 2010.

Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability

Reinhard Tartler[1], Anil Kurmus[2],
Bernhard Heinloth[1], Valentin Rothberg[1], Andreas Ruprecht[1], Daniela Dorneanu[2],
Rüdiger Kapitza[3], Wolfgang Schröder-Preikschat[1], and Daniel Lohmann[1]

[1]*Friedrich-Alexander University Erlangen-Nürnberg*
[2]*IBM Research - Zurich*
[3]*TU Braunschweig*

Abstract

The Linux kernel can be a threat to the dependability of systems because of its sheer size. A simple approach to produce smaller kernels is to manually configure the Linux kernel. However, the more than 11,000 configuration options available in recent Linux versions render this a demanding task. We report on designing and implementing the first automated generation of a workload-tailored kernel configuration and discuss the security gains such an approach offers in terms of reduction of the Trusted Computing Base (TCB) size. Our results show that the approach prevents the inclusion of 10% of functions known to be vulnerable in the past.

1 Introduction

The Linux kernel is a commonly attacked target. In 2011, 148 Common Vulnerabilities and Exposures (CVE)[1] entries for Linux have been recoded, and this number is expected to grow every year. This is a serious problem for system administrators who rely on a distribution-maintained kernel for the daily operation of their systems. On the Linux distributor side, kernel maintainers can make only very few assumptions on the kernel configuration for their users: Without a specific use case, the only option is to enable every available configuration option to maximize the functionality. The ever-growing kernel code size, caused by the addition of new features, such as drivers, file systems and so on, indicates that the risk of undetected vulnerabilities will constantly increase in the foreseeable future.

If the intended use of a system is known at kernel compilation time, an effective approach to reduce the kernel's attack surface is to configure the kernel to not compile unneeded functionality. However, finding a fitting configuration requires extensive technical expertise about currently more than 11,000 Linux configuration options, and needs to be repeated at each kernel update. Therefore, maintaining such a custom-configured kernel entails considerable maintenance and engineering costs.

This paper presents a tool-assisted approach to automatically determine a kernel configuration that enables only kernel functionalities that are actually necessary in a given scenario. We quantify the security gains in terms of reduction of the Trusted Computing Base (TCB) size. The evaluation section (Section 3) focuses on an appliance-like virtual machine that runs a web server similar to those used to power large distributed web services in the cloud. Our approach exhibits promising security improvements for this use case: Compared with a default distribution kernel, 10% of the kernel functions (i.e., 17 out of 179), for which in total 31 vulnerabilities have been reported, are removed from the tailored kernel.

The remainder of this paper is structured as follows: Section 2 presents the design and implementation of the first automated workload-specific kernel-build generation tool. Section 3 evaluates the usability of such an approach in a real-world scenario. Security benefits of the tailored Linux kernel are discussed in Section 4. Section 5 presents the related work. The paper concludes in Section 6.

2 Kernel-Configuration Tailoring

The goal of our approach is to compile a Linux kernel with a configuration that has only those features enabled which are necessary for a given use case. This section shows the fundamental steps of our approach to tailor such a kernel. The six necessary steps are depicted in Figure 1.

❶ **Enable tracing.** The first step is to prepare the kernel so that it records which parts of the kernel code are executed at run time. We use the Linux-provided `ftrace` feature, which is enabled with the KCONFIG configuration option `CONFIG_FTRACE`. Enabling this configuration option modifies the Linux build process to include profiling

[1] `http://cve.mitre.org/`

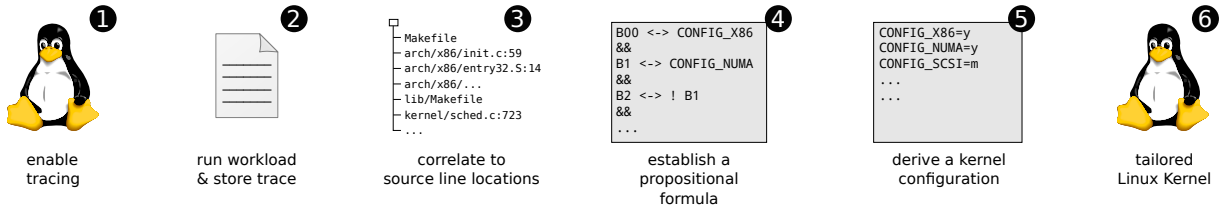


Figure 1: Workflow of the approach

code that can be evaluated at runtime.

In addition, our approach requires a kernel built with debugging information so that any function addresses in the code segment can be correlated to functions and thus source file locations in the source code. For Linux, this is configured with the KCONFIG configuration option `CONFIG_DEBUG_INFO`.

❷ **Run workload.** In this step, the system administrator runs the targeted application after enabling `ftrace`. The `ftrace` feature now records all addresses in the text segment that have been instrumented. For Linux, this covers most code, except for a small amount of critical code such as interrupt handling, context switches and the tracing feature itself.

To avoid overloading the system with often accessed kernel functions, `ftrace`'s own ignore list is dynamically being filled with functions when they are used. This prevents such functions from appearing more than once in the output file of `ftrace`. We use a small wrapper script for `ftrace` to set the correct configuration before starting the trace, as well as to add functions to the ignore list while tracing and to parse the output file, printing only addresses that have not yet been encountered.

❸ **Correlation to source lines.** A system service translates the raw address offsets to source line locations using the ADDR2LINE tool from the `binutils` tool suite. This identifies the source files and the `#ifdef` blocks that are actually being executed during the tracing phase. Technically, the tool stores its result to a text file with source-file names and line numbers on each line.

❹ **Establishment of the propositional formula.** This step translates the source-file locations into a propositional formula. The propositional variables of this formula are the *variation points* the Linux configuration tool KCONFIG controls during the compilation process. This means that every C Preprocessor (CPP) block, KCONFIG item and source file can appear as a propositional variable in the resulting formula. This formula is constructed with the variability constraints that have been extracted from `#ifdef` blocks, KCONFIG feature descriptions and Linux Makefiles. The extractors we use have been developed, described and evaluated in previous work [5, 21, 22]. The resulting formula holds for every KCONFIG configuration that enables all source lines simultaneously.

❺ **Derivation of a tailored kernel configuration.** A SAT checker proves the satisfiability of this formula and returns one concrete configuration that fulfills all these constraints. Note that finding an optimal solution to this problem is an NP-hard problem and was not the focus of our work. Instead, we rely on heuristics and configurable search strategies in the SAT checker to obtain a sufficiently small configuration.

As the resulting kernel configuration will contain some additional unwanted code, such as the tracing functionality itself, whitelists and blacklists are employed, allowing the user to specify additional constraints in order to force the selection (or deselection) of certain KCONFIG features. This results in additional constraints being conjugated to the formula just before invoking the SAT checker.

❻ **Compiling the kernel.** The resulting solution to the propositional formula, obtained as described above, can only cover KCONFIG features of code that has been traced. As the KCONFIG feature descriptions declare non-trivial dependency constraints [25], special care must be taken to ensure that as many KCONFIG features as possible are not selected while still fulfilling all dependency constraints. We therefore use the KCONFIG tool itself to process this feature selection to a KCONFIG configuration that is both consistent and selects as few features as possible.

3 Practical Application

We evaluate the usefulness of our approach by setting up a Linux, Apache, MySQL and PHP (LAMP)-based web presence in a manner that is suited for deployment in a cloud environment. The system serves static webpages, the collaboration platform DOKUWIKI [7] and the message board system PHPBB3 [19] as an example for typical real-world applications. We use the distribution-provided packages from the Debian distribution without further specific configuration changes or optimization. Evaluation results are summarized in Table 1.

3.1 Kernel Tailoring

To derive a minimized kernel configuration, the first step consists of compiling a tracing-enabled Linux ker-

nel. We use the standard Linux kernel source and configuration from the Debian distribution (version `2.6.32-41squeeze2`) as a template for our tracing kernel (Step ❶ in Figure 1). On this kernel, we enable the features `CONFIG_FTRACE` and `CONFIG_DEBUG_INFO` to include the `ftrace` tracing infrastructure and compile with debugging symbols. As our current prototype is not able to resolve functions from loadable kernel modules (LKMs) yet, we disable module support in the kernel configuration, which causes all compiled code to be loaded into the system at boot time.

Furthermore, a number of drivers cause compilation and linking errors when not compiled as LKMs. Most of these issues stem from drivers in the `staging`[2] area. Also, when trying to boot this kernel, we observe kernel panics during the initialization of a range of watchdog drivers. As these drivers turn out to be unnecessary for this application scenario, we turn off the KCONFIG options `CONFIG_STAGING` and `CONFIG_WATCHDOG`. These configuration changes account for the difference in size and features between the kernel shipped with Debian (∼42 MB of code in the `text` segment) and the intermediary kernel that is used for collecting traces (∼36 MB of code in the `text` segment).

With this intermediary tracing kernel, the system is tested against a test workload that covers all required functionality. We use the Skipfish [24] security analysis tool to systematically access all functionality of the appliance in an automated manner. This corresponds to Step ❷ in Figure 1 and results in a total of 5,377 observed kernel functions.

These traced kernel functions correlate to 4,686 different source lines in 379 source files (Step ❸). We use a modified version of the UNDERTAKER tool [22] to establish the propositional formula (Step ❹) and to derive a solution for it (Step ❺). To avoid unwanted functionality enabled in the resulting kernel, such as the `ftrace` infrastructure itself and LKM support, the UNDERTAKER tool obeys a blacklist that consists of the KCONFIG options `CONFIG_FTRACE` and `CONFIG_MODULES`. Also, we add eight additional,[3] use-case–agnostic KCONFIG items to the whitelist in Step ❺ to enable features that are used by the initialization startup scripts, which run before the system-wide tracing process starts. These steps take 69 sec on a commodity 2.8 GHz quad-core workstation with 4 GB of RAM.

[2]The `staging` area contains unfinished and incomplete drivers that are included as a technology preview.

[3]Specifically: `CONFIG_ACPI`, `CONFIG_UNIX`, `CONFIG_DEVTMPFS`, `CONFIG_DEVTMPFS_MOUNT`, `CONFIG_SERIAL_8250_CONSOLE` and `CONFIG_INOTIFY_USER`, `CONFIG_PM`

Kernel Shipped by Debian	
Loaded Code	5,465,602 Bytes
Total Loadable Code	42,188,538 Bytes
Loaded Kernel Modules	29
KCONFIG options set to `y`	1,093
KCONFIG options set to `m`	2,299
Functions with CVE entries	179
Intermediary kernel used for tracing	
Loaded Code	36,341,888 Bytes
Total Loadable Code	36,341,888 Bytes
Loaded Kernel Modules	0
KCONFIG options set to `y`	3,298
KCONFIG options set to `m`	0
Functions with CVE entries	207
Resulting application-tailored kernel	
Loaded Code	3,990,153 Bytes
Total Loadable Code	3,990,153 Bytes
Loaded Kernel Modules	0
KCONFIG options set to `y`	379
KCONFIG options set to `m`	0
Functions with CVE entries	162

Table 1: Results of the experiment at a glance. The code sizes were obtained with the SIZE tool from the BINUTILS suite by adding the sizes of the text segments of the bootable kernel image and all loadable `.ko` files.

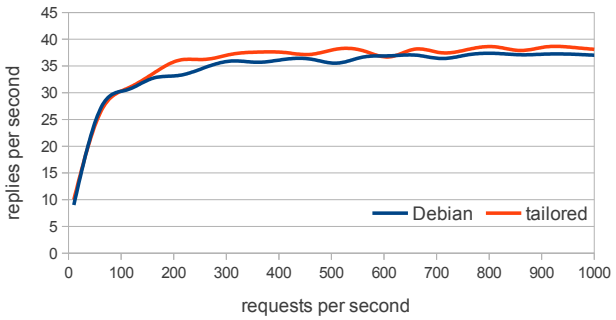


Figure 2: Comparison of reply rates of the web server with the tailored kernel and the standard distribution kernel.

3.2 Evaluation

To ensure the functionality of the appliance, we run the Skipfish [24] security scan again on the system with the tailored kernel, and compare the results with the previous run on the tracing kernel. The comparison of these two reports indicates no differences in the number of vulnerabilities or other issues.

The performance is tested with the httperf tool [18]. The tool accesses a static website continuously, at a constant number of requests per second in each run. We did two setups of the same test scenario, both times using the same system, but once booted with the Debian standard kernel, and once with our tailored kernel. The data shows that our tailored kernel achieves a performance very similar to that of the original kernel provided by the distribution.

4 Discussion

After the presentation of a practical use case for our approach, this section now evaluates the security benefits. For this, we present an applicable security model to determine the TCB, and discuss security improvements in terms of TCB reduction.

Security Model. In the context of the web service presented, we assume both local and remote malicious attackers that target the kernel. However, we do not consider attackers that have physical access to the machine nor attacks that directly target hardware and firmware vulnerabilities.

The security goal is to prevent an attacker from gaining full control with arbitrary code execution in kernel mode, information leakage (e.g., recover uninitialized kernel memory content) to breach confidentiality, and denial-of-service attacks by crashing the kernel to reduce the availability of the system.

TCB sizes. Following the literature [13], we define the TCB as "the subset of components that need to be trusted to fulfill the security goals given in the security model". Therefore, in the security model above, the TCB is solely composed of the kernel, including all LKM loaded during normal operation.

We apply three different metrics to measure the TCB reduction: a) the compiled code (text segment) size of the kernel, b) the total number of features that are enabled in KCONFIG and c) the number of functions compiled into the kernel for which there has been an CVE entry in the past 10 years. More precisely, through a semi-automated process, we map a subset of 197 out of 873 CVE entries to vulnerabilities in 215 unique functions in the kernel, and use this dataset. The results for all three kernels used in the experiment in Section 3 are shown in Table 1.

Results. The data shows that the Linux kernel shipped by Debian loads 5.5 MB of program code into the memory for the virtual machine in the scenario described in Section 3. Compared with the code size of 4 MB for our tailored kernel, the total TCB size is reduced by 27%.

The number of features enabled is also reduced significantly, from 3,392 (with 1,093 features compiled statically into the kernel and 2,299 as LKM) to 379. The omission of functionality to load further LKMs constitutes an additional security benefit.

Finally, for each function in the TCB, we record the number of known vulnerabilities that have been reported in the past 10 years. When comparing the default distribution kernel to the tailored kernel, we observe a reduction of 10% of functions for which vulnerabilities have been reported in the past. However, this number is a lower-bound estimate, as the Linux kernel supports on-demand insertion of LKM, resulting in a higher initial TCB size, and therefore higher TCB reduction.

Sampling bias. Compared with the code size reduction results above, the CVE reduction numbers may seem lower than expected. We hypothesize that this impression can be attributed to sampling bias: code that is used more often is also audited more often, and better care is taken in documenting the vulnerabilities of such functions. A comparison of the average number of CVEs in kernel functions that are loaded and used (9.8‰) with the average number of CVEs in kernel functions that are not used (3.7‰) supports this hypothesis. Previous studies [3] have also shown that code in the `drivers/` sub-directory of the kernel, which is known to contain a significant number of rarely-used code, on average contains significantly more bugs than any other parts of the kernel tree. Consequently, it is likely that unused features provided by the kernel still contain a significant amount of relatively easy-to-find vulnerabilities. This further confirms the importance of reducing the TCB size as presented in this work.

Unexpected impacts. The presented approach in this work could in turn cause a reduction of the security of the system – a drawback that is common to many security software but is often overlooked. Reviewing the process described in Section 2 (Step ❻), we cannot rule out that for some application scenarios, performance-critical or security features might be removed from the base kernel. Possible reasons for this include that a) the feature was not triggered during the system-wide trace, b) the functionality has been excluded from the instrumentation with `ftrace` (e.g., for performance reasons), or c) the configuration options influence the resulting kernel in non-functional ways (e.g., different compilation flags, etc.). Although we were not able to find any results confirming this in this experiment — for example, we have verified that the `CONFIG_CC_STACKPROTECTOR` configuration option, which toggles the inclusion of the GCC flag for adding a stack frame canary, remains enabled, in future work we intend to further evaluate potential adverse impacts.

Applicability and Incomplete Traces. The presented approach relies on the assumption that the use-case of the system is clearly defined. Thanks to this a priori knowledge, it is possible to determine what kernel functionalities the application requires and therefore, what kernel configuration options have to be enabled. Still, the lack of guarantees that a trace of a given scenario is sufficient

and captures all required functionality may raise concerns. However, this does not invalidate the approach. Our approach works best for service providers that instantiate fairly homogeneous services, for which the the chance to capture all necessary functionality is highest. Unfortunately, even the best tracing mechanism could still not guarantee catching all possible cases. Such a guarantee could only be provided with formal analysis and verification – a method that stands mathematical proof. However, for many scenarios organizations and system operators are likely to accept less costly approaches – such as the one presented in this paper. With the increasing importance of compute clouds, where customers employ virtual machines for very dedicated services such as the web server presented in Section 3, we expect that our approach can be easily applied to further use cases that are commonly deployed in the cloud.

Fortunately, incomplete traces do not lead to system crashes. With KCONFIG, changing the static configuration of the Linux kernel statically restricts the provided functionality to applications that interact with the kernel via well-defined interfaces. These interfaces, and the careful maintenance of kernel developers, allows applications to check for the presence of kernel features that are required for correct operation. In the absence of kernel bugs, applications must not be able to crash the kernel. On the other hand, applications that crash because of missing kernel features fail to provide proper error handling – which alone makes them unsuitable for security sensitive environments. In summary, crashes that occur because of incomplete traces always stem from easy-to-address software bugs for the scenario presented in this paper.

Usability. Most of the steps presented in Section 2 require no domain specific knowledge of Linux internals. We therefore expect that they can be conducted in a straightforward manner by system administrators without specific experience in Linux kernel development. The system administrator, however, continues to use a code base that continuously receives maintenance in form of bug fixes and security updates from the Linux distributor. We therefore are confident that our approach to automatically tailor a kernel configuration for specific use-cases is both practical and feasible to implement in real-world scenarios.

Extensibility. The experiment in Section 3 shows that the resulting kernel requires eight additional KCONFIG options for proper operation. Alternatively to adding these features to the whitelist with distribution-specific knowledge, starting the application tracer at the start of the boot process would also capture the missing functionality. However, in this way we demonstrate the ability to specify wanted or unwanted KCONFIG options independently of the tracing. This allows our approach to be assisted in the future by methods to determine kernel features that tracers such as `ftrace` cannot observe at all.

5 Related work

As we show below, this work relates to two research areas.

Kernel specialization. Several researchers have suggested approaches to tailor the Linux kernel, although security is usually not a goal, but improvements in code size or execution speed are: Lee et al. [14] manually modify the source code (e.g., by removing unnecessary system calls) based on a static analysis of the applications and the kernel. Chanet et al. [2], in contrast, propose a method based on link-time binary rewriting, but also employ static analysis techniques to infer and specialize the set of system calls to be used. Both approaches, however, do not leverage any of the built-in configurability of Linux to reduce unneeded code. Moreover, our approach is completely automated.

TCB reduction has always been a major design goal for micro-kernels [1, 15], which in turn facilitates a formal verification of the kernel [10] or its implementation in type-lafe languages, such as OCaml [16].

Kernel attack surface reduction. The security model used in this paper is commonly used when building *sandboxing* or *isolation* solutions, in which each process must be contained within a particular security domain, such as [4, 9, 17], which are all based on the Linux Security Module (LSM) framework [23]. The idea of directly restricting the system call interface on a per-process basis has been previously explored as well, e.g., by Fraser, Badger, and Feldman [6], Ko et al. [11], or Provos [20], although not with specific focus on reducing the kernel's attack surface. Seccomp [8] directly tackles this issue by allowing processes to be sandboxed at the system call interface. Ktrim [12] is a current research project which goes beyond simply limiting the system call interface, and explores the possibility of finer-granularity kernel attack surface reduction by restricting individual functions (or sets of functions) inside the kernel. In contrast, this work focuses on compile-time removal of functionality from the kernel at a system-wide level instead of a runtime removal at a per-application level.

6 Conclusion and Future Work

This paper presents an approach for automatically tailoring a Linux kernel configuration to a given use case. The result is a Linux kernel in which unnecessary functionality is removed at compile-time, hence significantly reducing

TCB size. The reduction can be quantified with 27% less code loaded and at least 10% fewer kernel functions which were previously vulnerable to attacks.

While the current prototype shows promising results, we intend to improve on the usability and applicability to additional use-cases. For instance, the current prototype unconditionally disables module loading support. As this may be undesirable in some cases, we intend to improve the handling of LKMs, as well as to remove the the need for an intermediary tracing kernel.

Acknowledgments

This research has been partially supported by the TClouds project[a] funded by the European Union’s Seventh Framework Programme (FP7/2007-2013) under grant agreement number ICT-257243.

[a] http://www.tclouds-project.eu

References

[1] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. “MACH: A New Kernel Foundation for UNIX Development”. In: *USENIX Summer Conference*. USENIX, 1986.

[2] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. “System-wide Compaction and Specialization of the Linux Kernel”. In: *2005 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES ’05)*. ACM, 2005. DOI: 10.1145/1065910.1065925.

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An empirical study of operating systems errors”. In: *18th ACM Symp. on OS Principles (SOSP ’01)*. ACM, 2001. DOI: 10.1145/502034.502042.

[4] Kees Cook. *Yama LSM*. 2010. URL: http://lwn.net/Articles/393012/ (visited on 06/04/2012).

[5] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “A Robust Approach for Variability Extraction from the Linux Build System”. In: *16th Software Product Line Conf. (SPLC ’12)*. (To appear). ACM, 2012.

[6] Timothy Fraser, Lee Badger, and Mark Feldman. “Hardening COTS software with generic software wrappers”. In: *20th Symp. on Security and Privacy*. IEEE, 1999. DOI: 10.1109/SECPRI.1999.766713.

[7] Andreas Gohr. *DokuWiki*. URL: http://dokuwiki.org (visited on 06/03/2012).

[8] *Google Seccomp Sandbox for Linux*. URL: http://code.google.com/p/seccompsandbox/wiki/overview (visited on 06/05/2012).

[9] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. “Task Oriented Management Obviates Your Onus on Linux”. In: *Japan Linux Conference* (2004).

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: formal verification of an OS kernel”. In: *22nd ACM Symp. on OS Principles (SOSP ’09)*. ACM, 2009. DOI: 10.1145/1629575.1629596.

[11] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. “Detecting and countering system intrusions using software wrappers”. In: *9th Conf. on USENIX Security Symposium (SSYM ’00)*. USENIX, 2000.

[12] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. “Attack surface reduction for commodity OS kernels: trimmed garden plants may attract less bugs”. In: *4th Eur. W’shop on system security (EUROSEC ’11)*. ACM, 2011. DOI: 10.1145/1972551.1972557.

[13] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. “Authentication in distributed systems: theory and practice”. In: *ACM Trans. Comp. Syst.* 10.4 (1992). DOI: 10.1145/138873.138874.

[14] C.T. Lee, J.M. Lin, Z.W. Hong, and W.T. Lee. “An Application-Oriented Linux Kernel Customization for Embedded Systems”. In: *Journal of information science and engineering* 20.6 (2004).

[15] Jochen Liedtke. “On μ-Kernel Construction”. In: *15th ACM Symp. on OS Principles (SOSP ’95)*. ACM OSR. ACM, 1995. DOI: 10.1145/224057.224075.

[16] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft. “Turning Down the LAMP: Software Specialisation for the Cloud”. In: *2nd USENIX Conf. on hot topics in cloud computing (HOTCLOUD’10)*. USENIX, 2010.

[17] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux By Example: Using Security Enhanced Linux*. Prentice Hall, 2006.

[18] David Mosberger and Tai Jin. “httperf. A tool for measuring web server performance”. In: *SIGMETRICS Performance Evaluation Review* 26.3 (1998). DOI: 10.1145/306225.306235.

[19] *phpBB. Free and Open Source Forum Software*. URL: www.phpbb.com (visited on 06/03/2012).

[20] Niels Provos. “Improving host security with system call policies”. In: *12th Conf. on USENIX Security Symposium (SSYM ’03)*. Vol. 12. USENIX, 2003.

[21] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “Efficient Extraction and Analysis of Preprocessor-Based Variability”. In: *9th Int. Conf. on Generative Programming and Component Engineering (GPCE ’10)*. ACM, 2010. DOI: 10.1145/1868294.1868300.

[22] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2011 (EuroSys ’11)*. ACM, 2011. DOI: 10.1145/1966445.1966451.

[23] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. “Linux Security Module Framework”. In: *Ottawa Linux Symposium*. 2002.

[24] Michal Zalewski, Niels Heinen, and Sebastian Roschke. *skipfish. Web application security scanner*. URL: http://code.google.com/p/skipfish/ (visited on 06/03/2012).

[25] Christoph Zengler and Wolfgang Küchlin. “Encoding the Linux Kernel Configuration in Propositional Logic”. In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*. 2010.