Static and Dynamic Weaving in System Software with AspectC++

Wolfgang Schröder-Preikschat, Daniel Lohmann, Fabian Scheler, Wasif Gilani, Olaf Spinczyk
Friedrich-Alexander University Erlangen-Nuremberg
{wosch,lohmann,scheler,gilani,spincyk}@cs.fau.de

***Abstract*— System software strongly relies on the availability of static as well as dynamic adaptation techniques. With Aspect-Oriented Programming (AOP) it is now possible to adapt even policy-like crosscutting concerns in the implementation of system software. While this is straightforward in the static case, dynamic adaptation of crosscutting concerns requires an expensive dynamic aspect weaving infrastructure. Furthermore, the relation between static and dynamic aspects is widely unexplored.**

In this paper we present our experiences with static and dynamic adaptation of crosscutting concerns in the embedded operating system eCos. The work is based on the novel "single language approach", which allows us to configure the binding time of aspects, and a "family-based dynamic weaver infrastructure", which reduces the resource consumption needed for dynamic AOP by tailoring the run time system. In our prototype implementation all this has been integrated into an environment, which allows us to "play" with the binding time of aspects and supported dynamic weaver features. Thus, we can now answer questions about the resource consumption of these adaptation techniques and the relationship of static and dynamic aspects in general.

I. Introduction

System software, especially in the domain of resource constrained embedded systems, strongly relies on the availability of static adaptation techniques. An application- and platform-specific tailoring of the system is usually performed before it is compiled during a special configuration step. Besides static adaptation at compile-time, dynamic adaptation of the running system is often an important requirement too, as stopping and re-starting system software means that all applications would have to be stopped as well. Therefore, modern operating systems in the workstation domain support dynamically loadable modules.

A. System Software Product Lines

It is simply impossible to build a one-fits-all system that fulfills the requirements of all potential applications, while still being thrifty and economical with system resources. The solution is therefore to tailor down the operating system so that it provides exactly the functionality required by the intended application, but nothing more. This leads to a family-based or product-line approach, where the variability and commonality among OS family members is expressed by feature models [8]. Special tools are used to extract and statically configure the concrete operating system based on an application-specific feature selection [4]. As an example, the well known eCos system [1], which we used for the case study presented in this paper (see section VII), can be regarded as a statically configurable operating system product line.

The overall quality of an OS product-line depends mostly on the offered levels of variability and granularity. A crucial point is the mapping of all selectable and configurable features to their corresponding, well encapsulated implementation components. Fundamental system policies, like synchronization or activation points for the scheduler, are typically reflected in many points of the OS component code. This crosscutting character makes it almost impossible to implement them as independent encapsulated entities and thereby restricts variability and granularity. Aspect-oriented programming (AOP) has proven to be a promising way to deal with crosscutting concerns in OS code [7], [23], [10]. It allows encapsulating the implementations of crosscutting concerns in entities called *aspects*, which are then woven into the OS component code (e.g. classes) at build time. A well-directed application of AOP principles in the development of OS product lines can therefore lead to a higher variability and granularity of the selectable OS features, as their implementations can not only be encapsulated by classes, but also by aspects. This potentially results in very flexible systems that offer configurability of even fundamental architectural properties [17], [18].

Static adaptation of system software for a specific application profile or specific hardware works well in many domains. However, in the emerging domain of smart devices (like mobile phones, personal digital assistants or wearables), the set of executed applications as well as the non-functional requirements to the operating systems do vary. Manufacturers are responding to this challenge by enlarging their devices by more system resources and large operating systems that implement many features, but are less reusable and scalable. This is unsatisfactory, as it noticeably increases production costs, weight and power consumption of mobile devices. We therefore advocate adaptable operating systems, which provide a well-balanced way of adaptability, while still being based on application-specific tailoring. Static and dynamic aspect weaving are crucial techniques for the design and implementation of these novel system software product lines.

B. Outline

The outline of this paper is as follows: we start with a motivation by describing the vision of adaptable aspect-oriented operating system product lines in the next section. This is followed by a discussion of related work. The sections

IV and V describe two novel concepts which we consider crucial for the implementation of the vision, namely the "Single Language Approach" and the "Family-Based Dynamic Weaving Infrastructure". In section VI, implementation issues of these concepts are discussed. Section VII presents a case study which was conducted with the embedded operating system product line eCos. The final section will summarize our conclusions.

II. Motivation: Adaptable Operating systems

A. *Static Weaving for Static Adaptation*

The set of requirements (services and non-functional properties) that has to be fulfilled by a tailored OS depends on the requirements defined by the (potentially) executed applications as well as on the requirements defined by global user policies, like the energy or security mode. The requirements set leads to a feature selection, which corresponds to a specific member of the operating system family. The feature selection is fed into generators, which configure and build the final product. The static aspects are superimposed onto the primary functionality in an additive manner without altering the existing architecture. These aspects cannot be removed or reconfigured later during runtime. The result of static adaptation is an application-specific product, which contains only those features that are needed by the application.

B. *Dynamic Weaving for Dynamic Adaptation*

Once the system starts running, it may be subject to the changing requirements during runtime. Several approaches have been applied in the past to achieve dynamic adaptation and reconfiguration of the software systems during runtime. Some try to provide adaptability by using patterns in several features [22]. However, the customization resulting from this approach is still unsatisfactory as it leaves hooks in the core code and null strategies substitute the excluded features [12]. This adds to the complexity of code as well as to the memory footprint. Other approaches suggest the use of reflection and component frameworks [15], [14]. In some of these approaches, the system implementation adapts itself according to the changed environment by means of selecting different implementation strategies. These approaches mainly address the customizability and adaptability aspect of the systems. The drawback of these techniques is that they have rather large memory requirements and also incur performance overhead.

Dynamic weaving is a natural choice for implementing an adaptable system due to the reason that it can apply code retrospectively to a running application. The dynamic adaptation of complex software systems is generally dependent on policies, which all tend to be crosscutting concerns, and, hence are realized as dynamic aspects.

Reconfiguration of an adaptable operating system takes place when the set of requirements changes. This happens, if a new application is about to be executed with some requirements that are currently not offered by the system, or if a global policy, such as switching to low-power mode, is applied.

To our understanding, all of these configurations are still members of one family of operating systems, where each configuration (feature selection) leads to one distinct family member. The process of adaptation can be understood as morphing from one feature selection into another. By adaptation to the closest set of the demanded features, the system is always tailored with respect to the actually executed applications.

C. *Flexible Feature Binding Times*

It should be configurable at compile time whether certain operating system features may be selected or deselected at runtime. The general idea behind this is that static features can be implemented more efficiently than dynamically changeable features and should always be preferred. If a feature has a crosscutting nature it will be implemented by an aspect. Depending on the system configuration an aspect could be static or dynamic. Furthermore, static and dynamic aspects have to coexist in the system. Solutions that only support dynamic weaving are not acceptable due to their low efficiency.

D. *Minimal Overhead*

The motivation for our work on dynamic adaptation is to provide a better, i.e. more resource efficient, OS support for applications in a dynamically changing environment than it could be provided by a one-size-fits-all solution. If the necessary infrastructure for dynamic weaving costs more than we can save, the approach fails.

A facility for on-demand adaptation needs to be provided by the OS itself in some way. The main question is, how to rebind features at runtime (in particular their implementation classes and/or aspects), if the set of required features changes?

For classes and libraries this task can be done by a dynamic loader/linker, which loads the component and performs all necessary steps to bind it, like relocation and component registration. Such a dynamic library loader is already present in commodity operating systems such as Windows, Linux or Solaris.

However, as our product line follows an AOP-based approach, features may also be implemented by aspects. For dynamic loading/unloading of aspects, the system has to provide facilities for dynamic weaving. Besides supporting flexible binding times, a minimal overhead dynamic aspect weaver is the second big challenge of the described vision.

III. Related Work

Many different approaches have been proposed by the AOSD community for dynamic weaving. Most of them target the Java domain. Much fewer have been suggested for C or C++. As the performance and memory overhead of Java is not feasible for the domain of embedded operating systems, we summarize the Java-based approaches only briefly, but focus on the C/C++ based approaches.

TABLE I
LANGUAGE FEATURES AVAILABLE IN SEVERAL DYNAMIC WEAVING APPROACHES FOR JAVA (ASPECTWERKZ, STREAMLOOM), JAVABEANS (JASCO), C (TOSKANA, ARACHNE, TINYC^2) AND C++ (DAO C++)

Dynamic Weavers		Java-based				C/C++ based			
category	feature	AspectWerkz	SteamLoom	Jasco	...	Toskana	Arachne	TinyC^2	DAO C++
join point types	call	√	-	√		-	√	-	-
	execution	√	√	√		√	√	√	√
	set/get	√	-	√		√	√	-	-
	cflow	√	√	√		-	√	-	-
advice types	before	√	√	√		√	-	√	√
	after	√	√	√		√	-	√	√
	around	√	-	√		√	√	-	-
intro-ductions	attributes	-	-	-		-	-	-	-
	functions	√	-	-		-	-	-	-
interaction	ordering	√	?	√		-	-	-	-
context	args, ...	√	-	√		-	-	-	-

A. *Dynamic Weaving Approaches for Java*

Several approaches have been suggested for dynamic weaving in Java [19], [20], [3], [6], [5], [21]. These approaches are based on Java-specific APIs, JVM debugging interface, static byte code instrumentation, runtime byte code manipulation and virtual machine extensions. Overall, dynamic weaving in Java programs seems to induce significant costs. In a recent paper, HAUPT and MEZINI compared several dynamic weavers and observed a performance loss factor in the range of 10 to 10,000 [13]. There seems to be a relationship between overhead and supported AOP features, as the dynamic weaver which performed best in their study (*SteamLoom* [5]) provides the smallest set of AOP features (Table I).

B. *Dynamic Weaving Approaches for C/C++*

Most approaches to support dynamic weaving in C are based on runtime binary code manipulation. *Arachne* [9], *TOSKANA* [10] and *TinyC*^{2} [26] rewrite the binary code at runtime to weave and unweave aspects. The actual weaving positions in the binary code are examined with the help of symbol and/or debug information, generated by the C compiler during compilation of the targets.

Due to the principle of binary code manipulation, the performance overhead of these weavers is significantly lower than for the Java-based systems [10], [9]. The offered AOP features are, on the other hand, also limited (Table I). The most serious limitation is, however, their platform-dependence, as especially in the domain of embedded systems a broad variety of CPU and hardware platforms is used. The available dynamic weaver implementations are not only limited to a specific processor platform, but also to a specific compiler, as the weaving process depends on symbol/debug information and specific code patterns generated by the compiler for potential joinpoints such as function calls. State-of-the-art optimization techniques, such as code inlining or stripping of symbol information, have to be disabled. While this may be considered acceptable for C, most C++ compilers implicitly perform such optimizations.

We know only one approach which is platform-independent and targeting the C++ domain. In *DAO C++* [2], aspects are woven by registering them against a runtime registration system. The original C++ code has to be instrumented, either by hand or with the help of tools, to call the runtime system at each potential joinpoint.

C. *Dynamic Weaving in Operating Systems*

Most existing work on using AOP in operating systems is focused on static weaving [7], [23]. So far, only TOSKANA [10] addresses dynamic weaving of aspects in operating systems, namely the FreeBSD kernel.

IV. THE SINGLE LANGUAGE APPROACH

In order to support flexible binding times of features that are crosscutting concerns in the implementation (see section II-C), we are working on an extension to AspectC++ that allows the developer to write both static and dynamic aspects in the same AspectC++ language. Thus, it would be transparent for the developer whether s/he is describing a static or a dynamic aspect. Following this "single language approach", the decision whether some aspect is static or dynamic can be postponed to the configuration stage and has no impact on the implementation stage. This approach supports deriving systems that offer as much dynamism as necessary while it still allows resolving as much statically as possible.

The single language approach gives rise to the question, which of the well-known aspect-oriented language features are able to be implemented with a dynamic aspect weaver. As mentioned in section III, only a few dynamic aspect weavers targeted at the C/C++ domain have been described. Most of them support only a very limited set of aspect-oriented language features in comparison to the (normally) statically woven AspectC++ language [24]. Dynamic aspect weavers in the Java domain typically support more AOP features than their counterparts for C/C++, but still a lot of features known from static weavers are not available (see table I). The most important missing features in dynamic weaving environments are:

- all kinds of introductions
- access to join point context information by the advice code
- advice ordering to deal with aspect interaction problems

For the single language approach we would like to support the same set of AOP features regardless of the weaving time. Therefore, we will now discuss the features, which are typically not available for programmers of dynamic aspects. The goal is to find out whether (1) there is a fundamental semantic problem that prohibits this feature in a dynamic weaver environment, (2) the feature could be implemented, but is too expensive, or (3) there is no reason for the omission and a viable implementation is possible.

A. Dynamic Introductions – General Remarks

In AspectC++ (and also in AspectJ) the component code may reference new elements (functions, attributes, or types) of a class, which are introduced by an aspect. In this case the static weaver makes sure that the declaration of the new element is visible *before* the referencing code fragments are compiled. However, this is simply impossible if the introduction is performed by a dynamic aspect, which is unknown when the component code is compiled. The elements introduced by a dynamic aspect can only be referenced by code that knows the aspect. Thus, the referencing code has either to be loaded together with the dynamic aspect (such as advice or aspect member functions) or is loaded later, but aware of the aspect. The result is a "knowledge hierarchy" of aspects, which introduces elements, and the code which references the introductions.

On the one hand the hierarchical relationship of modules means that the runtime system has to make sure that no dynamic aspect is unloaded as long as other modules, which are aware of it, are still loaded. On the other hand we could statically prepare a dynamically loadable module with respect to the aspects it is aware of. We will later see that this possibility is important for coping with some of the problematic feature that will be discussed in the following paragraphs.

B. Dynamic Introduction of Types

A dynamically introduced type could, for example, be a type alias (`typedef`) or a inner class. As inner classes and type definitions do not affect the object layout in C++, introductions of types are not problematic. A newly loaded module would be aware of all the types introduced by its parents in the knowledge hierarchy, while the root module knows none of these extensions.

C. Dynamic Introduction of Attributes

Dynamic attribute introduction has the fundamental problem that either (1) the layout of objects, which are already instantiated in the running system, would have to be changed dynamically or that (2) objects with the new and the old layout have to co-exist. This problem is at least as old as the problem of schema evolution in databases. As far as we know, no dynamic weaver supports this feature yet.

Our own dynamic weaver implementation also does not support this feature, yet, but the following idea might solve the problem in our context: For compiled languages with their tight coupling of code and data, changing the object layout would require too many dynamic changes in the machine code. Therefore, all objects should contain an additional "introduction pointer", which can be used at runtime to add and remove dynamically introduced attributes. Because *all* objects contain this pointer, no relocation or code manipulation will be necessary. Modules, which are aware of an introduced attribute, are compiled with the introducing aspect. The weaver, which prepares the modules, could transform any attribute access operation into a call of an accessor function that looks up the attribute in a data structure referred by the introduction pointer.

Objects, which already exist in the running system shall also be extended. This avoids the need to check, whether an object contains the introduced attribute, in code, which is aware of the introduction. The aspect programmer will have to provide an initialization expression for the introduction. The same expression can be used in the case of static weaving for the initialization during object construction.

D. Dynamic Introduction of a Non-Virtual Member Function

The implementation of dynamic introductions of member functions is in some cases very simple, because a referencing module knows the introducing aspect and, thus, can be affected by the aspect during compilation. For example, inlined member functions could be introduced into the target class during compilation of the referencing modules. All calls to this function can be resolved and replaced by an inlined version of the introduced code. The running system is not affected by this manipulation. The same technique even works for non-inline functions. In this case the introducing aspect has to provide the compiled code of the introduced function when it is loaded. It furthermore should statically affect the definition of the target class for the modules, which are compiled later. These modules will find the declaration of the introduced function in their version of the target class definition and the compiler will generate a function call with an unresolved reference. This reference will later be resolved by the dynamic linker when the module is loaded into the system after the introducing aspect.

E. Dynamic Introduction of a Virtual Member Function

More problematic are introductions of virtual member functions. Simply inserting a declaration and definition into the target class while compiling referencing modules would result in inconsistent versions of virtual function tables or even inconsistent assumptions about the object layout in different subsystems. This leads to unpredictable behavior.

Introductions of virtual member functions into a base class should furthermore be forbidden, if a non-virtual function with the same name and signature already exists in any derived class. Doing this would effectively mean to convert a non-virtual function into a virtual function during runtime. Too many points in the machine code (all calls) would have to be manipulated by this operation. Furthermore, this would significantly change the behavior of the running system and its expectations about the targets of calls.

Nevertheless, dynamic introductions of virtual member functions are possible. As a referencing module has to be aware of the introducing aspect, all calls to virtual member functions introduced by the aspect can be manipulated to use a separate dynamic dispatch mechanism. This can be implemented, for instance, by using an additional virtual function table which is maintained by the aspect.

F. Ordering of Advice Invocations

As far as we know current dynamic weavers for C/C++ ignore the problem of ordering the execution of multiple advice definitions that affect the same join point. However, if the advice invocation is controlled by a runtime system, it would be possible to sort the advice by considering arbitrary policies.

G. Access to Context Information

Providing context information about the current join point for advice code is a problematic feature in the case of dynamic weaving in compiled languages. The reason is that some parts of the context, which we are used to have in static aspect implementations, are generated "on demand". The most prominent example is the string representation of the join point signature. However, when the component code is compiled the dynamic advice code is still unknown. Therefore, no information is available about the actually needed context information. If we assume that all context information (e.g. a signature string, argument values, current and target object pointer) will or might be needed, the overhead could become tremendous.

The second problem is to access the context information in a type-safe way from within the dynamic advice code. Providing a pointer to the current stack frame is not enough. Therefore, the dynamic weaver environment has to pass this context information correctly typed as functions parameters to the advice code. To perform this task additional wrapper code is necessary, which at best has to be loaded together with the dynamic aspect, because the runtime system is not aware of the advice and its demands.

H. Generic and Generative Advice

An additional level of complexity is reached with the AspectC++ feature of generic and generative advice [16]. This kind of advice uses the static type information about the target join points to access the context information in a generic way or even to instantiate C++ templates or template meta-programs with these types. This means that join point-specific advice code instances are created. In the case of a dynamic aspect, this kind of advice would either have to be compiled dynamically or the instances have to be created when the dynamic aspect is compiled. In many cases the instantiation at compile time would be possible, because the target join points might already be known.

I. Conclusions: The Combination of Static and Dynamic Weaving

In the previous sections we discussed the most important language features, which are usually supported by static aspect weavers, but not available in most dynamic weavers for compiled languages like C/C++. The discussion did not go very much into the details of possible solutions. Nevertheless, it shows that in many cases these problematic features are implementable and, thus, a single language approach is realistic.

An important lesson learned from the discussion is that many of the features can only be implemented if dynamically loaded aspects and other modules can be prepared at compile time. For example, dynamic introductions of non-virtual member functions become almost trivial with this assumption. Therefore, we conclude that the combination of static and dynamic weaving is even more beneficial than pure dynamic weaving.

V. A Family-Based Dynamic Weaving Infrastructure

According to the goal of *minimal overhead* (Section II-D) the costs of AOP should scale. This means that the costs should depend only on the *actually affected* joinpoints, *actually woven* aspects, *actually given* advice, and *actually used* AOP features. There should be little to no "ground overhead" induced by the general facility to use aspects at all.

For static weaving, it is relatively simple to provide such scalability and thereby fulfill the goal, as the actual set of affected joinpoints, used AOP features, etc. is fixed and well-known. Hence, it is possible to automatically tailor down the woven-in code and data at weave time according to the specific requirements.[1] A dynamic weaver, however, has to consider *potential* joinpoints, which may be affected by *potential* aspects that *potentially* use certain AOP features. It is not possible to *automatically* tailor this down, as theoretically any joinpoint and any available feature may be required by an unforeseen aspect loaded at runtime.

This flexibility, undoubtedly desirable, comes at a high price: Compared to static weaving, dynamic weaving induces high overhead ([13]) and/or is limited to a small set of features only (Table I). Different applications, however, have different demands regarding both, flexibility and features. In one case, a specific dynamic weaving technique may not provide a required feature, while in another case a less demanding application is forced to pay for functionality and flexibility that is not required. Such one-size-fits-all approach is especially not acceptable in the domain of embedded systems. Most embedded devices are special-purpose systems with very specific requirements, but strictly limited memory resources.

A. Application-specific Weaver Construction

To overcome these disadvantages, we propose a family-based approach of constructing *application-specific* dynamic

[1]AspectC++, for instance, tailors the stored context information for each joinpoint to the amount which is actually accessed by advice.

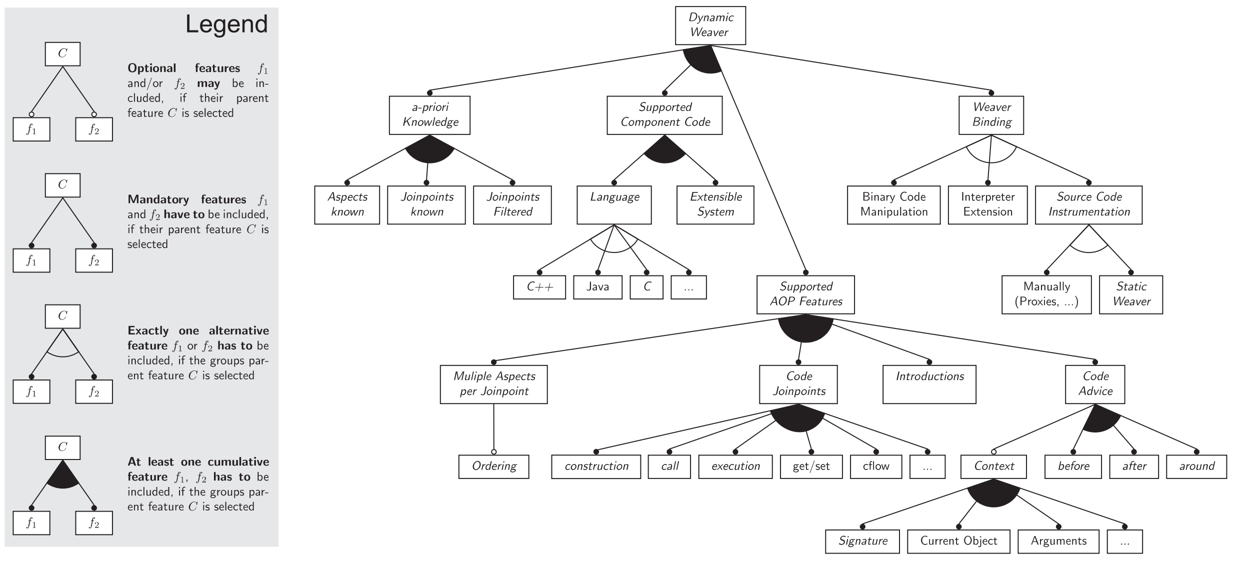
Legend
C
f1
f2
Optional features f1 and/or f2 may be included, if their parent feature C is selected
Mandatory features f1 and f2 have to be included, if their parent feature C is selected
Exactly one alternative feature f1 or f2 has to be included, if the groups parent feature C is selected
At least one cumulative feature f1, f2 has to be included, if the groups parent feature C is selected
Dynamic Weaver
a-priori Knowledge
Supported Component Code
Weaver Binding
Aspects known
Joinpoints known
Joinpoints Filtered
Language
Extensible System
Binary Code Manipulation
Interpreter Extension
Source Code Instrumentation
C++
Java
C
...
Supported AOP Features
Manually (Proxies, ...)
Static Weaver
Muliple Aspects per Joinpoint
Code Joinpoints
Introductions
Code Advice
Ordering
construction
call
execution
get/set
cflow
...
Context
before
after
around
Signature
Current Object
Arguments
...

Fig. 1. The Family of Dynamic Weavers (extended from [11]). Features actually used/supported by our current prototype implementation are typed in *italics*.

weavers from a *family of weavers*. The feature diagram in Figure 1 describes the commonalities and variabilities of the different dynamic weavers. An application-specific tailored weaver is constructed from the weaver family by leaving out as much as possible and selecting only those features, which are required to fulfill the applications demands. Besides a fine-grained selection of the available AOP features (*Supported AOP Features)*, it is especially possible to exploit *a-priori-knowledge* about the system and its execution environment:

- In the domain of embedded devices, for instance, the set of classes – and thereby the set of potential joinpoints – is usually known in advance (*JoinPoints Known*). Hence, it is possible to match aspects already at their compile-time to the set of joinpoints they later need to be registered for. An expensive match engine to retrieve joinpoint shadows at runtime can be omitted from the dynamic weaver.
- It is often possible to explicitly filter the set of potential joinpoints even further to get a much smaller subset that "makes sense" (*JoinPoints Filtered*), such as the potential points of interest for system strategies and other crosscutting concerns. The vast majority of potential joinpoints in a system is never used by any aspect, as many joinpoints hardly contribute to the application semantics. The execution or control flows of basic library functions, for instance, can be considered as such "low-semantics joinpoints".
- If even the set of potential aspects is known in advance (*Aspects Known*), it is possible to generate the joinpoint filter automatically from their pointcut descriptions. Furthermore, the maximum number of registered aspects for each joinpoint can be pre-calculated in this case, so it is possible to fix the size of the runtime advice lists associated with each joinpoint and omit the necessity for using costly dynamic data structures. One more benefit is that the order of aspect execution can be defined and resolved statically, if all aspects are known in advance.

This approach facilitates low-cost dynamic weavers by providing and exploiting as much knowledge about the system and its execution environment as possible. This is comparable to the optimizations performed by an ideal static weaver, which basically exploits the same information for this purpose: *actually affected* joinpoints, *actually woven* aspects, *actually given* advice, and *actually used* AOP features. The main difference is that this information is *implicitly* available to a static weaver, while it has to be *explicitly* provided for the generation of a tailored dynamic weaver.

B. Summary

Overall, a family-based dynamic weaver infrastructure allows a fine-grained adjustment of the trade-off between flexibility and required resources. In conjunction with the *single language approach* (Section IV), this perfectly fulfills the goal of *minimal overhead:* For any kind of application, it is now possible to weave as much as possible statically, while providing as much runtime flexibility as necessary. Static versus dynamic weaving of aspects becomes a configurable and tailorable property.

VI. Implementation

The following sections provide a brief overview of our dynamic weaver family, as appropriate to understand the integration of the AspectC++ "single language approach" and "family-based weaver infrastructure" into an operating system product line. Static weaving with AspectC++ is already used and approved [25], [16], [23], [24], therefore we do not discuss the static part. A detailed description about the dynamic weaver architecture and runtime has already been presented in [11]. In the following, we give only a brief overview of some

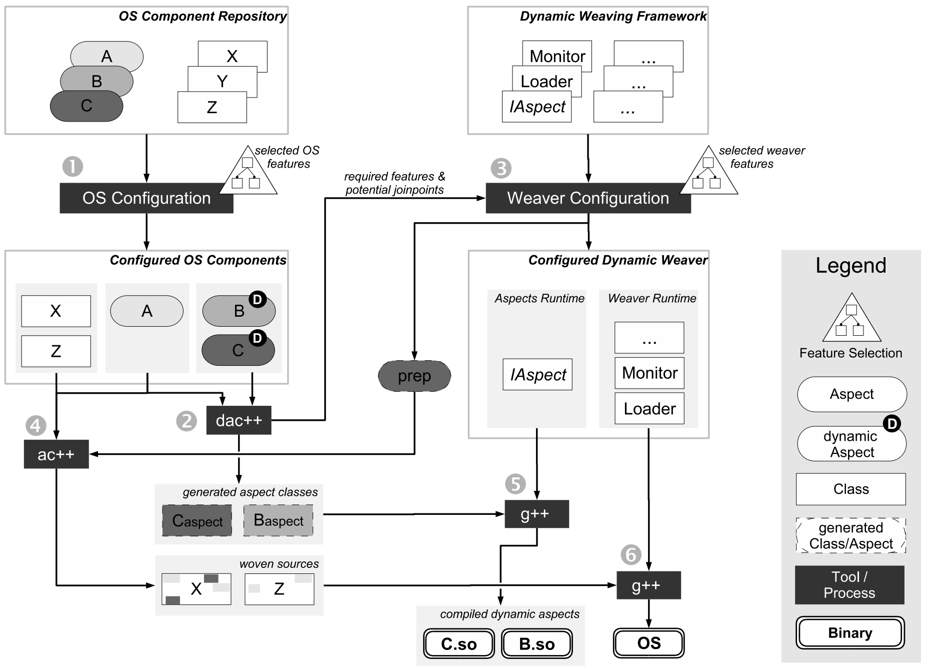

Fig. 2. Adaptable OS Tool Chain. The process starts with configuration of the OS (1), in which features (and thereby their implementing aspects) are configured as static or dynamic. For each dynamic aspect a corresponding implementation class is generated (2). The required features and potential joinpoints are determined and, together with additionally selected features, used to configure the dynamic weaver (3). The configured OS components are woven with the generated instrumentation aspect prep and all other static aspects (4). Each generated dynamic aspect class is compiled with the configured aspect runtime into a loadable module (5). The woven operating system sources are compiled with the configured weaver runtime into the final OS image (6).

fundamental concepts, but concentrate on the application of the "single language approach", which is the main contribution of this paper.

A. Basic Structure of the Dynamic Weaver Family

The prototype implementation of our tailorable dynamic weaving infrastructure uses *source code instrumentation by static weaving* to bind the component code to the dynamic weaver. For this purpose, the component code is instrumented in all joinpoints of interest by a *static preparation aspect* (`prep`), which activates the *runtime monitor* (`Monitor`) if the control flow passes such joinpoint. The runtime monitor, in turn, activates the registered advice and passes joinpoint-specific context information to the advice code. Code advice is implemented as member functions of *(loadable) dynamic aspect classes*, which have to provide a specific interface (`IAspect`). Dynamic aspects register and deregister themselves against the runtime monitor with their pointcuts and advice. The dynamic aspect code itself can be linked either statically with the component code, or loaded at runtime by means of a *dynamic aspect loader* (`Loader`).

B. Implemented Dynamic Weaver Features

The feature diagram in Figure 1 shows, which features are already available in our implementation. The *Supported AOP Features* cover *before*, *after* and *around* advice for *call* as well as *execution* joinpoints. Currently not supported are *cflow* and *get/set* joinpoints. While an implementation of the former should be straightforward[2], the latter can be considered as challenging to impossible in languages that support C-style pointers.[3]

The support for *Introductions* (static crosscutting) covers non-virtual member functions, static functions and new types. Introduced elements are afterwards accessible by all modules in the "knowledge hierarchy" that can be aware of them, as already discussed in Section IV. Hence, a selection of the *Introductions* feature requires to select *Extensible System* as well.

Our weaver family supports a very fine-grained selection of the mentioned features. The induced memory overhead scales with the amount of selected features [11]. Additionally, it is possible to exploit the different types of *a-priory Knowledge* in our implementation as discussed in Section V-A. This allows an even further reduction of the overhead for dynamic weaving. The obtained benefits are demonstrated in the case study (Section VII).

[2]It basically requires an extension of the static preparation aspect by some advice that increments/decrements a cflow-counter at all joinpoints where a cflow of interest is entered/leaved.

[3]The support for get/set joinpoints in existing weavers (namely Arachne and TOSKANA, see Table I) is quite limited, as it is restricted to direct access of global variables.

C. Build System

Figure 2 demonstrates the envisioned tool chain and configuration process for the adaptable OS product line. Because of the single language approach, it is possible to use the same aspect code for both statically and dynamically bound features. Aspects configured as static are woven with the component code by our ac++ weaver.[4] Aspects configured as dynamic are transformed into C++ classes and compiled by the C++ compiler into loadable modules. Hence, application-tailored OS family members with configurable dynamism and minimal overhead can be generated according to a specific application's requirements.

VII. Case Study: Static and Dynamic Aspects in the eCos Operating System

A. System Overview

eCos is a small and highly configurable operating system developed by *Cygnus Solutions* and now maintained an distributed by *eCosCentric Limited,* targeted for the market of embedded systems. It is available for a broad variety of 16 and 32 bit microprocessor architectures (PPC, x86, H8/300, ARM7, ARM9, ...) and used in many different application domains. The eCos system itself is provided as a repository of various components, which are configured *statically* with a configuration tool called *eCosConfig*. The components are implemented in a mixture of C++, C, C-preprocessor macros and assembly code. After the user selects an appropriate eCos configuration within *eCosConfig*, a configuration-specific system of headers and makefiles is generated, which is used to build the *eCos-library*. Against this library the final applications will be linked.

B. Analysis

As a larger case study, we analyzed several parts of the eCos system, namely the kernel, C library, POSIX subsystem, μITRON subsystem, Memory Management, Wallclock Driver, and Watchdog Driver. For the following discussion we will concentrate on the eCos kernel, which is the biggest of these components.

The first goal was to figure out the positions and the amount of code that implements highly crosscutting concerns and locally crosscutting optional features. The analysis revealed that 23.54% of the kernel source code are needed to implement four highly crosscutting concerns: *Tracing*, *Assertion*, and *Kernel Instrumentation* (profiling) for development support and *Interrupt Synchronization*. Table II (column "original") presents the numbers for each of these concerns individually.

Besides highly crosscutting concerns, the degree of scattering of local configuration options in the thread management of the kernel were analyzed. Table III (column "#original"[5]) shows that almost all of these configurable features affect more than one point in the source code. They are crosscutting concerns in specific subsystems.

[4]available at http://www.aspectc.org/

[5]number in brackets will be explained later

TABLE II

Amount of CCCs in the source code of the kernel before and after refactoring

	Kernel			
	original		aspectized	
LOC	5205	100 %	4527	100 %
Tracing	336	6.46 %	4	< 0.1 %
Assertions	384	7.38 %	286	6.32 %
Kernel Instrumentation	319	6.13 %	0	0 %
Interrupt Synchronization	186	3.57 %	0	0 %
CCC Code	1225	23.54 %	290	6.41 %
Component Code	3980	76.46 %	4237	93.59 %

TABLE III

#ifdef blocks caused by configurable thread features. Numbers in brackets show how many of them were in C++ code.

Option	**# original**	**# aspectized**
`THREADS_NAME`	14 (3)	12 (1)
`THREADS_LIST`	9 (4)	5 (0)
`THREADS_STACK_LIMIT`	8 (7)	1 (0)
`THREADS_STACK_CHECKING`	6 (6)	1 (1)
`THREADS_STACK_MEASUREMENT`	9 (2)	7 (0)
`THREADS_DATA`	7 (3)	4 (0)
`THREADS_DESTRUCTORS`	5 (3)	2 (0)
`THREADS_DESTRUCTORS_PER_THREAD`	12 (11)	3 (1)

C. Aspects in eCos

During the case study, we further increased the modularity and configurability of eCos by "aspectizing" the highly crosscutting concerns and crosscutting configurable features mentioned in the previous section. Additionally, several new configurable features have been implemented. Overall, we have implemented 147 aspects for 17 concerns, which are woven by the static AspectC++ weaver.

The effort to refactor the implementation was low, because the affected code was easy to identify. The eCos developers mostly use macros for the implementation of highly crosscutting concerns such as *Tracing* to avoid code redundancy. Configurable feature implementations are always encapsulated in an `#ifdef` block for conditional compilation.

The results are shown in table II and III (right columns).

TABLE IV

Memory overhead (in bytes) introduced by the static and the dynamic version of the tracing aspect

		Kernel		Tracing Aspect		Dyn. Weaver		Sums	
		RAM	ROM	RAM	ROM	RAM	ROM	RAM	ROM
	no tracing	16474	6758	0	0	0	0	0	0
static	complete	0	11733	0	212	0	0	0	11945
	common	0	4866	0	212	0	0	0	5078
	interrupt	0	1252	0	212	0	0	0	1464
	scheduler	0	4800	0	212	0	0	0	5012
	sync	0	1016	0	212	0	0	0	1228
dynamic	complete	0	12078	28	471	756	28	784	12577
	common	0	5000	28	471	304	28	332	5499
	interrupt	0	1211	28	471	84	28	112	1710
	scheduler	0	5058	28	471	184	28	212	5557
	sync	0	916	28	471	184	28	212	1415

TABLE V

RUNTIME OVERHEAD (IN CLOCK CYCLES) INTRODUCED BY THE STATIC AND THE DYNAMIC VERSION OF THE TRACING AND THE COUNTING ASPECT

		Tracing				Counting			
		create thread	resume thread	change thread	exit thread	create thread	resume thread	change thread	exit thread
	base	178	127	236	1219	175	129	236	1219
static	complete	54520	141120	251157	1171898	4	19	58	235
	common	34356	29087	37053	215661	0	9	11	-10
	interrupt	25	26769	34115	169776	1	3	25	79
	scheduler	19635	84379	179388	732644	2	17	31	22
	sync	7	6	7	53818	2	13	0	22
dynamic (prepared)	complete	84	286	484	2294	49	123	309	1137
	common	64	62	73	429	30	36	51	197
	interrupt	-2	52	97	378	1	23	69	196
	scheduler	41	133	314	1272	19	60	195	633
	sync	1	0	0	115	3	0	4	55
dynamic (woven)	complete	54749	141783	260619	1216087	264	1317	3941	21928
	common	34746	29334	37707	217688	129	176	298	2728
	interrupt	51	28753	35999	173309	1	112	232	819
	scheduler	20119	84473	187559	765476	76	474	2007	8471
	sync	25	23	116	59569	2	1	4	333

Besides a much cleaner and more reusable kernel code, we (almost) achieved the desired direct 1:1 relation of configurable features and implementation artifacts even for crosscutting concerns. The main problems we encountered were that we were not able to modularize assertions, due to their individual semantic, and the technical problem that our weaver was not able to weave in pure C code. This had a negative impact on the modularization of the configurable kernel features as many of their variation points (`#ifdef` blocks) are located in the C-API of eCos or other subsystems implemented in C. However, the numbers in brackets in table III show that almost all variation points that were addressable by AspectC++ have been replaced by modular aspect implementations.

D. Dynamic Aspects

Based on the "aspectized" eCos, which was described in the last section, we conducted several experiments with our configurable dynamic aspect weaver. In a first scenario the *Tracing* aspect was compiled as a dynamic aspect. The memory consumption of the kernel, the aspect itself, and the dynamic weaver was compared with a statically woven *Tracing* aspect implementation. For this experiment the weaver was configured to support *before* and *after* advice for execution join points and the join point signature as context information. The results are shown in table IV. *Common*, *interrupt*, *scheduler*, and *sync* are the four main subsystems of the eCos kernel. The corresponding rows describe the costs of weaving only in these subsystems and not in the *complete* kernel. The table shows that overhead for having a dynamic weaver and a dynamic aspect instead of a static one is acceptable. Only 756/28 bytes (RAM/ROM) for the dynamic weaver and additional 28/259 bytes (259 = 471-212) for the dynamic version of the aspects are needed. This is not much compared to the normal footprint of a system like eCos[6] and no problem for a development aspect. These are the numbers for *complete* instrumentation. The ability of the dynamic weaver to support only a filtered set of potential dynamic join points facilitates even further scaling of costs.

[6]The costs for a dynamic loader and linker are ignored here, because these are not specific costs of dynamic weaving.

Besides memory consumption, the performance of typical system calls was analyzed. The results are shown in table V. The left part contains the results for the *Tracing* aspect while the right part shows the results for a new *Counting* aspect. The reason for a further aspect, which implements simple kernel profiling, was the enormous runtime consumption of the Tracing aspect. *Counting* better represents typical production aspects. The table shows that the statically woven *Counting* aspect costs almost no time while *Tracing* costs several thousand clock cycles, which is magnitudes bigger than the costs of the whole system call. *Dynamic (prepared)* means that during the measurements the kernel code was only prepared for a dynamic aspect, but none was woven (case *dynamic(woven)*). In all test cases, the numbers for *dynamic(prepared)* for *Counting* are about 50% of the value for *Tracing*. The reason is that the *Counting* aspect does not need *after* advice. We could therefore configure the dynamic weaver and the preparation aspect to avoid the run-time costs of this feature. Furthermore, the table shows that also the runtime significantly benefits if the set of prepared join points is tailored according to the specific demands.

E. Discussion

In general, the numbers acquired during this case study show that for many concerns in system software aspect-oriented implementations and especially dynamically woven aspects are affordable. Our prototypical dynamic weaver implementation even leaves some space for further improvements, especially with respect to performance.

After the refactoring and the integration of the dynamic weaver infrastructure into eCos, the system now offers an even better static configurability as well as dynamic adaptability. For mobile devices in changing environments we could imagine several applications of this new feature. For example, by preparing the system call API for dynamic aspect weaving various security policies could be woven into the running system without the need to stop and restart.

VIII. Conclusions and Future Work

Dynamic adaptation of system software and especially dynamic aspect weaving are on the one hand very convenient for the programmers and users. On the other hand dynamic adaptation and dynamic aspect weaving are always more expensive than their static counterpart. Therefore, we advocate for a development tool chain that supports both. With a *Single Language Approach* we can even easily switch from static adaptation to dynamic adaptability of features and vice versa.

The costs of dynamic aspect weaving are crucial in the domain of system software. These costs are caused by the runtime system, which, for instance, has to manage aspect registration and ordering, and the weaver binding, which is responsible for detecting that a join point has been reached. Both costs can be significantly reduced if the dynamic weaver is tailored with respect to the specific demands in a particular project. Understanding a dynamic weaver as a product line, i.e. a Family-Based Dynamic Weaver Infrastructure, therefore helps to avoid monolithic solutions that are eventually satisfiable for no one. The main goal is to allow developers to use as much *a priori* knowledge as possible in order to avoid dynamism wherever s/he can.

Currently, our `dac++` implementation exists only as a prototype. The transformation from static aspect implementations into dynamic aspect classes still requires manual corrections and also our runtime system requires further improvements. Therefore, we will continuously improve our implementation. On the conceptual side the single language approach and the combination of static and dynamic weaving will be further investigated.

References

[1] eCos homepage. http://ecos.sourceware.org/.

[2] S. Almajali and T. Elrad. A Dynamic Aspect-Oriented C++ Using MOP with Minimal Hook Weaving Approach. In *2004 Dynamic Aspects Workshop (AOSD-DAW '04)*, pages 1–8, March 2004. published as RIACS Technical Report 04.01.

[3] S. Aussmann and M. Haupt. Axon - Dynamic AOP through Runtime Inspection and Monitoring. In *2003 Advancing the State-of-the-Art in Run-Time Inspection Workshop (ECOOP-ASARTI '03)*, July 2003.

[4] 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 *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[5] C. Bockisch, M. Haupt, M. Mezini, K. Ostermannd, and G. Kiczales. Virtual machine support for dynamic join points. In *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD '04)*, pages 83–92, Lancaster, UK, March 2004. ACM.

[6] J. Boner. AspectWerkz - Dynamic AOP for Java. In Karl Lieberherr, editor, *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD '04)*, pages 51–62, Lancaster, UK, March 2004. ACM.

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

[8] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* AW, May 2000.

[9] R. Douence, T. Fritz, N. Loriant, J. M. Menaud, M. S. Devillechaise, and M. Suedholt. An expressive aspect language for system applications with Arachne. In Peri Tarr, editor, *4th Int. Conf. on Aspect-Oriented Software Development (AOSD '05)*, pages 27–38, Chicago, Illinois, March 2005. ACM.

[10] M. Engel and B. Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. In Peri Tarr, editor, *4th Int. Conf. on Aspect-Oriented Software Development (AOSD '05)*, pages 51–62, Chicago, Illinois, March 2005. ACM.

[11] Wasif Gilani and Olaf Spinczyk. Dynamic aspect weaver family for family-based adaptable systems. In *NetObjectDays (NODe '05)*, Lecture Notes in Informatics, Erfurt, Germany, September 2005. German Society of Informatics.

[12] Jeff Gray, Jing Zhang, Yuehua Lin, Suman Roychoudhury, Hui Wu1, Rajesh Sudarsan, Aniruddha Gokhale, Sandeep Neema, Feng Shi, and Ted Bapty. Model-driven program transformation of a large avionics framework. In G. Karsai and E. Visser, editors, *3rd Int. Conf. on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *LNCS*, pages 361–378. Springer, October 2004.

[13] Michael Haupt and Mira Mezini. Micro-measurements for dynamic aspect-oriented systems. In *NetObjectDays (NODe '04)*, volume 3263 of *LNCS*, pages 81–96, Erfurt, Germany, September 2004. Springer.

[14] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. In *IFIP/ACM Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware '00)*, New York, USA, April 2000. ACM.

[15] T. Ledoux, W. Cazzola, and F. Rivard. OpenCorba: A reflective open broker. In Pierre Cointe, editor, *Reflection '99*, volume 1616 of *LNCS*, pages 197–214, London, UK, 1999. Springer.

[16] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *3rd Int. Conf. on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *LNCS*, pages 55–74. Springer, October 2004.

[17] Daniel Lohmann and Olaf Spinczyk. Architecture-Neutral Operating System Components. *23rd ACM Symp. on OS Principles (SOSP '03)*, October 2003. WiP session.

[18] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *4th AOSD W'shop on Aspects, Components and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, March 2005. Northeastern University, Boston (NU-CCIS-05-03).

[19] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible framework for AOP in Java. In Akinori Yonezawa amd Satoshi Matsuoka, editor, *Reflection '01*, volume 2192 of *LNCS*, pages 1–24, Kyoto, Japan, 2001. Springer.

[20] A. Popovici, G. Alonso, and T. Gross. Just in Time Aspects: efficient dynamic weaving for java. In Mehmet Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 100–109, Boston, MA, USA, March 2003. ACM.

[21] Y. Sato, S. Chiba, and M. Tatsubori. A selective, just-in-time aspect weaver. In *2nd Int. Conf. on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *LNCS*, pages 189–208, Erfurt, Germany, October 2003. Springer.

[22] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[23] Olaf Spinczyk and Daniel Lohmann. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS European W'shop*, pages 188–192, Leuven, Belgium, September 2004. ACM.

[24] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in AOP with AspectC++. In *4th Int. Conf. on Software Methodologies, Tools and Techniques (SoMeT '05)*, Frontiers in Artificial Intelligence and Applications, Tokyo, Japan, September 2005. IOS Press. (to appear).

[25] A. Tešanović, K. Sheng, and J. Hansson. Application-tailored database systems: a case of aspects in an embedded database. In *8th Int. Database Engineering and Applications Symp. (IDEAS '04)*, Coimbra, Portugal, July 2004. IEEE.

[26] C. Zhang and H. A. Jacobson. TinyC: Towards building a dynamic weaving aspect language for C. In *2003 Foundations of Aspect-Oriented Languages Workshop (AOSD-FOAL '03)*, March 2003.