# A Quantitative Analysis of Aspects in the eCos Kernel*

Daniel Lohmann, Fabian Scheler, Reinhard Tartler,
Olaf Spinczyk, and Wolfgang Schröder-Preikschat
Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
{lohmann,scheler,tartler,spinczyk,wosch}@cs.fau.de

## ABSTRACT

Nearly ten years after its first presentation and five years after its first application to operating systems, the suitability of *Aspect-Oriented Programming (AOP)* for the development of operating system kernels is still highly in dispute. While the AOP advocacy emphasizes the benefits of AOP towards better configurability and maintainability of system software, most kernel developers express a sound skepticism regarding the thereby induced runtime and memory costs: Operating system kernels have to be lean and efficient.

We have analyzed the runtime and memory costs of aspects in general, on the level of $\mu$-benchmarks, and by refactoring and extending the eCos operating system kernel using AspectC++, an AOP extension to the C++ language. Our results show that most AOP features do not induce a intrinsic overhead and that the actual overhead induced by AspectC++ is very low. We have also analyzed a test case with significant aspect-related costs. This example shows how the structure of the underlying kernel can have a negative impact on aspect implementations and how these costs can be avoided by an aspect-aware design.

Based on this analysis, our conclusion is that AOP *is* suitable for the development of operating system kernels and other kinds of highly efficient infrastructure software.

## Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.3 [**Programming Languages**]: Language Constructs and Features

## General Terms

Languages, Measurement, Experimentation, Design

## Keywords

Aspect-oriented Programming (AOP), AspectC++, eCos, Footprint

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


*EuroSys'06,* April 18–21, 2006, Leuven, Belgium.


## 1. INTRODUCTION

Nearly ten years after its first presentation[22] and five years after its first application to operating systems[4, 7], it is still a controversial question whether *Aspect-Oriented Programming (AOP)* provides a real benefit for the development of operating system kernels. Many OS developers have a sound skepticism regarding the suitability of high-level paradigms, such as AOP, for the system-level development problems they have to solve in their daily work. By the notion of *aspects,* AOP provides means to encapsulate and separate the implementation of concerns that otherwise would have to be scattered over implementation artifacts of other concerns. Typical examples for such "cross-cutting" concerns in operating systems are multi-user support or proper synchronization. Separation of concerns is clearly an important issue in the domain of operating systems. This was shown by retroactive studies in the context of the FreeBSD kernel [6]. It is also evident from the "`#ifdef`-hell" that can be found in the code of many current operating systems. Especially systems that strive for a high level of tailorability and configurability suffer from the negative impacts of scattered concern implementations on readability, maintainability and reuse. A good example for this class of systems is the embedded configurable operating system eCos, which is the subject of a case study that will be described later in this paper.

Nevertheless, there are serious doubts whether AOP provides an efficient solution for these problems. The most mentioned concerns regarding a more extensive use of aspects in operating systems are *comprehensibility*, *resource efficiency*, and *real applications*:

### *Comprehensibility*

Separating concern implementations into aspects involves the risk of a *loss of comprehensibility*. The reason is that aspects "affect" the static structure or the dynamic behavior of other modules without being "visible" in the source code of these modules. While such higher level of abstraction may be considered as an advantage in other domains, system developers usually prefer a more explicit representation of the program's implementation in the code.

Actually, AOP fits surprisingly well into this philosophy, as on the technical level it is still a very code-driven approach. AOP provides novel declarative means to specify *where* something should apply and thus facilitates to separate the *what* of a concern's *implementation* from the *where* of it's *application*. The actual implementation (the *what*), however, is still specified in the same language as other component code, e.g. Java, C, or C++. Hence, it is possible to present developers a precise view of the merged aspect and component code on source level and thereby reach the required level of comprehensibility. This is basically a question of tools, namely of AOP-aware editors and debuggers that visualize in the source how aspects affect the currently viewed or edited

component. For AspectJ[21] such tool support already exists and has reached an industry-strength level of maturity[20]. For other languages, such as AspectC++[28], it is actively being developed (http://acdt.aspectc.org). Overall, the comprehensibility issue is solvable and not a fundamental problem of AOP.

### *Resource Efficiency*

Efficiency in terms of runtime and memory overhead is probably the most important requirement for any paradigm to be adopted for the development of OS kernels. In a domain where C and C++ for good reasons are still the dominating languages, efficiency is not a feature among others, it is a prerequisite. OOP, for instance, is still rarely used as some of its fundamental concepts, such as *late binding*, induce a significant (and in many cases inevitable) overhead[11].

So far, only few studies analyze the costs of aspects. The existing studies[12, 3] were mainly conducted in the Java domain using AspectJ and are somewhat disappointing, as they show that AOP with AspectJ indeed induces some overhead[12]. It is, however, questionable if these results are suitable for any generalization regarding the *intrinsic overhead* of aspects. In this paper we show that most AOP concepts do *not* induce an inevitable overhead.

### *Applications*

One often hears the question if there is "any real application besides tracing" for using general-purpose aspect languages in kernel development. Coady et al. retroactively evaluated the evolution of four scattered OS concerns (prefetching, disk quotas, blocking, and page daemon activation) in the FreeBSD kernel using the general-purpose AspectC language[7, 6]. It was shown that an aspect-oriented implementation would have led to significantly better evolvability. Due to missing tool support (namely an "aspect weaver"), her study did cover only a relatively small part of the kernel code base and no heavily cross-cutting concerns such as tracing or kernel diagnostics. Our group conducted some experiments with AspectC++ in the PURE embedded operating system family[5] by implementing OS concerns like interrupt synchronization[25] and the driver execution model[27] with aspects. Not a general-purpose AOP language, but an AOP-inspired language of temporal logic was used by Åberg et al. to integrate the Bossa scheduler framework into the Linux kernel[2]. C4 uses AOP concepts to implement a "semantic patch system" for the application of kernel patches[16]. Several other publication show that AOP provides benefits for the development of configurable *infrastructure software* in the broad sense, namely middleware [8, 31, 19] and databases[29] product lines.

All these studies demonstrate that there are real cases for aspects in system software. From the separation of concerns viewpoint, which is the focus of the existing studies, the use of AOP is highly beneficial. A broader application to the resource-thrifty domain of OS kernels, however, requires an in-depth analysis of the costs of aspects, which is still missing.

## 1.1 About this Paper

The main contribution of this paper is an in-depth cost analysis. By refactoring and extending a widely-used OS kernel with aspects written in AspectC++, we have developed a case study that can be considered big enough for general conclusions. It thus helps to demystify this novel programming paradigm and to bridge the gap between the AOP community, which normally has a stronger focus on "soft properties" such as reusability and configurability, and the operating systems community, which is still skeptical and more concerned about "hard properties" such as code size or clock cycles.

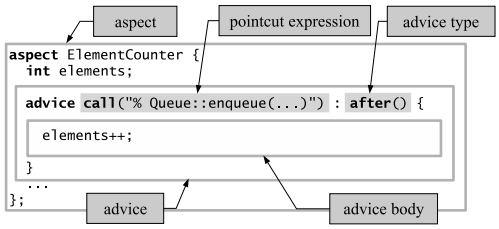



**Figure 1: Syntactical Elements of an Aspect in AspectC++**

The outline of the paper is as follows: In section 2 we give a brief introduction into the AOP terminology and the most important language features of AspectC++. Section 3 then discusses the costs of these features in terms of performance, code size, and dynamic memory consumption. We also discuss whether the costs are only related to the AspectC++ weaver implementation or inevitable costs that would even be unavoidable with an ideal weaver implementation. Section 4 presents our case study in the context of the eCos operating system kernel, including a detailed cost analysis in section 4.4. A general discussion of the results is presented in section 5. The paper ends with a discussion of related work and a summary in sections 6 and 7.

# 2. AOP AT A GLANCE

AOP is a programming paradigm that attempts to aid programmers in the separation of concerns. With classical modularization techniques programmers often have problems to achieve separation of concerns if two concerns are "cross-cutting". This typically leads to the *code tangling* and *code scattering* phenomena. Code tangling means that on the implementation level the code of two (or more) concerns is intermixed rather than separated. Scattering means that the code of one concern is not localized, but can be found in various different modules. As an example consider a simple synchronization policy: "Whenever a thread accesses a shared data structure, a semaphore shall be used to guarantee mutual exclusion". This is a concern that cross-cuts many other concerns. Its code is typically scattered over many implementation artifacts.

Although there is still a long road ahead, AOP research aims at supporting modular high-level concern implementations. For example, the code that implements the synchronization policy should be a separate module that represents the human-readable policy description almost directly in a programming language. Hence, the synchronization policy could be evolved independently from the other modules, which also could be reused in other contexts without or with different synchronization schemes.

Today, most AOP languages use the concepts and terminology that was first introduced by AspectJ[21]. In the remaining parts of this section, we will give a brief overview of the most common AOP language elements in general and the AspectC++ notion in particular, as required for understanding the remaining parts of this paper. Even though the introduction is based on AspectC++, it basically holds for any statically woven AOP language.

## 2.1 Terminology

The most relevant AOP concepts are *join-point* and *advice*. An *advice* definition describes a transformation to be performed at specific positions either in the static program structure (*static cross-cutting*) or in the runtime control flow (*dynamic cross-cutting*) of a target program. A *join-point* denotes such a specific position in the target program. Advice is given by *aspects* to sets of join-points called *pointcuts*. Pointcuts are defined declaratively in a *join-point description language*. The sentences of the join-point description

language are called *pointcut expressions*. An *aspect* encapsulates a cross-cutting concern and is otherwise very similar to a class. Besides advice definitions, it may contain class-like elements such as methods or state variables.

As an example, figure 1 illustrates the syntax of aspects written in AspectC++. The aspect increments the member variable `elements` *after* each call of the function `Queue::enqueue()`. In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. Match expressions are already primitive pointcut expressions and yield a set of *name join-points*. Name join-points represent elements of the static program structure such as classes or functions. Technically, match expressions are given as quoted strings that are evaluated against the identifiers of a C++ program. The expression `"% Queue::enqueue(...)"`, for instance, returns a name pointcut containing every (member-) function of the class `Queue` that is called `enqueue`. In the case of overloaded functions with different argument types the expression would match all of them. *Code join-points* on the other hand, represent events in the dynamic control flow of a program, such as the execution of a function. Code pointcuts are retrieved by feeding name pointcuts into certain pointcut functions such as `call()` or `execution()`. The pointcut expression `call("% Queue::enqueue(...)")`, for instance, yields all events in the dynamic control flow where a function `Queue::enqueue` is about to be called.

As pointcuts are described declaratively, the target code itself has not 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. These principles of *obliviousness* and *quantification* [14] are considered a major advantage of AOP.

## 2.2 Static Cross-cutting

An aspect that encapsulates *static cross-cutting* alters the static structure of the program. In most AOP languages, such modifications of the static structure are restricted to the extension of classes by new elements like methods, state variables or base classes.

In AspectC++, the encapsulation of static cross-cutting is supported by a specific type of advice called *introduction*. Consider the following aspect, which adds support for thread local storage to a thread control block class:

```
aspect ThreadLocalStorage {
  advice "os::ToC" : int tlsentry;
public:
  advice "os::ToC" : int getTLS() {
    return tlsentry;
  };
  advice "os::ToC" : void setTLS(int v) {
    tlsentry = v;
  };
  ...
};
```

The above aspect *introduces* a (private) state variable and some (public) accessor methods into the thread control block class, or, more precisely, into all classes that are matched by the expression `"os::ToC"`.

## 2.3 Dynamic Cross-cutting

An aspect that encapsulates *dynamic cross-cutting* intercepts certain events in the control flow of a running program. Aspects basically provide means to execute some advice code *before*, *after*, or instead of (*around*) the current statement if the event occurs. In the following, this is demonstrated by three different variants of an aspect which intercepts entries into and exits from the kernel to implement some kernel locking strategy. The advice body is identical for all three variants of the `KernelLock_x` aspect: it acquires the lock (which is a member of the aspect), proceeds to the intercepted function (`tjp->proceed()`) and finally releases the kernel lock.

```
aspect KernelLock_1 {
  pointcut kernel() = "% kernel::%(...)";
  os::Lock lock;    // aspect member variable

  advice execution(kernel()) : around() {
    lock.enter();
    tjp->proceed(); // execute the intercepted method
    lock.leave();
}};
```

In variant 1, the advice is triggered, whenever any function or method from the class or namespace `kernel` is about to be *executed*. This, however, works only if kernel functions do not invoke each other, as calls to `lock.enter()`/`lock.leave()` must not be nested. Variant 2 provides a less restrictive solution by intercepting the kernel invocation on the *caller* side:

```
aspect KernelLock_2 {
  ...
  advice call(kernel())
  && !within(kernel()) : around() {
    ...
}};
```

The `call()` pointcut function yields all events in the control flow, where a given function is about to be *called*. The `within()` pointcut function simply returns all join-points in the given classes, functions or namespaces. By *intersecting* (`&&`) all calls to `kernel()` with the negation (`!`) of all join-points inside `kernel()`, the pointcut expression finally evaluates to those calls to a `kernel()` function that are not made from a `kernel()` function itself. This, however, has another potential drawback: as the interception now takes place on the caller side, not only the OS code, but also the application code has to be woven with the aspect. In many cases, this is not feasible. In variant 3 kernel invocation is therefore again intercepted on the callee side, but further filtered to certain control flows:

```
aspect KernelLock_3 {
  ...
  advice execution(kernel())
  && !cflow(within(kernel())) : around() {
    ...
}};
```

The `cflow()` pointcut function yields all code join-points that occur while being in a given control flow. The `execution()` pointcut function yields all code join-points, where a given function is about to be *executed*. The above pointcut expression therefore evaluates to any non-nested execution of a `kernel()` function. Compared to variant 2, this solution does not require to weave the application code and furthermore reliably detects indirectly nested kernel calls.

## 2.4 Join-Point Context

In many cases, advice for dynamic cross-cutting needs to read and/or modify the join-point-specific invocation context such as the actual argument values passed to the intercepted function. To fulfill the goal of quantification, join-point specific context information has to be provided through a generic interface, as the same advice implementation should be applicable to many different join-points, such as functions with different signatures. Most AOP languages provide a *join-point API* for this purpose. In AspectC++, the join-point API is implicitly available in advice bodies through the `JoinPoint *tjp` type and instance pointer:

```
aspect Tracing {
  ...
  advice execution("% ...::%(...)"
   && !"void ...::%(...)") : after() {
    JoinPoint::Result res = *tjp->result();
    cout << "leaving " << tjp->signature()
         << " returning" << res;
}};
```

The after-advice implementation of the above `Tracing` aspect is generic. It can be applied to any function with a non-void return type, as the join-point API provides the required abstractions from the actual return type.

## 2.5 Weaving

Aspect weaving is the term used to describe the process of transforming the structure or behavior of a program in order to let aspects "affect" other modules. The AspectC++ compiler weaves by transforming AspectC++ code into ordinary C++ code. It is a preprocessor that mainly generates transparent wrapper functions. This kind of weaving is called "static weaving" as it is performed at compile-time. "Dynamic weaving" is a different weaving approach that supports to weave aspect code into a running program. In this paper we focus on the costs of static weaving only. Note that a static aspect weaver can support aspects that affect static as well as dynamic join points.

# 3. THE COSTS OF ASPECTS

In the previous section, we introduced the most typical AOP language concepts to achieve a better separation of statically and dynamically cross-cutting concerns. This gives rise to the question, what are the actual *runtime costs* in terms of CPU and memory overhead if using these concepts. Which AOP language elements do lead to significant costs and which do not? Are there *fundamental reasons* for these costs or are they basically a result of non-optimal weaving? Fundamental reasons would limit the applicability of AOP to efficient system software in general, while otherwise it would be a question of better aspect weavers.

In the following, we first take the perspective of an *ideal aspect weaver* to figure out for which AOP features in the general case an overhead is inevitable, probable, or not necessary at all. The discussion is based on the assumption of using *static weaving* with a statically typed and compiled language, such as C or C++.

After the discussion from the ideal weaver perspective, we present some results from an cost analysis of a real weaver, namely the static AspectC++ weaver `ac++`.

## 3.1 The Ideal Weaver Perspective

Considering an *ideal static weaver* for a statically typed and compiled language, most AOP concepts do not lead to an inevitable runtime overhead in the target program, as they are completely resolvable at weave-time:

### *Static Cross-cutting*

Introductions, the AOP concept to encapsulate static cross-cutting, generally do not lead to any overhead. In a statically typed and compiled language, the thereby specified transformations of the static program structure have to be performed at build-time anyway. Usually, it does not matter if transformations of the static structure, such as adding a field to some class, are performed manually by a programmer or automatically by an aspect weaver.

There might be one exception from this rule, though. For some very low-level data structure inside an OS, the actual order of members inside the class might be important, e.g. to exploit hardware-dependent caching effects. This is, if using aspects, typically not under the control of the programmer. It remains, however, questionable if such a situation is anyway a case for separation of concerns by aspects.

### *Dynamic Cross-cutting*

For code advice, the AOP concept to encapsulate dynamic cross-cutting, it depends on the actual pointcuts the advice is given for. *Conceptually*, code pointcuts are always evaluated at *runtime*, as they describe events in the running control flow. However, many code pointcuts can *actually* be evaluated at weave-time, as the event occurrences are unambiguously connected to specific positions in the code. This is obviously the case for `execution()` pointcuts, where interception always takes place before the first and/or after the last statement of a function. Many aspect languages also evaluate `call()` pointcuts at weave-time.[1] For such *statically evaluable pointcuts*, an ideal aspect weaver is able to completely inline the advice functionality into the target program. Hence, advice for statically evaluable pointcuts (as in variant 1 and variant 2 of the `KernelLock_x` aspect) is implementable without any AOP-related runtime overhead.

The `cflow()` pointcut function on the opposite is not always statically evaluable. On the implementation side, it requires in most cases to maintain some cflow-counter in the running program that has to be incremented, decremented, and tested at certain positions. Hence, advice for such *dynamic pointcuts* (as in variant 3 of the `KernelLock_x` aspect) induces, by principle, some runtime overhead.[2]

### *Join-Point Context*

Access to join-point-specific context in code advice might lead to some additional overhead. This is obvious, if the advice implementation uses context information that is specifically generated by the aspect weaver, such as the string representation of the affected function's name and signature. It is less obvious if the advice code accesses information that should be available in the current context anyway, such as the arguments passed to the affected function. As already pointed out, this information has to be provided through a generic interface—an additional level of abstraction that might induce some overhead. An ideal aspect weaver, however, should be able to optimize this in the woven code by substituting calls to the generic interface with direct context access. Furthermore, by analyzing the advice code, the aspect weaver can tailor the amount of provided context information to what is actually requested.

## 3.2 AspectC++

AspectC++ is an AOP-extension for the C++ language, specifically aimed for the application of AOP in resource-constrained environments such as embedded systems. A major goal of AspectC++ is cost efficiency in the generated code *without* compromising on the expressiveness of AOP concepts. On the language level, this is achieved by integrating AOP with the C++ philosophy of static typing and compile-time genericity[28]. On the tool level, AspectC++ follows a source-to-source weaving approach with generation of code patterns that (1) do not use "expensive" C++ language elements (such as RTTI or exceptions), and (2) can well be optimized by current C++ compilers. Another benefit of the source-to-source weaving approach is platform-independence.

### *Code Transformation*

The static AspectC++ weaver `ac++` is a source-to-source weaver that transforms AspectC++ code (C/C++ code with AspectC++ language elements) into C++ code. Advice is transformed into member functions of the corresponding aspect, which in turn are transformed into C++ classes (Figure 2). The generated C++ code can

[1] This has the implication that *indirect calls* through function pointers are not matched by `call()` advice.

[2] Strictly speaking, this overhead can be considered as "AOP-related" but not "AOP-faulted". A "hand-written" (tangled) implementation of the same functionality would induce similar overhead.

**a)** incrementer

| | advice | cycles abs | cycles Δ | stack abs | stack Δ | code abs | code Δ |
|---|---|---|---|---|---|---|---|
| | *tangled* | *4* | | *0* | | *4128* | |
| execution | before | *6* | *2* | *0* | *0* | *4128* | *0* |
| execution | after | *6* | *2* | *0* | *0* | *4128* | *0* |
| execution | around | *6* | *2* | *0* | *0* | *4128* | *0* |
| call | before | *6* | *2* | *0* | *0* | *4128* | *0* |
| call | after | *6* | *2* | *0* | *0* | *4128* | *0* |
| call | around | *6* | *2* | *0* | *0* | *4128* | *0* |

**b)** multiaspect

| | # aspects | cycles abs | cycles Δ | stack abs | stack Δ | code abs | code Δ |
|---|---|---|---|---|---|---|---|
| execution | 1 | *6* | | *0* | | *4080* | |
| execution | 2 | *5* | *-1* | *0* | *0* | *4080* | *0* |
| execution | 3 | *6* | *1* | *0* | *0* | *4096* | *16* |
| call | 1 | *6* | | *0* | | *4096* | |
| call | 2 | *5* | *-1* | *0* | *0* | *4096* | *0* |
| call | 3 | *6* | *1* | *0* | *0* | *4096* | *0* |

**c)** parameters, jp-api

| | tjp-> | cycles abs | cycles Δ | stack abs | stack Δ | code abs | code Δ |
|---|---|---|---|---|---|---|---|
| int B::bar(..) | *plain,* n=0 | *5* | | *16* | | *3968* | |
| int B::bar(..) | that() | *7* | *2* | *20* | *4* | *3968* | *0* |
| int B::bar(..) | target() | *8* | *3* | *20* | *4* | *3968* | *0* |
| int B::bar(..) | result() | *11* | *6* | *16* | *0* | *3968* | *0* |
| int B::bar(..) | *plain,* n=1 | *13* | | *24* | | *3968* | |
| int B::bar(..) | arg<0>() | *13* | *0* | *24* | *0* | *3968* | *0* |
| int B::bar(..) | *plain,* n=2 | *13* | | *32* | | *3984* | |
| int B::bar(..) | arg<1>() | *13* | *0* | *32* | *0* | *3984* | *0* |

**d)** dynamic pointcuts

| pointcut function | Δ cycles | Δ stack | Δ code | Δ data |
|---|---|---|---|---|
| cflow() | | | | *4* |
| enter/leave | *6* | *16* | *8* | |
| test | *12* | *52* | *56* | |
| that() | *10* | *24* | *128* | *50* |
| target() | *10* | *24* | *144* | *50* |

Overhead of AspectC++ features in terms of CPU time (*cycles*), dynamic memory consumption (*stack*, bytes), and static memory consumption (*code/data,* bytes of whole test application) . **a)** costs of incrementing a global counter either in the body of a function `void f()` (*tangled*) or by giving advice (*before/after/around* for *call/execution* join-points) to the same function. Δ denotes the difference to *tangled*. **b)** scaling of costs if *1–3* aspects give around advice to the same *call/execution* join-point. Δ denotes the difference to the previous line. **c)** costs of a member function call to int B::`bar( n )` (*n* = 0–2 `int` parameters) for which some advice is given that either does not use the join-point API (*plain*) or calls a join-point API function (*that()*, ...). Δ denotes the difference to the corresponding *plain* line. **d)** costs of the dynamic pointcut functions *cflow()*, *that()* and *target()*. Δ denotes the difference to a plain *execution()* pointcut.

[Linux, Pentium 3, `ac++` 1.0pre1, `g++` 3.3.5 (`-O3 -mpreferred-stack-boundary=2 -fno-align-functions -fno-align-jumps -fno-align-loops -fno-align-labels -fno-reorder-blocks -fno-prefetch-loop-arrays`), *cycles* averaged from 100 series of 1000 iterations ($\sigma < 0.1\%$ for all cases)]

**Table 1: AspectC++ $\mu$-benchmark results**

be compiled using any standard-compliant C++ compiler as back-end.

On the first sight, the output of `ac++` seems to be huge and somewhat over-complicated (Figure 2-c). The generated code is, however, not intended to be read by a human developer, but tweaked for flexibility and for an optimizing C++ back-end compiler. An advice method, for example, is not called directly, but routed over an forward-declared extra `invoke_xxx()` function (line 58). Only because of this detour it is possible to place the aspect definition *behind* all class definitions in the translation unit. Thereby, the aspect itself can utilize the affected classes, e.g. use them as types for state variables or method parameters.

## *Efficiency of the Generated Code*

To evaluate the overhead induced by the `ac++` generated code for various AspectC++ language features, we conducted a series of $\mu$-benchmarks. The condensed results are depicted in Table 1:

The **ground overhead** of applying advice to a parameterless function is very low (Table 1-a). On a Pentium 3, advice invocation takes only 2 cycles, independent from the type of advice (*before/after/around*), the join-point type (*call/execution*), and even the number of aspects giving advice to the join-point (Table 1-b). This is noteworthy, as in AspectJ, for instance, *around* advice induces significantly higher costs than *before/after* advice[12]. The size of the text segment (*code*) remains also stable, the increase by 16 bytes in one case was mainly caused by linker alignment of the affected section.

While advice for parameterless functions does not lead to additional stack costs, the stack space allocated by the compiler to pass call-by-value parameters is actually doubled. This becomes evident from the *plain* lines in Table 1-c, which represent the costs of a member function call with 0–2 `int` parameters for that some advice was given. (1-c numbers are not directly comparable with those from 1-a and 1-b, as they include the costs of the method call itself.) Each additional *4*-byte `int` parameter increases the absolute stack costs by *8* bytes. The reason turned out to be a limitation of the `g++` inliner/optimizer: Whenever a function is inlined, the compiler ensures call-by-value semantics by pushing an extra copy of all function parameters on the stack. In most cases, the parameter passing code is later replaced by the optimizer with direct register access, but the (now completely useless) stack reservation remains in the code. As `ac++` surrounds the original call with an additional wrapper (Figure 2, lines 32–42) this effect takes place here.

The **overhead to retrieve join-point specific context** is quite low. Compared to *plain* advice, only 0-6 extra cycles are consumed to provide access to context. Accessing context which is implicitly available at the join-point (such as argument and result values) does furthermore not induce any additional stack costs. The join-point context data generated by the waver for this purpose, such as the array of argument references (Figure 2, line 36), is later replaced by the optimizer with direct access to the referenced parameters. Only "extra" context, such as the pointer to the affected instance (`tjp->that()`) or the pointer to the target of the call (`tjp->target()`), requires additional stack space (4 bytes).

The results show furthermore the benefits of the context tailoring performed by `ac++`. The additional 4 stack bytes to store the pointer returned by `tjp->that()`, for instance, are only consumed if the advice code actually uses `tjp->that()`.

Compared to these numbers, the **overhead of dynamic pointcut functions** is relatively high (Table 1-d). As pointed out in Section 3.1, dynamic pointcuts induce by principle an overhead, because they can not be resolved completely at weave-time.

For the `cflow()` pointcut function, a runtime counter has to be maintained, which takes 4 additional bytes of data. For every pass through the observed control flow (*enter/leave)*, this counter has to be incremented/decremented, which takes 6 CPU cycles altogether. The pointcut evaluation itself, which basically is a null-test against the counter, (*test*) takes 12 CPU cycles at runtime. For both cases, the stack overhead is with 16 respectively 56 bytes higher than necessary. This seems to be again a problem with the `g++` inliner/optimizer, as useless stack reservations can be found in the object code.

The `that()` and `target()` pointcut functions (not to be mixed up with the equally named join-point API methods) yield all join-points in the dynamic control flow, where the *runtime type* of an object instance affected by advice (`that()`) / receiving a call (`target()`) matches a given type. They require a dynamic type test for evaluation, for which ac++ inserts a virtual function into all relevant classes. The additional 10 cycles of CPU and 24 bytes of stack are basically the costs of calling this virtual function. The more than 120 extra bytes of code and 50 additional bytes of data can be considered as a "worst-case-scenario", as `that()` and `target()` were applied to classes that did neither contain a virtual function table nor a constructor. If applied to a class hierarchy that already uses virtual functions, the costs would be lower, as the compiler has

**a) translation unit *Test.cpp***

```
volatile int global;
struct B {
  int bar(int a, int b) {...}
};
struct A {
  int foo(B &b) {
    return b.bar(47, 11);
} };
```

**b) aspect header *jpapi.ah***

```
extern volatile int global;

aspect jpapi {
  advice call("int B::bar(int, int)") : before () {
    global = *tjp->arg<1>();
} };
```

**c) woven translation unit *Test.acc***

```
1  class jpapi;
2  template <class JoinPoint>
3  inline void invoke_jpapi_a0_before (JoinPoint *tjp);
4
5  volatile int global;
6  struct B {
7    int bar(int a, int b) {...};
8  };
9
10 struct A {
11   int foo (B &b) {
12     return __call_foo_0_0 (this, &b, 47, 11);
13   }
14
15   struct TJP_foo_0_0 {
16     typedef int Result;
17     typedef ::A That;
18     typedef ::B Target;
19     ...
20     template <int I, int DUMMY = 0> struct Arg {...};
21     template <int DUMMY> struct Arg<0, DUMMY> {
22       typedef int Type;
23       typedef int ReferredType;
24     };
25     template <int DUMMY> struct Arg<1, DUMMY> {...}
26
27     void **_args;
28     template <int I> typename Arg<I>::ReferredType *arg () {
29       return (typename Arg<I>::ReferredType*) _args[I];
30   } };
31
32   static inline int __call_foo_0_0 (
33     ::A *srcthis, ::B *dstthis, int arg0, int arg1)
34   {
35     AC::ResultBuffer< int  > result;
36     void *args_foo_0[] = { (void*)&arg0, (void*)&arg1 };
37     TJP_foo_0_0 tjp;
38     tjp._args = args_foo_0;
39     invoke_jpapi_a0_before<TJP_foo_0_0> (&tjp);
40     ::new (&result) int  (dstthis->bar(arg0, arg1));
41     return (int &)result;
42   }
43 };  // struct A
44
45 class jpapi {
46 public:
47   static jpapi *aspectof () {
48     static jpapi __instance;
49     return &__instance;
50   }
51   template<class JoinPoint>
52   void __a0_before (JoinPoint *tjp) {
53     global = (int) tjp->template arg<1>();
54 } };
55
56 template <class JoinPoint>
57 inline void invoke_jpapi_a0_before (JoinPoint *tjp) {
58   ::jpapi::aspectof()->__a0_before (tjp);
59 }
```

Example for `ac++` code generation. **a)** Translation unit *Test.cpp* with classes `A` and `B`, `A::foo()` calls `B::bar()`. **b)** Aspect header *jpapi.ah*. The aspect `jpapi` gives *before* advice for all *calls* to `B::bar()` and uses the join-point API to retrieve the value of the second parameter passed to `B::bar()`. **c)** Woven translation unit *Test.acc*. The aspect `jpapi` has been transformed into a C++ class `jpapi` (lines 45–54), the advice into a member function `jpapi::_a0_before()` (line 52). The original call to `B::bar()` has been replaced by a call to the generated wrapper function `A::_call_foo_0_0` (line 12). Furthermore, a corresponding join-point class `TJP_foo_0_0` has been generated which encapsulates the join-point-specific static and dynamic context. In the wrapper function (lines 32–42), an instance `tjp` of TJP_foo_0_0 is created, initialized (lines 36–37), and passed as template argument to the before-advice invocation function `invoke_jpapi_a0_before()` (line 39), which retrieves the aspect instance and calls the generated advice method (line 58). Finally, the call to the original `B::bar()` is performed (line 40).

**Figure 2: Code Transformations by `ac++`**

not to generate support for a completely new virtual function table, but only extend an existing table.

## 3.3 Summary

Overall, AOP does not lead to an extra overhead that makes it *per se* unacceptable for system software development. Considering an ideal static aspect weaver for a statically typed language, only dynamic pointcut functions and the access of additional context information induce *inevitable* costs. Any other runtime costs in the code generated by a real weaver has therefore be considered as *evitable*, that is, caused by the actual weaver, but not by AOP in general. For the development of system software with AOP, one would intentionally avoid using "expensive" features, such as dynamic pointcut functions.

In combination with an optimizing C++ compiler, the static AspectC++ weaver `ac++` generates efficient code. In $\mu$-benchmarks, the overhead of simple before/after/around-advice for `call()` and `execution()` join-points is practically null, the overhead for accessing advice context is very low, even though in some cases the stack overhead turned out to be higher than necessary. As expected, the overhead induced by the not statically evaluable pointcut functions `cflow()`, `that()` and `target()` is significantly higher, especially regarding code and data size.

To get such results the compiler needs to provide some basic optimization capabilites. The most important are (1) embedding of functions explicitly marked as *inline* and (2) performing a local alias analysis to detect and remove unnecessary parameter copies. Without any optimization (especially function embedding), the resulting code would be much worse. This is, however, a quite unrealistic scenario, as such basic optimizations are provided by almost every C++ compiler. Additional measurements performed with the Intel and Microsoft C++ compilers (icc, Visual C++) support this. Due to space limitations we are not able to show and discuss any details here, however, the results turned out to be very similar to the g++ results[3]. Hence, a low overhead of aspects can be expected with other back-end compilers as well.

The AspectC++ language and weaver provide high-level AOP with

[3]Unfortunately even with respect to some of the unnecessary stack reservations discussed above.

```
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_..._PRI_INVERSION_PROTO_DEFAULT_PRI
20   ceiling = CYGSEM_..._PRI_INVERSION_PROTO_DEFAULT_PRI;
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 3: Code Example from eCos**

close-to-ideal efficiency in the generated code—at least on the level of $\mu$-benchmarks. The $\mu$-benchmark results are promising. They provide, however, only limited meaningfulness regarding the effects of a broader application of aspects. To judge this global effects, additional investigations in real system software are required.

# 4. USING ASPECTS IN THE ECOS OPERATING SYSTEM FAMILY

*eCos* [1] is a small and highly configurable operating system developed by *Cygnus Solutions* and now maintained and 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 embedded application domains. The eCos system itself is provided as a repository of various components, which are configured *statically* with a configuration tool called *eCosConfig*. Components are implemented in a mixture of C++, C, C-preprocessor and assembly code, the eCos kernel itself is implemented in C++. 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*. The final applications will be linked against this library.

To support a broad scale of applications in the strictly resource-constrained domain of embedded systems, *configurability* and *tailorability* are major goals of the eCos OS family. In the source code this becomes apparent in an extensive use of the preprocessor. Cross-cutting concerns manifest themselves in eCos as macro or function invocations. Configuration options manifest themselves as `#ifdef` blocks. Especially the latter are quite dominant in the component code. The `Cyg_Mutex` constructor (Figure 3) is a typical example. Only 4 of the 27 code lines are dedicated to the plain mutex implementation (lines 1, 3–4, 27). The remaining parts are occupied by one *cross-cutting concern* (namely tracing of function entries and exits, lines 2 and 26) and four *configuration options* (namely the different versions of the priority inversion protocol to use, lines 5–25).

## 4.1 Case Study Overview

The goal of this case study is to gain insights regarding the still open question if AOP is suitable for separation of concerns in system software with respect to costs. For this purpose, we analyzed the effects of using aspects by *refactoring* and *extending* the eCos kernel. The case study is therefore divided into two parts:

For the **refactoring part**, we extracted cross-cutting concerns and configuration options from the eCos kernel sources and merged their implementation snippets into aspects. This part specifically targets the costs question by comparing the original scattered solution of configuration options with an functionally identical aspect-based solution. The refactoring of cross-cutting concerns targets furthermore at the scalability question, as the resulting aspects affect a high number of join-points. Overall, we refactored three cross-cutting concerns and twelve configuration options from the eCos thread package (Table 2).

For the **extension part**, we used aspects to implement a *new* configuration option that has not been available before in the eCos kernel. This part targets at the relationship between system design and it's influence on the costs of aspect-based implementations for unanticipated extensions.

## 4.2 Refactoring Cross-cutting Concerns

The following three concerns were identified as highly cross-cutting the eCos kernel code base:

**tracing:** To observe the control flow through the system, entrances to and exits from system functions are recorded. Furthermore, it is possible to track values of function arguments, local variables and function results.

**interrupt synchronization:** In order to guarantee the consistency of operating system data structures, most kernel functions must run mutual exclusive to interrupt handlers. This is achieved by explicitly enabling/disabling the propagation of interrupts in all relevant functions.

**kernel instrumentation:** Inside the eCos kernel, special macros are employed to count the occurrences of various events, such as thread creation or mutex locking.

In the original code base, these three cross-cutting concerns account for more than 680 lines (>13%) of the kernel sources, spread over 23 source files (Table 2-a, column *original*). As *tracing* is a typical development aspect,[4] we focus in the following on the refactored implementation of the more OS-specific and cost-critical *kernel instrumentation* and *interrupt synchronization* concerns.

### 4.2.1 Interrupt Synchronization

eCos uses a typical two-level interrupt handling scheme. Hardware IRQs are handled by *interrupt service routines (ISRs)*, which usually just register a *deferred service routine (DSR)* to perform the actual IRQ handling. DSR execution is delayed, while the active thread is in a synchronized kernel function. Delayed DSRs are propagated before thread dispatching and when the active thread leaves the kernel. DSR propagation has to be disabled explicitly by calling `Cyg_Scheduler::lock()` at the begin every kernel function that requires synchronization. It has to be re-enabled by explicitly calling `Cyg_Scheduler::unlock()` at each exit point of such function. Overall, this sums up to 187 `lock()`/`unlock()` calls (Table 2-a, column *original*), spread over 80 kernel functions.

The refactored version of the *interrupt synchronization* concern is quite simple:

[4] It is moreover already the most cited (not to say over-featured) example of a cross-cutting concern.

**a)** cross-cutting concerns

| concern | original loc # | original loc % | refactored loc # | refactored loc % | refactored join-points # | refactored join-points % |
|---|---|---|---|---|---|---|
| *total* | *5205* | *100* | *4527* | *100* | *931* | *100* |
| *total functional* | *4520* | *86.8* | *4423* | *97.7* | | |
| tracing | *336* | *6.5* | *4* | *0.1* | *632* | *67.9* |
| instrumentation | *162* | *3.1* | *0* | *0.0* | *139* | *14.9* |
| irq syncchronization | *187* | *3.6* | *0* | *0.0* | *160* | *17.2* |
| *total crosscutting* | *685* | *13.2* | *4* | *0.1* | | |

**b)** thread configuration options

| configuration option CYGVAR_KERNEL_THREADS... | original #ifdef blocks | refactored #ifdef blocks | refactored join-points code | refactored join-points intro |
|---|---|---|---|---|
| *total* | *39* | *3* | *21* | *29* |
| ..._NAME | *3* | *1* | *2* | *2* |
| ..._LIST | *4* | *0* | *2* | *6* |
| ..._STACK_LIMIT | *7* | *0* | *3* | *4* |
| ..._STACK_CHECKING | *6* | *1* | *8* | *1* |
| ..._STACK_MEASUREMENT | *2* | *0* | *2* | *2* |
| ..._DATA | *3* | *0* | *1* | *8* |
| ..._DESTRUCTORS | *3* | *0* | *1* | *3* |
| ..._DESTRUCTORS_PER_THREAD | *11* | *1* | *2* | *3* |

**c)** mutex configuration options

| configuration option CYGSEM_KERNEL_SYNC_MUTEX... | original #ifdef blocks | refactored #ifdef blocks | refactored join-points code | refactored join-points intro |
|---|---|---|---|---|
| *total* | *34* | *6* | *22* | *26* |
| ..._PROTOCOL | *14* | *4* | *9* | *11* |
| ..._PROTOCOL_INHERIT | *5* | *1* | *4* | *3* |
| ..._PROTOCOL_CEILING | *10* | *1* | *4* | *6* |
| ..._PROTOCOL_DYNAMIC | *5* | *0* | *5* | *6* |

**Table 2: Refactored Concerns in the eCos Kernel**

```
aspect int_sync {
  pointcut sync() = execution(...) // kernel calls
           || construction(...)    // to sync
           || destruction(...);

  // advice kernel code to call lock() / unlock()
  advice sync() : before() {
    Cyg_Scheduler::lock();
  }
  advice sync() : after() {
    Cyg_Scheduler::unlock();
  }
  // A new thread starts with a lock value of 0
  advice execution(
  "%Cyg_HardwareThread::thread_entry(...)") : before() {
    Cyg_Scheduler::zero_sched_lock();
  }
  ...
};
```

Before/after advice is used to superimpose the invocation of `Cyg_Scheduler::lock()`/`unlock()` into the execution[5] of all kernel functions that require synchronization. Overall, 160 code join-points are affected (Table 2-a, column *refactored*). The number of join-points is below the number of original function calls, as in the original some functions contained more than one exit point, each with a call to `Cyg_Scheduler::unlock()`, while after advice implicitly affects all exit points of a function.
Additionally (not shown above), all functions that implement locking itself have been refactored from the scheduler class into a set of introductions, so that the `int_sync` aspect provides a complete encapsulation of the *interrupt synchronization* concern.

### 4.2.2 Kernel Instrumentation

The eCos kernel optionally maintains a set of counters for the occurrence of various events, such as thread suspension, mutex locking or interrupt processing. For each counter and event type, a specific preprocessor-macro is provided. 162 invocations (Table 2-a, column *original*) of these macros are distributed over the whole kernel source base.
In the refactored version, the invocation of the particular macro is given as advice to the affected kernel functions such as follows:

```
aspect kernel_instrument_mutex {
  ...
  advice execution("% Cyg_Mutex::lock(...)") : after() {
    if(*tjp->result()) {
      CYG_INSTRUMENT_MUTEX(LOCKED,tjp->that(),0);
  } }
  advice call("% Cyg_Thread::wake(...)")
         && within("% Cyg_Mutex::unlock(...)") : after() {
    CYG_INSTRUMENT_MUTEX(WAKE,tjp->that(),tjp->target());
  }
  ...
};
```

In the advice bodies, the join-point API is used to retrieve the object instances involved in a particular event. Overall, 13 aspects with 85 code advice definitions such as above affect a total of 139 join-points (Table 2-a). Again, the number of affected join-points is below the number of original macro invocations due to multiple exit points in some functions.

## 4.3 Refactoring Configuration Options

For the sake of configurability and tailorability, eCos offers a high number of *configuration options* that may or may not be selected in *eCosConfig* if generating a concrete eCos system. The kernel optionally provides support for thread-local storage (CYGVAR_KERNEL_THREADS_DATA), the execution of destructors of global object instances (CYGVAR_KERNEL_THREADS_DESTRUCTORS) and much more. Table 2-b/c lists some of the available options. As already shown by the `Cyg_Mutex` constructor example (Figure 3), the implementation of configuration options is embedded into the base code and activated by means of conditional compilation, which leads to well-known phenomenon of "`#ifdef`-hell". For the 12 analyzed thread and mutex configuration options overall 73 `#ifdef`-blocks are spread over the source code (39 for *thread configuration options* / 34 for *mutex configuration options*, Table 2-b/c).
In the refactored version, each configuration option has been encapsulated into a single aspect that superimposes the functionality into the base component. Additional member functions and state variables specific for a certain configuration option are applied by introductions, option-specific behavior is applied by code advice. Overall, 98 join-points are affected (21/22 code join-points and 29/26 introductions). The number of join-points is above the number of `#ifdef`-blocks in the original (73), as some `#ifdef`-blocks embrace the definition of multiple identifiers (state variables, member functions), while in the refactored version each identifier is given by a separate introduction. Some few `#ifdef` blocks (3 for *thread configuration options* / 6 for *mutex configuration options*), all caused by inter-feature dependencies, have not been resolved completely, but were simply moved into the corresponding aspect implementation. While it had been possible to remove these `#ifdef` blocks as well, we went on without further refactorings, as this had had some impact on the comparability of the AOP version with the original eCos regarding resource utilization.

## 4.4 Cost Analysis

We have analyzed memory and performance effects of using AOP in eCos by comparing the refactored and extended versions of eCos with the original eCos for various configurations.

[5] Some kernel functions are actually class con-/destructors, for which the `con-/destruction()` pointcut function is used instead of `execution()`.

| | a) thread | | b) mutex | | c) semaphore | |
|---|---|---|---|---|---|---|
| | cyg_thread_... | | cyg_mutex_... | | cyg_semaphore_... | |
| 1 | ..._create() | *215* | ..._init() | 52 | ..._init() | *62* |
| 2 | ..._resume() | *327* | ..._lock() | *47* | ..._post() | *50* |
| 3 | ..._resume() | *127* | ..._unlock() | *22* | ..._wait() | *723* |
| 4 | ..._yield() | *274* | ..._try_lock() | *46* | ..._trywait() | *416* |
| 5 | ..._exit() | *354* | ..._try_lock() | *49* | ..._trywait() | *44* |
| 6 | ..._yield() | *77* | ..._lock() | *381* | ..._wait | *46* |
| 7 | ..._resume() | *91* | ..._unlock() | *429* | ..._post() | *22* |
| 8 | ..._kill() | *102* | ..._destroy() | *17* | ..._destroy() | *19* |
| 9 | ..._suspend() | *336* | | | | |
| 10 | ..._suspend() | *96* | | | | |
| 11 | ..._suspend() | *53* | | | | |
| 12 | ..._resume() | *63* | | | | |
| 13 | ..._resume() | *115* | | | | |
| 14 | ..._delete() | *38* | | | | |
| ∑ | [cycles] | *2268* | [cycles] | *1043* | [cycles] | *1382* |

Sequence of system calls performed by the *thread*, *mutex*, and *semaphore* test applications. Numbers denote *CPU cycles* taken by the particular system call. CPU cycles differ among several invocations of the same call due to different context-switch operations the kernel has to perform internally. **a) thread:** three threads activate each other in turns using operations from the kernel thread API. **b) mutex / c) semaphore:** two threads synchronize each other using kernel mutex/semaphore objects.

[eCos original, base configuration, Pentium 3, Cycles averaged from 10 series of 1000 iterations ($\sigma < 0.1\%$ for all cases)]

**Table 3: Test applications**

### 4.4.1 Setup

13 different eCos configurations, each in an AOP and an original version, were generated and built, among them:

- the eCos basic configuration (*_base*, no additional features), to compare the effects regarding *interrupt synchronization*
- a configuration with additional support for instrumentation (*_instrumentation*), to compare the effects regarding *kernel instrumentation*
- several other configurations, each with one selected extra feature, to compare the effects regarding the refactored *configurable options*

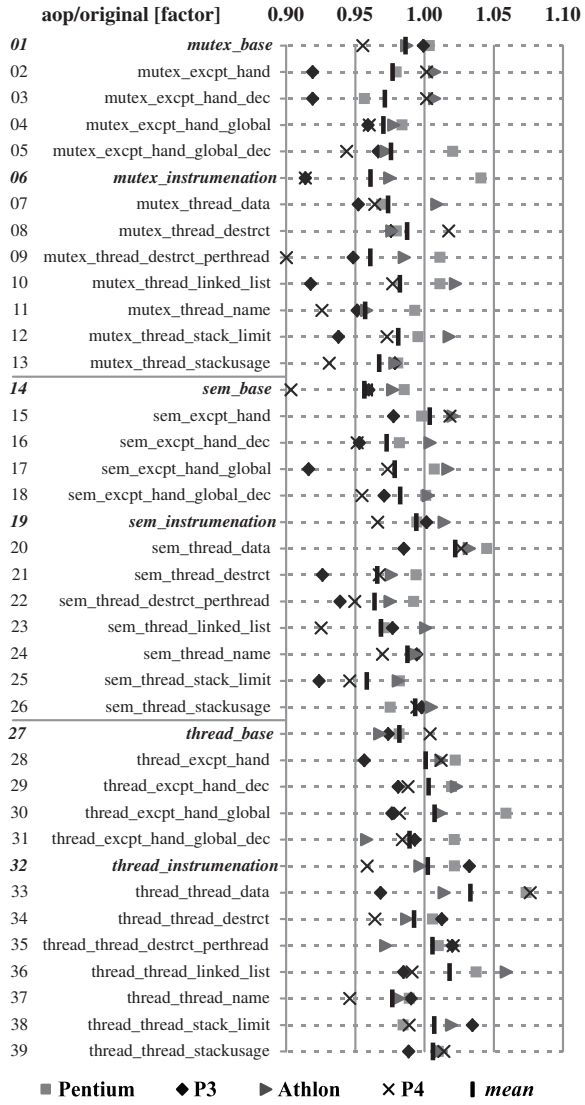
Each of the resulting 26 variants of the *eCos-library* was linked with the same set of 3 multi-threaded test applications, which specifically use the affected parts of the eCos kernel. The test applications themselves are quite simple, their threads just invoke a sequence of system calls and do not perform further calculations. Table 3 depicts the sequence of system calls performed by each test in conjunction with the number of CPU cycles taken by each system call on a basic eCos system.

The AOP versions were woven with `ac++ 1.0pre1`, all versions were compiled and linked with `gcc 3.3.5`, using a set of optimization flags that favors code size over performance[6].

### 4.4.2 Runtime Costs

Runtime costs (*cycles*) were measured in the running test applications and averaged from 10 series of 1000 iterations to reduce caching effects. With a relative standard derivation $\sigma < 0.1\%$ for all series, the results can be considered as stable. Besides the P3

[6] `-O3 -mpreferred-stack-boundary=2 -fno-align-functions -fno-align-jumps -fno-align-loops -fno-align-labels -fno-reorder-blocks -fno-prefetch-loop-arrays`
(A simple `-Os` actually causes higher code size, as it prevents embedding of the `ac++` generated wrapper functions which are join-point specific and therefore used only once.)




Relative performance loss/gain of the refactored (*aop*) version of eCos compared to the *original.* Each of the 3 test applications (*mutex*, *sem[aphore]*, *thread*) has been analyzed with respect to 13 different eCos configurations, which results in 39 test cases, measured on 4 different CPU types. *Mean* denotes the average over all CPU types.

**Figure 4: Per-test-case relative CPU**

CPU, which was the target platform for this study, additional measurements with the same binary images were performed on Pentium, Athlon and P4 CPUs. The results are depicted in Figure 4 by the relative AOP runtime cost factor ($\frac{aop}{original}$) for each test case, configuration, and CPU type[7]:

- Over all test cases, the AOP cost factor does vary. The highest variation is in the numbers of the P4 CPU ([0.90, 1.07], $\sigma$=0.04), the distribution of all other CPU types is in between (Pentium: [0.96, 1.07], $\sigma$=0.03, P3: [0.91, 1.03], $\sigma$=0.03, Athlon: [0.96, 1.06], $\sigma$=0.02). For an averaged CPU (*mean*), the cost factor is distributed with [0.96, 1.02], $\sigma$=0.02.

[7] Depicted results were measured with enabled CPU caches. Additional measurements with caching disabled resulted in a much higher standard derivation (probably caused by DRAM-Timing and bus load effects), but basically the same average cost factors.

- Even for each single test case, the AOP cost factor varies noticeable among the different CPU types. The maximum variation is [0.91, 1.04], σ=0.06 (test case 06: *mutex_instrumentation*). The minimum is [0.99, 1.01], σ = 0.01, (test case 39: *thread_thread_stackusage*). At average, the CPU-dependent cost factors are distributed with a standard derivation of σ = 0.03.
- For no single test case, AOP can be considered as clearly beneficial (factor < 0.98) or disadvantageous (factor > 1.02) for all CPU types.
- Over all test cases, a slight tendency towards a beneficial influence of AOP can be observed for the P3 and P4 CPUs, for which the average AOP cost factor is 0.97. For the Pentium and Athlon CPUs, however, no such effect can be found (factor 1.00)—as well as for the averaged CPU (*mean*), for which the factor is 0.99.

We conclude from these numbers that a general effect of AOP on the runtime costs of eCos can not be ascertained. The observed variation has to be attributed to CPU-internal "performance noise", probably caused by mechanism such as branch prediction, instruction reordering and memory alignments. The distribution of the per-CPU cost factors, which is stable and reproducible for each single test case, but highly differs between different test cases, is a clear indicator for such effects. It has furthermore to be considered that the actual amount of cycles consumed by most eCos kernel functions is quite low (Figure 3), which increases the relative effects of such CPU-internal variability. On the other hand, a potential AOP-related overhead should have become evident in the results, as it would affect every CPU type.

To gain additional evidence, we analyzed and compared the generated machine code of the aspect-based and original eCos in the *_base* configuration (test cases 01, 13, 27). From the 28 analyzed kernel functions that are called directly or indirectly from the test applications and that are affected by *interrupt synchronization*, 19 functions turned out to be actually *identically* in the machine code[8]. Eight functions turned out to be minimal different with respect to the order of instructions and differences in sub-sequences of up to 4 instructions. While we had the impression that in these cases the code generated for the AOP versions is slightly better (especially regarding instruction ordering), we were not able to find any general patterns that would support this impression. We found major differences between the AOP and the original version in a single functions only, namely `Cyg_Scheduler_Implementation::add_thread()`. The differences are caused by the back-end compiler, which embeds some member functions that are not explicitly marked as to inline only in the AOP version. With the synchronization code removed into aspects, the size of these (relatively small) functions has probably fallen under a compiler-internal threshold causing the extra inlining. This effect seems to be even higher in the other configurations. It can, however, not be considered as a typical effect of using AOP and had no significant influence on the measured runtimes. Overall, AOP does neither lead to better nor to worse runtime.

### 4.4.3 Memory Costs

The additional inlining performed by the compiler in the AOP version can also be observed in the memory overhead. Table 4 shows the differences between the AOP and the original eCos.

In almost all cases, the AOP versions induce some *ROM* overhead.

[8] if ignoring differences regarding symbol addresses and function-wide register allocation (such as always using `ecx` instead of `edx` and vice versa)

| | [bytes] | ROM | | RAM | | stack | |
|---|---|---|---|---|---|---|---|
| | | Δ | % | Δ | % | Δ | % |
| 01 | *mutex_base* | *101* | *0.6* | *0* | *0.0* | *24* | *2.0* |
| 02 | mutex_excpt_hand | *55* | *0.3* | *0* | *0.0* | *24* | *2.0* |
| 03 | mutex_excpt_hand_dec | *-35* | *-0.2* | *0* | *0.0* | *24* | *2.0* |
| 04 | mutex_excpt_hand_global | *92* | *0.5* | *0* | *0.0* | *24* | *2.0* |
| 05 | mutex_excpt_hand_global_dec | *85* | *0.5* | *0* | *0.0* | *24* | *2.0* |
| 06 | *mutex_instrumenation* | *543* | *3.0* | *0* | *0.0* | *4* | *0.3* |
| 07 | mutex_thread_data | *74* | *0.4* | *0* | *0.0* | *28* | *2.3* |
| 08 | mutex_thread_destrct | *74* | *0.4* | *0* | *0.0* | *28* | *2.3* |
| 09 | mutex_thread_destrct_perthread | *224* | *1.3* | *0* | *0.0* | *24* | *2.0* |
| 10 | mutex_thread_linked_list | *88* | *0.5* | *0* | *0.0* | *8* | *0.6* |
| 11 | mutex_thread_name | *146* | *0.9* | *0* | *0.0* | *16* | *1.3* |
| 12 | mutex_thread_stack_limit | *311* | *1.8* | *0* | *0.0* | *24* | *2.0* |
| 13 | mutex_thread_stackusage | *341* | *2.0* | *0* | *0.0* | *24* | *2.0* |
| 14 | *sem_base* | *91* | *0.5* | *0* | *0.0* | *24* | *2.0* |
| 15 | sem_excpt_hand | *51* | *0.3* | *0* | *0.0* | *24* | *2.0* |
| 16 | sem_excpt_hand_dec | *-54* | *-0.3* | *0* | *0.0* | *24* | *2.0* |
| 17 | sem_excpt_hand_global | *88* | *0.5* | *0* | *0.0* | *24* | *2.0* |
| 18 | sem_excpt_hand_global_dec | *81* | *0.5* | *0* | *0.0* | *24* | *2.0* |
| 19 | *sem_instrumenation* | *588* | *3.1* | *0* | *0.0* | *0* | *0.0* |
| 20 | sem_thread_data | *70* | *0.4* | *0* | *0.0* | *28* | *2.3* |
| 21 | sem_thread_destrct | *70* | *0.4* | *0* | *0.0* | *28* | *2.3* |
| 22 | sem_thread_destrct_perthread | *220* | *1.3* | *0* | *0.0* | *24* | *2.0* |
| 23 | sem_thread_linked_list | *84* | *0.5* | *0* | *0.0* | *8* | *0.6* |
| 24 | sem_thread_name | *142* | *0.8* | *0* | *0.0* | *16* | *1.3* |
| 25 | sem_thread_stack_limit | *307* | *1.8* | *0* | *0.0* | *24* | *2.0* |
| 26 | sem_thread_stackusage | *337* | *2.0* | *0* | *0.0* | *24* | *2.0* |
| 27 | *thread_base* | *104* | *0.6* | *0* | *0.0* | *12* | *0.7* |
| 28 | thread_excpt_hand | *58* | *0.3* | *0* | *0.0* | *12* | *0.7* |
| 29 | thread_excpt_hand_dec | *-32* | *-0.2* | *0* | *0.0* | *12* | *0.7* |
| 30 | thread_excpt_hand_global | *95* | *0.6* | *0* | *0.0* | *12* | *0.7* |
| 31 | thread_excpt_hand_global_dec | *88* | *0.5* | *0* | *0.0* | *12* | *0.7* |
| 32 | *thread_instrumenation* | *563* | *3.1* | *0* | *0.0* | *-4* | *-0.2* |
| 33 | thread_thread_data | *77* | *0.4* | *0* | *0.0* | *12* | *0.7* |
| 34 | thread_thread_destrct | *77* | *0.4* | *0* | *0.0* | *12* | *0.7* |
| 35 | thread_thread_destrct_perthread | *227* | *1.3* | *0* | *0.0* | *12* | *0.7* |
| 36 | thread_thread_linked_list | *91* | *0.5* | *0* | *0.0* | *-12* | *-0.6* |
| 37 | thread_thread_name | *149* | *0.9* | *0* | *0.0* | *0* | *0.0* |
| 38 | thread_thread_stack_limit | *333* | *1.9* | *0* | *0.0* | *12* | *0.7* |
| 39 | thread_thread_stackusage | *344* | *2.0* | *0* | *0.0* | *12* | *0.7* |

Relative overhead of the AOP version compared to the original eCos ($\Delta = aop - original$) regarding dynamic memory utilization (*stack*, accumulated from all threads) and static memory utilization (*ROM* = code + data + rodata, *RAM* = data + bss). Dynamic memory utilization was measured in the running test applications. Static memory utilization was retrieved off-line and byte-exact from the linker map files.

**Table 4: AOP Memory Overhead**

This overhead, 0.9% at average, is caused by larger code sections. The *instrumentation* test cases (06, 19, 32) show with more than 500 extra code bytes (an overhead of 3%) the maximum difference. Most of this overhead can again be attributed to inlining effects. In the original eCos, the compiler does not embed the final call to `Cyg_Scheduler::unlock()` if a function is affected by *instrumentation*. Instead of, a `jmp` statement to a shared copy of `Cyg_Scheduler::unlock()` is generated. In the AOP version, the call is always embedded and no code sharing takes place, regardless of *instrumentation* being enabled or not.

Generally, the AOP versions show also some stack overhead, at average 1.3% more stack bytes are used. This is mainly caused by (unnecessary) reservations for call-by-value parameters, as explained in Section 3.2. The generally higher overhead of the *mutex* and *semaphore* test cases (compared to *thread*) can be explained by this effect as well, as the `Cyg_Mutex` and `Cyg_Semaphore` opera-

```
1  aspect kernel_stack {
2    // case a) dynamic join-points
3    pointcut stack_switch() =
4    execution(kernel()) && !cflow(within(kernel()));
5
6    // case b) static join-points
7    pointcut stack_switch() =
8    call(kernel()) && within(user_api());
9
10   advice stack_switch() : around(){
11     HAL_TO_KERNEL_STACK(Cyg_Thread::self());
12     tjp->proceed();
13     HAL_FROM_KERNEL_STACK();
14 } };
```

**Figure 5: The `kernel_stack` Aspect**

tions lead to higher call-depths inside the kernel.
According to the $\mu$-benchmark results in Section 3.2, some additional stack usage can also be expected if the advice code accesses context information via the join-point API. Surprisingly, the *instrumentation* test cases (06, 19, 32) which make extensive use of the join-point API, show a very low stack overhead. The reduced call-depth (due to inlining) seems to compensate the AOP-induced overhead in these cases.
No differences can be observed regarding *RAM* utilization. As our aspects are stateless, the linker seems to be able to omit even the aspect instances from the final image.

## 4.5 Applying Unanticipated Extensions

AOP is frequently considered suitable for *unanticipated* changes[13]. As aspects can superimpose additional behavior without having to touch the existing software structure, it should in principle be possible to use aspects to bring in unanticipated change. For the domain of system software, however, it is not (only) the question if it is *possible* to implement unanticipated extensions by means of aspects, but if it is possible to implement them *efficiently*.
As pointed out in Section 3, aspects do not lead to an intrinsic overhead, if their pointcuts are resolvable statically. On the other hand, an overhead is inevitable if advice has to be given for dynamic join-points. Hence, an overhead of using aspects for unanticipated extensions depends on the availability of "the right static join-points". To analyze the effect of having to use dynamic join-points instead, we extended eCos by a new *kernel stack* feature. It is implemented by an aspect without any modifications in the existing sources.

### 4.5.1 The Kernel Stack Aspect

The general idea is to reserve a dedicated part of a thread's stack exclusively for the kernel, which guarantees that no stack overflow error can occur while executing kernel code. For this purpose, the stack pointer is switched to the part reserved for the kernel, whenever a thread enters the kernel. It is switched back to the application stack immediately before the kernel is left, as sketched by the following advice definition:

```
advice ... : around() {
  HAL_TO_KERNEL_STACK( Cyg_Thread::self() );
  tjp->proceed();
  HAL_FROM_KERNEL_STACK();
};
```

The platform-specific `HAL_TO_KERNEL_STACK` macro performs the actual stack switch. Afterwards, the join-point API method `proceed()` is used, which calls the original function and thereby copies all parameters from the join-point context to the now active kernel stack.[9]

[9]This is, of course, platform-dependent. The `proceed()` method is inlined and accesses (on x86) the join-point context via the frame pointer, which is not modified by `HAL_TO_KERNEL_STACK`.

| kernel-stack cost factor [cycles] | dynamic join-points | static join-points |
|---|---|---|
| mutex_base | *2.24* | *1.05* |
| sem_base | *1.95* | *1.07* |
| thread_base | *1.88* | *1.09* |

Relative overhead of the *kernel-stack* feature if using *dynamic join-points* or *static join-points*.

**Table 5: *Kernel Stack* CPU Overhead**

In eCos, most kernel functions are not only invoked from application level, but also used internally, that is, called by other kernel functions. As a consequence, it is not possible to give a statically resolvable pointcut expression that yields all join-points where a thread enters the kernel. We have to distinguish at runtime, if a kernel function is executed in the control flow of an application or of another kernel function. This basically leads to a pointcut expression such as `execution(kernel()) && !cflow(within(kernel()))` (Figure 5, case a).[10].

### 4.5.2 Costs of Dynamic Join-Points

The *kernel stack* feature is a highly cross-cutting concern, as it affects the execution of *every* kernel function. Not surprisingly, this leads to a noticeable overhead. Table 5 (column *dynamic join-points*) lists the cost factor of the `kernel_stack` aspect, compared to the original eCos for the three *_base* test-cases. The cost factor is between 1.88 and 2.24 (average 2.02).
To evaluate, which part of these costs are induced by the concern implementation itself (the actual stack switch) and which part is induced by the need to use dynamic join-points, we compared the effects with a *kernel stack* implementation for an alternative eCos. This alternative eCos offers a thin *user access* layer on top of the kernel, which is exclusively used by application code to invoke kernel functionality. The user access layer consists of a set of inline wrapper functions and classes and does not lead to any extra overhead. However, it makes it possible to distinguish *statically* between application $\mapsto$ kernel and kernel $\mapsto$ kernel calls (Figure 5, case b). As shown in Table 5 (column *static join-points*), the overhead of the `kernel_stack` aspect is significantly lower in this case, with cost factors between 1.05 and 1.09 (average 1.07). Thus, most of the overhead of the first `kernel_stack` implementation was actually caused by the fact that advice had to be given for dynamic join-points.

# 5. DISCUSSION

In our eCos study, the AOP version showed at average a 0.9% higher code size and a 1% better performance. These very small effects can both be explained by additional inlining the compiler performed in the AOP version: Due to the extraction of *synchronization* and *instrumentation* code from almost each function, the remaining code size of some functions seems to have fallen under an internal threshold which caused the compiler to consider them for inlining in the AOP version, but not in the original version. This, however, has to be considered as a local phenomenon which can not be generalized. The machine code similarity for most of the affected functions rather confirms the $\mu$-benchmark results from Section 3.2, which show that for simple before/after/around advice no considerable overhead has to be expected.

[10]AspectC++ currently does not support a per-thread `cflow()`, which can, however, easily be simulated by other language mechanisms. For the actual `kernel_stack` implementation a counter was introduced into the thread control block and internally used as cflow variable.

All this leads to the conclusion that the AOP-based application of cross-cutting concerns and configuration options does not induce noticeably different costs than a scattered and tangled implementation. This even holds, if the resulting aspects affect a high number of join-points and exploit join-point specific context information. In the following we discuss potential methodical issues of our study and under what circumstances the obtained results can be generalized to the domain of operating systems and system software in general.

## 5.1 Methodical Issues

### *Evaluation Approach*

Results were obtained by combining $\mu$-benchmarks with a larger case-study based on refactoring. The refactoring approach makes sure that (1) the results apply to real-world software, and that (2) the *implementation* of the concerns is identical in the AOP and original version. Both versions differ only regarding the *mechanism* used to apply the concern implementations. Thus, the measured results actually depict the net overhead of using aspects instead of tangled code and are not caused by a "smarter implementation". As the results from the case study are confirmed by the $\mu$-benchmark results and vice versa, both can be considered as plausible.

### *Evaluation Targets*

Results were conducted by refactoring concerns from the eCos operating system using AspectC++ and measuring the effects with a specific set of test applications. This, however, should not limit their general significance:

**eCos** can be considered as a most demanding test subject regarding the costs of aspects. Designed for and widely accepted in the very resource-thrifty domain of embedded systems, eCos has certainly been optimized for runtime and memory efficiency. Most system calls take only a few cycles (see Table 3), the typical kernel image size is a only few KB. An AOP-induced overhead, if any, should become more evident in this system than in "big" operating systems such as Windows or Linux.

**AspectC++** was basically a natural choice. Developed as a general-purpose AOP language for the C/C++ domain, it offers all concepts that are typical for this paradigm. In principle, similar results should be reachable with other AOP approaches as well. As pointed out in Section 3.1, most AOP concepts do not induce an inevitable overhead.

**Test applications** chosen for cost measurements can not be considered as "typical applications". They have been designed specifically for the measurements and spend most of their time executing the affected kernel code. Regarding the costs of aspects, this has to be considered as more demanding than using "real" applications that only occasionally call kernel functions. Thus, an AOP-induced overhead, if any, should become evident with this setup.

### *Significance of the Refactored Concerns*

Results are naturally based on a *selection* of concerns, namely two cross-cutting concerns and twelve configuration options. The selected concerns offer, however, a high coverage, as they (1) represent a cross-section of real-world and cost-critical concerns from the domain that (2) affect a high number of join-points, and (3) represent the typical classes of concerns that are generally addressable by AOP:

**Interrupt synchronization** is a typical example of a *symmetrically cross-cutting* concern. The same advice functionality is given to a high number of join-points. With 160 affected code join-points (Table 2-a), the `int_synch` aspect qualifies well for evaluation of the costs AOP may induce for symmetrically cross-cutting and performance-critical concerns.

**Kernel instrumentation** is a typical example of an *asymmetrically cross-cutting* concern. This means that not only one or two advice definitions are applied to a high number of join-points (as in the `int_synch` aspect), but many different advice definitions affect only a relatively small number of join-points each. With 139 affected code join-points (Table 2-a) given by 13 aspects with 85 code advice definitions, the aspect-based implementation qualifies well for evaluation of the costs AOP may induce for asymmetrically cross-cutting concerns.
*Kernel instrumentation* is, moreover, an example of a *highly context-dependent* concern, as not only events (such as unlocking a mutex), but also the involved kernel objects (such as the mutex and thread instances) are reported, which leads to an intensive use of the join-point API. Hence, *kernel instrumentation* also qualifies well for evaluation of the costs AOP may induce for context-dependent concerns.

**Configuration options** are a typical example of *scattered* concerns. Such concerns do not cross-cut the whole source base, but some few components only. Intended to be configurable in eCos, they were *conceptually* already well separated. Nevertheless, their *implementation* is spread out over multiple methods and classes. The only reason to scatter the implementation of an otherwise clearly identified concern is that a separated implementation is either not possible (which is an indicator for a cross-cutting concern) or not *efficiently* possible. Hence, an aspect-based implementation of configuration options qualifies well to evaluate the efficiency of AOP-generated code.

Hence, similar results should be achievable with other concerns from the domain of system software as well.

## 5.2 General Issues

### *Applicability to Existing System Software*

Refactoring cross-cutting concerns and configuration options into aspects was relatively easy in eCos, as (1) most concerns were already separated on the conceptual level, intermingled in the implementation only for technical reasons, and (2) the remaining parts of the implementation already offered a reasonable separation of concerns with a fine-grained design and implementation. This results in a high number of potential join-points with strong semantics.
In monolithic systems with a high coupling among concern implementations, such as Linux[30], the number of potential join-points is probably much lower and join-points are semantically ambiguous. In this case it might be more difficult or even impossible to implement concerns such as *interrupt synchronization* or *kernel instrumentation* by aspects without performing additional refactoring in the primary structure of the target systems. For such scenarios, special-purpose AOP approaches as suggested by Fiuczynski[16] and Åberg[2] might be more appropriate.
The number of potential join-points could also be increased by a join-point model that supports join-points on the statement and expression level, such as loops and local variable access[17]. This clearly remains a topic for further research, even though we doubt that most of these join-points would offer enough semantics for robust aspect implementations.

### *The Role of Design*

Ambiguous join-points can also lead to a situation where it is possible, but not efficient to use aspects. The ambiguity of the available static join-points can often be resolved by exploiting runtime context information and dynamic pointcut functions. However, as demonstrated by the `kernel_stack` aspect, dynamic pointcut functions induce additional costs, especially if the concern is highly cross-cutting. Generally, the use of dynamic pointcut functions should be avoided in system software.

On the other hand, the costs of even unanticipated extensions can be very low, if the semantically important control-flow transitions are available as statically evaluable pointcuts. The transition between application-level and kernel-level, for instance, can be considered semantically important, as it is of particular interest not only for the *kernel stack* feature, but for many other system concerns such as *protection*, *validation*, or *scheduling*. If the primary system design offers enough structure to retrieve such neuralgic transitions as static join-points, many fundamental OS concerns can be implemented well-separated and cost-neutral by aspects.

### *The Role of Language*

The amount of available join-points depends moreover on the primary implementation language. The eCos kernel is written in C++. Compared to C, which is still the dominant language in the domain of system software, C++ offers with `class`, `namespace` and `template` additional syntactical constructs to make the module structure and component composition explicit in the code. Thereby, C++ programs usually offer a better separation of concerns and reflect more design decisions in the code. This generally leads to more available static join-points with strong semantics and, thus, a potentially broader applicability of aspects. From the costs perspective, however, there is no fundamental difference between C and C++. As both are statically compiled languages, AOP can in principle be implemented cost-neutral for C as well.

### *Conclusions For the Development of New Software*

The usefulness of AOP rises and falls with the availability of join-points that offer strong semantics. The cost-efficiency depends on the availability of static join-points. This gives the chance to develop "aspect-aware" software by making concern boundaries and design decisions explicit in the code. Besides language support and following the principle of separation of concerns, annotations might be used to explicitly "mark" potentially interesting join-points that offer specific semantics. Newer versions of AspectJ, for instance, already provide support for annotations.

## 6. RELATED WORK

Related work basically exists two domains, namely *AOP in system software* and *costs of aspects*. This furthermore can be subdivided into work based on *static weaving*, and work based on *dynamic weaving*.

An overview of related work regarding *AOP in system software with static weaving* was already presented in the introduction [7, 6, 25, 27, 2, 16, 8, 31, 29]. The main contribution of this paper over existing work is the in-depth cost-analysis, performed with system-specific and cost-critical concerns.

Some related work regarding *costs of AOP with static weaving* has been conducted in the AspectJ domain. Dufour presented a benchmark suite to measure the dynamic behavior of AspectJ programs[12]. His work focuses on a novel measuring approach, however, it also shows that several AspectJ features induce significant overhead. Based on this work, Avgustinov suggested some improvements for the AspectJ code generation[3] that would specifically reduce the overhead caused by *cflow* and *around* in AspectJ programs.

Several papers also suggest to use *dynamic weaving in system software*. Engel presented an approach to dynamically weave C code in the BSD kernel[15]. Arachne [9, 10], a dynamic weaver for C (newer version support C++ as well), has been used to weave aspects into the squid web-proxy. While dynamic weaving clearly has its merits, we consider it as too expensive for small systems and cost-critical concerns such as interrupt synchronization. In the C/C++ domain, the platform-dependence of most dynamic weaving approaches is another issue. A combination of static and dynamic weaving[26], would be promising here, as it combines the best of both worlds.

The *costs of dynamically woven aspects* was analyzed by Haupt for several Java-based dynamic weaving approaches[18]. According to the numbers presented in this study, most Java-based approaches cause tremendous costs. The overhead of the existing C-based approaches for dynamic weaving is, however, much lower[9, 15].

## 7. SUMMARY AND CONCLUSIONS

By comparing a refactored version that uses aspects with the original version of eCos, we could show that separating and applying concerns by aspects instead of tangled and scattered code does not lead to an extra overhead. Software developers can have better separation of concerns for free.

At a first glance, this result might be no surprise, as it should not matter if code is inserted manually or by an aspect weaver. However, aspects obtain context information through a generic interface. Furthermore, some aspect weavers require a runtime system and, in fact, earlier quantitative studies indicate that there *is* an overhead related to AOP [12, 19]. Our work shows that these earlier results do not mean that AOP imposes an overhead *in general*. Especially, aspect weaving in C and C++, which are the dominant languages in the domain of system software, implies neither a runtime system nor expensive late binding. The aspect-oriented implementation of a concern performs as good (or as bad) as a hand-written tangled version.

The second important contribution of this paper is the insight that "ambiguous" join-points are dangerous. Although aspects are able to deal with ambiguous join-points by dynamic language features (e.g. cflow), the programmer might unintentionally spoil the system's performance with only a few lines of aspect code. To reduce this risk, the resource consumption of the different AOP language features has to be much better documented than it is today. In contrast to OOP, it is fortunately relatively easy to identify and avoid the expensive AOP features. As a rule of thumb, dynamic pointcut functions should only be used with extra care.

Ambiguous join-points are the result of a *system design* that was done without aspects in mind. For example, the eCos designers did not distinguish between user↦kernel and kernel↦kernel calls on the interface level. If they had known about aspects, it would have been easy and overhead free to integrated an additional layer for kernel function invocations from the user level. This would have been useful for an overhead-free implementation of the kernel stack feature by an aspect and also various other aspects.

The discussion of an *aspect-aware design* raises the question, how an ideal operating system kernel from the AOP perspective would look like and how far we could go with separation of concerns. To answer this question is the goal of our ongoing and future work on the CiAO operating system family [23, 24]. The results of this paper are an important first step towards a truly aspect-oriented operating system. We are now convinced that the resource consump-

tion of the CiAO system will eventually be competitive with other embedded operating systems for small devices.

# 8. REFERENCES


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

[2] 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 *18th IEEE Int. Conf. on Automated Software Engineering (ASE '03)*, pages 196–204, Montreal, Canada, Mar. 2003. IEEE.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising aspectj. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '05)*, pages 117–128, New York, NY, USA, June 2005. ACM Press.

[4] D. Beuche, A. A. Fröhlich, R. Meyer, H. Papajewski, F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On architecture transparency in operating systems. In *9th SIGOPS European W'shop "Beyond the PC: New Challenges for the Operating System"*, pages 147–152, Kolding, Denmark, Sept. 2000. ACM.

[5] 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 *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

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

[7] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE '01*, 2001.

[8] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using AspectJ for component integration in middleware. In *18th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '03)*, pages 339–344, New York, NY, USA, 2003. ACM.

[9] M. Devillechaise, J. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In M. Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 110–119, Boston, MA, USA, Mar. 2003. ACM.

[10] 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 P. Tarr, editor, *4th Int. Conf. on Aspect-Oriented Software Development (AOSD '05)*, pages 27–38, Chicago, Illinois, Mar. 2005. ACM.

[11] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *11th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '96)*, Oct. 1996.

[12] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of AspectJ programs. In *19th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '04)*, pages 150–169, New York, NY, USA, 2004. ACM.

[13] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of aop. *CACM*, 44(10):33–38, 2001.

[14] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *CACM*, pages 29–32, Oct. 2001.

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

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

[17] B. Harbulot and J. R. Gurd. Using aspectj to separate concerns in parallel scientific java code. In *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD '04)*, pages 122–131. ACM, Mar. 2004.

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

[19] F. Hunleth and R. 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.

[20] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In P. Tarr, editor, *4th Int. Conf. on Aspect-Oriented Software Development (AOSD '05)*, pages 159–168, Chicago, Illinois, Mar. 2005. ACM.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *15th Eur. Conf. on OOP (ECOOP '01)*, volume 2072 of *LNCS*, pages 327–353. Springer, June 2001.

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

[23] D. Lohmann and O. Spinczyk. Architecture-Neutral Operating System Components. *23rd ACM Symp. on OS Principles (SOSP '03)*, Oct. 2003. WiP presentation.

[24] D. Lohmann, O. Spinczyk, and W. 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, Mar. 2005. Northeastern University, Boston (NU-CCIS-05-03).

[25] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the PURE operating system family. In *5th ECOOP W'shop on Object Orientation and Operating Systems*, pages 49–54, Malaga, Spain, June 2002.

[26] W. Schröder-Preikschat, D. Lohmann, W. Gilani, F. Scheler, and O. Spinczyk. Static and dynamic weaving in system software with AspectC++. In Y. Coady, J. Gray, and R. Klefstad, editors, *39th Hawaii International Conference on System Sciences (HICSS '06) - Mini-Track on Adaptive and Evolvable Software Systems*. IEEE, 2006.

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

[28] O. Spinczyk, D. Lohmann, and M. Urban. Advances in AOP with AspectC++. In H. Fujita and M. Mejri, editors, *New Trends in Software Methodologies, Tools and Techniques (SoMeT '05)*, number 129 in Frontiers in Artificial Intelligence and Applications, pages 33–53, Tokyo, Japan, Sept. 2005. IOS Press.

[29] 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.

[30] L. Yu, S. R. Schach, K. Chen, and A. J. Offutt. Categorization of common coupling and its application to the maintainability of the linux kernel. *IEEE TOSE*, 30(10):694–706, 2004.

[31] C. Zhang and H.-A. 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 Press.