# Interrupt Synchronization in the CiAO Operating System

## Experiences from Implementing Low-Level System Policies by AOP

Daniel Lohmann, Jochen Streicher, Olaf Spinczyk
and Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
{lohmann,streicher, spinczyk,wosch}@cs.fau.de

## ABSTRACT

Configurability is a major issue in the domain of embedded system software. Existing systems specifically lack good techniques to implement configurability of architectural OS concerns, such as the choice of *isolation* or *synchronization* policies to use. As such policies have a very cross-cutting character, aspects should provide good means to implement them in a configurable way. While our results show that this is in fact the case, 1) things could have been easier if additional language features were available, and, 2) additional means to influence the back-end code generation turned out to be very important. This paper presents our experiences in using AspectC++ to design and implement interrupt synchronization as a configurable property in the CiAO operating system.

## 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, Experimentation, Design

## Keywords

Aspect-oriented Programming (AOP), AspectC++, CiAO, Configurability, Aspect-aware Operating System

## 1. INTRODUCTION

Configurability is a major issue in the domain of embedded system software. System software for this domain has not only to cope with extremely limited hardware resources, but also with a very broad variety of functional and non-functional requirements [4]. It has to be special-purpose, that is, tailorable to provide exactly the functionality required by the intended application, but nothing more.

The huge diversity of embedded application requirements with respect to the functional and non-functional properties of the underlying OS can also be observed on the system software market. While the number of general-purpose operating systems for PC- and server-like computers has undergone a strong consolidation over the last two decades (eventually resulting in Windows, Linux, MacOS and a few Unices), embedded application developers can select among a zoo of probably more than 100 operating systems, most of them furthermore available as *software families* configurable for dozens of variants. Nevertheless, in more than 50% of all embedded applications the OS-functionality is still proprietary, as none of the existing systems seems to be "configurable enough" to fulfill their particular demands [13]. Existing systems specifically lack good techniques to implement configurability of architectural OS concerns, such as the choice of *isolation* or *synchronization* policies to use. Configurability of such fundamental concerns in embedded OS product lines becomes more and more important. For instance the upcoming automotive standard core AUTOSAR-OS [2] defines different OS feature sets called *conformance classes*. In some conformance classes *isolation* (memory protection) is a mandatory feature, while in the other it is not available. Thus, to cover all conformance classes with a single kernel implementation, the architectural OS concern *isolation* has to become a configurable feature.

### *About this Paper*

In the 2005 ACP4IS workshop we presented an approach towards architecture-neutral OS components that provide configurability of such architectural concerns by AOP [10]. In the paper we sketched an implementation model to configure the interrupt synchronization policy applied to device drivers. The model offers a wide variety of strategies for this policy, ranging from *hard synchronization* (global disabling of IRQs while an IRQ handler is executed) up to *driver threads* (IRQ handler execution by independent preemtable threads).

Meanwhile, we have implemented the suggested model in our CiAO embedded OS. This was an interesting test case to evaluate the suitability of our AspectC++ language and weaver for the particularities of *really* low-level system code[1]—a domain AspectC++ is specifically targeted for [11]. While our results show that this is in fact the case, 1) things could have been easier if additional language features were available, and, 2) additional means to influence the back-end code generation turned out to be very important.

In this paper, we present the design and implementation of interrupt synchronization in CiAO and discuss the lessons we learned about implementing low-level policies by means of AOP.

[1]The handling of hardware IRQs can be considered as the lowest layer of abstraction provided by an OS.

## 2. IRQ SYNCHRONIZATION IN CIAO

### 2.1 The CiAO Embedded OS

In the CiAO project (CiAO is Aspect-Oriented), our group has been developing a family of *aspect-aware* operating systems for embedded and deeply embedded applications. The system is *aspect-aware* in the sense that it has been developed with the idea of configurability by aspects from the very beginning. The goal is to come up with a system design that provides enough join-points to influence all semantically important transitions by aspects without compromising on runtime- or memory efficiency. CiAO is aimed to support a variety of 8-32 bit architectures. Primary development platform is the *Infineon TriCore,* an architecture of 32 bit $\mu$-controllers mostly used in the automotive industry.

CiAO is designed and implemented as a program family. An embedded system designer configures a concrete CiAO variant according to his or her particular requirements. The configuration process is supported by pure::variants [3], an Eclipse-based graphical tool for engineering and configuration of software product-lines. Depending on the chosen configuration, pure::variants selects the set of aspects and classes that implement the concrete CiAO variant.

### 2.2 CiAO IRQ Models

If an IRQ handler needs to access some resource which is currently in use by some thread (or some other IRQ handler), it cannot wait for the resource to be released. Therefore, every OS needs some mechanism to delay the execution of the interrupt code, or at least of those parts accessing the resource, until it is available. That mechanism is called *interrupt synchronization*.

Interrupt service routines in CiAO are explicitly divided into two parts: The first part, called *prologue*, is intended for time-critical tasks and restricted with respect to the resources it may access, typically only hardware registers. Before termination, the prologue may request the (potentially delayed) execution of a second part. The second part, called *epilogue*, is allowed to access other OS components, such as the scheduler. Many operating systems use such an explicit division of the handler code (e.g. Tasklets in Linux or DPCs in Windows). The general idea is to execute the critical part immediately on interrupt level and the second part at a later time when the required resources are available.

CiAO currently provides configurations for three fundamentally different models of interrupt synchronization: *hard synchronization*, *two-phase synchronization* and *continuation synchronization*. They are all based on well-known techniques that are also used in other operating systems [10].

**Hard synchronization.** In this configuration, the two parts are actually combined into one. If an interrupt occurs, prologue and epilogue are just executed consecutively on the interrupt level. Threads accessing shared resources have to disable interrupts. The advantage of this model is its simplicity and low overhead. However, if interrupts are disabled too long, latency rises and IRQ signals might be lost.

**Two-phase synchronization.** The prologue is executed with low latency at interrupt level. Epilogues are queued until the kernel propagates them for execution, which is the case after all nested prologues have terminated and before the scheduler is activated. Epilogues thereby have priority over threads, but are interruptible by prologues if new IRQ signals come in. Threads inside the kernel can temporarily disable the propagation of epilogues to access shared resources. In this case, epilogue propagation is delayed until the thread finishes its

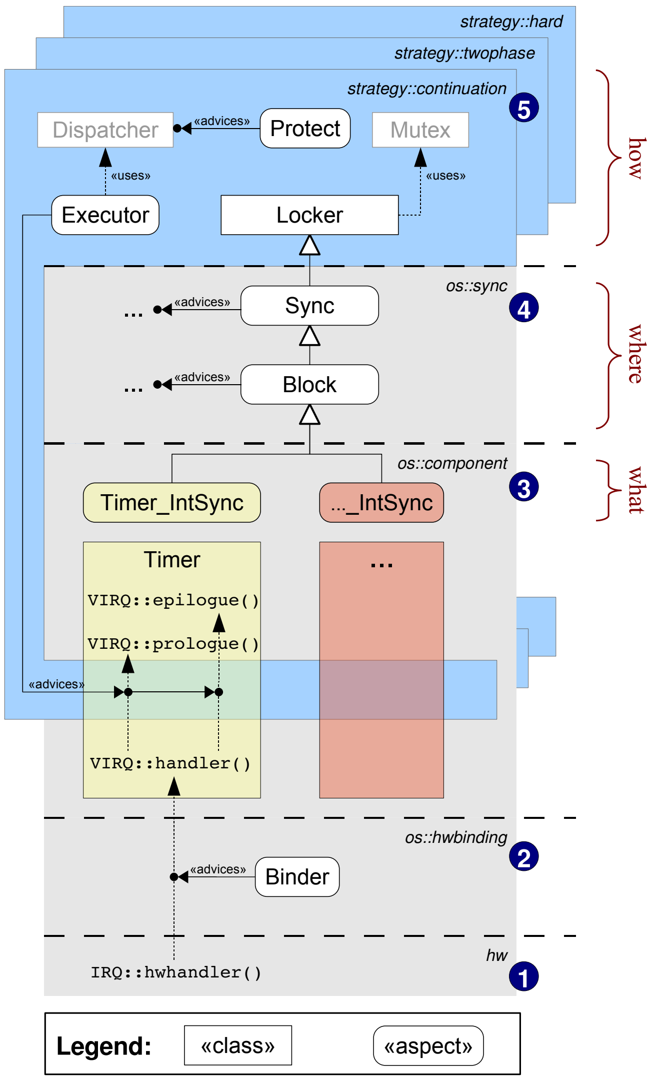

**Figure 1: CiAO IRQ synchronization architecture**

access. Interrupts have only to be disabled, if a thread operates on prologue-accessible state. If low latencies for critical handler code are crucial, this is our model of choice, as prologue deferment is very rare and short.

**Continuation synchronization.** The role of the prologue is the same as above. If an epilogue is requested, a new *continuation* (basic thread abstraction in CiAO) is started to begin execution of the epilogue code. The major advantage of this model is that the execution of a continuation can be delayed if a shared resource is currently in use by some other thread or interrupt. This facilitates fine-grained locking of kernel components. Interrupts and threads are synchronized via mutex objects using a priority inheritance protocol.

### 2.3 Design

An overview of CiAO's interrupt synchronization architecture is given by the graphical illustration in Figure 1. This section briefly describes the fundamental design layers bottom-up. Further implementation details will then be presented in Section 3.

**(1)** Interrupt handling starts in the *hw*-layer, the abstraction of the underlying hardware, which contains an IRQ class for each (platform-specific) hardware IRQ. Each IRQ class contains a static `hwhandler()` function, which is executed when the corresponding interrupt occurs[2].

**(2)** The *os::hwbinding*-layer establishes the link from hardware IRQ abstractions to corresponding software abstractions (VIRQs) in the *os::component*-layer (**3**). The `hwhandler()` has to invoke the corresponding VIRQ's `handler()`. In CiAO, this *upcall* is done statically by the `Binder` aspects, nevertheless it is configurable. An example of such an aspect is also given in the implementation part.

**(3)** The *os::component*-layer contains the functionality of the operating system. It is independent of the interrupt synchronization policy and (partly) also from the underlying hardware. Device drivers, but also other components like the scheduler, are placed in this layer. Device drivers implement the interrupt service code, which has to meet the common handler model's requirements, but has neither information nor any influence on the actual circumstances of its execution. As a driver may service more than one IRQ, prologue and epilogue are contained in *VIRQs* inside the driver. VIRQs are the operating system's software abstraction for a hardware IRQ. Depending on the chosen synchronization model, a VIRQ may also act as a continuation or a queueable object.[3] Every component used by interrupts is subject to interrupt synchronization and provides an `IntSync` aspect which describes its synchronization requirements.

**(4)** The *os::sync*-layer is responsible for enforcing the synchronization constraints. The `Block` aspect enforces disabling of interrupts when methods are called that operate on prologue-accessible state. It may be deactivated if we want to combine prologues and epilogues, which means that they are actually synchronized with the same mechanism. This is the case with the *hard synchronization* strategy. The `Sync` aspect enforces protection of all methods that run on epilogue level. For this distinction, a driver's methods are assigned to different *synchronization classes*. Methods that require synchronization usually belong to the class *synchronized*. However, if they access prologue-accessible state, they belong to the class *blocked*. If they only perform atomic operations or use interruption-transparent algorithms, they do not need to perform any synchronization and belong to the class *transparent*.

**(5)** Finally, a *strategy*-layer implements the chosen model of interrupt synchronization. It provides at least a `Locker` and an `Executor`. Whereas the `Locker` provides the implementation of "locking a resource", the `Executor` is responsible for the proper execution of prologues and epilogues. This responsibility includes the necessary transformations of VIRQs in a way that they are able to act as a continuation or a queueable object.

[2]When an interrupt is requested by peripheral hardware, the CPU starts interrupt service as soon as the required conditions are fulfilled (e.g. interrupts must not be globally disabled). The CPU saves the current thread's context and jumps to a predefined address, the *interrupt table entry*, which invokes the corresponding `hwhandler()`.

[3]Using *two-phase synchronization,* several different prologues may be executed while a thread operates on common data. The possibly resulting requests for different epilogues are managed by putting the VIRQs into the *epilogue queue*.

| aspect type | # | concern |
|---|---|---|
| Binder | per IRQ | upcall (model independent execution) |
| Executor | per model | model dependent execution |
| Sync | 1 | synchronization (where) |
| Block | 1 | synchronization (where) |
| IntSync | per component | synchronization (what) |

**Table 1: concerns of interrupt handling and corresponding aspects in CiAO**

In the shown *strategy::continuation* strategy, the `Executor` activates epilogues as new continuations using the OS thread dispatcher. VIRQs have to be equipped with a continuation context for this purpose; locking is implemented by mutex objects. As the dispatcher is now also activated from interrupt level, it has to be protected by disabling IRQ propagation during dispatching. For this purpose, the strategy implementation contains an additional `Protect` aspect that gives advice to the dispatcher.

## 2.4 Separation of Concerns

The architectural concern of interrupt handling can be divided into few sub-concerns. The *synchronization* of critical system parts deals with the question, which synchronization code (interrupt locks, mutexes, ...) has to be inserted where. The *execution* concern deals with the activation of an interrupt service routine as an entity of its execution model (prologue/epilogue, thread, ...). Although both concerns are obviously not independent of each other, they are separated in the design above and taken care of by different layers.

The *execution* concern is fully located in the strategy layer and taken care of by the `Executor`. The *synchronization* concern is further divided into three sub-concerns which may be expressed as the simple questions *what*, *how* and *where*:

**What.** First, we have the question *what* has to be synchronized, i.e. we need a representation for knowledge of the synchronization requirements of each operating system component. For that purpose, every component provides an `IntSync` aspect. The synchronization requirements are independent of the synchronization strategy.

**How.** The answer to the question *how* we want to synchronize is the strategy-dependent part of synchronization and defines the action to be performed if a method requiring synchronization is called. It is therefore located in the strategy layer.

**Where.** Finally, we have to ensure, that the synchronization code is executed at the right points in the control flow. This is an model-independent task, which is accomplished by the `Sync` and `Block` aspects in the *os::sync* layer.

As the interrupt synchronization is a system-wide policy of very crosscutting character, all device interrupts are managed in the same manner.

The concerns and their corresponding aspects are briefly summarized in Table 1.

## 3. IMPLEMENTATION

The following parts will take a closer look at the implementation of CiAO's interrupt handling architecture using AspectC++ [11].

## 3.1 Driver Implementation

We adapt the simple device driver for the system timer, introduced in [10], to CiAO's driver model.

```
class Timer {
  ... // state
public:
  void init( long time );
  long get() const;
  void add_event(const EventCallback* cb);
private:
  void tick();
  void process_events();
  class VIRQ {
    void handler();
    void prologue() {tick();}
    void epilogue() {process_events();}
  };
};
```

By modifying some of the hardware registers, the `init()` method arms the timer device to request an interrupt after the specified `time`. However, a timer should do this periodically, therefore the `tick()` method simply repeats this procedure. This can be considered time-critical, so it is done in the prologue. Therefore, these hardware registers belong to *prologue-accessible state*. Consequently, `init()` and `tick()` belong to the synchronization class *blocked*. The callback functions which are executed by the epilogue are registered by `add_event()` and have to be held in a queue, which requires synchronization. The `get()` method simply reads the timer value atomically and does not need synchronization.

The information about the synchronization requirements is held by the `Timer_IntSync` aspect, which defines pointcuts for each synchronization class:

```
aspect Timer_IntSync : IntSync {
  pointcut virtual pcSynchronized() =
                  "% Timer::add_event(...)"
               || "% Timer::process_events()";
  pointcut virtual pcBlocked() =
                  "% Timer::init(...)"
               || "% Timer::tick()";
  pointcut virtual pcTransparent() = "% Timer::get()";
};
```

## 3.2 Hardware Binding

`Binder` aspects stablish the link from platform-specific hardware IRQs to the plattform-independed VIRQs used by the OS. A `Binder` aspect is needed for every IRQ/VIRQ pair, the one for the timer driver looks like this:

```
aspect Timer_VIRQ_Binder {
  advice execution("% IRQTimer::handler(...)")
        : after() {
    Timer::VIRQ::handler();
} };
```

The method `IRQTimer::handler()` is actually just an inline defined hook especially for this advice and is called by the real hardware handler. It is a workaround, as the real hardware handler must not be affected by execution advice, because it has interrupt handler specific compiler attributes, which AspectC++ can not handle.

## 3.3 Model Implementation

The `Locker` is simply a type alias to a class which provides methods to be called in order to protect critical method calls. In case of *hard synchronization* that class looks like this:

```
struct Hard {
  static void enter () {ints_disable();}
  static void leave () {ints_enable();}
};
```

For the other models, more sophisticated actions are performed by these two methods.
With respect to configurability, the more interesting part of model implementation is the `Executor` aspect, which looks again very simple in the case of *hard synchronization*:

```
aspect Executor_Hard {
  advice execution("% ...::VIRQ%::handler(...)")
        : after() {
    if (JoinPoint::That::prologue())
      JoinPoint::That::epilogue();
  }
};
```

If we want to have the *two-phase* model, the aspect has also to take care of transforming the VIRQs into queueable objects by introducing a `Queueable` base class:

```
aspect Executor_ProEpi  {
  advice "...::VIRQ%" : slice class QueueSlice
                      : public Queueable {};
  advice execution("...::VIRQ%::handler(...)")
        : after() {
    if ( JoinPoint::That::prologue() ) {
        Guard::relay(JoinPoint::That::getInstance());
  } }
};
```

To be queueable, an actual instance is needed for each VIRQ class, even though the VIRQ classes contain just static elements. To provide such instance, VIRQs are always implemented as singletons. In configurations where the instance is not needed, it is automatically removed by the compiler and linker.

The realization of epilogues as *continuations* requires a thread context the `Executor` may switch to:

```
aspect Executor_Continuation  {
  advice "...::VIRQ%" : slice class {
    static Continuation ctx;
    static char *stack[cfIRQ_EPISTACK];
    static void cfHAL_STARTFUNC_ATTRIBUTES entry() {
      // invoke (usually inlined) epilogue
      epilogue();
    }
  };

  advice execution("...::VIRQ%::handler(...)")
        : after() {
    typedef JoinPoint::That VIRQ;
    ...
    if ( VIRQ::prologue() ) {
      // save current context, start new continuation
      Continuation::getActive()->saveAndStart(
        &VIRQ::stack[cfIRQ_EPISTACK],
        VIRQ::entry, &VIRQ::ctx);
  } ... }
};
```

No object instance for the VIRQ is needed in this case as all members can be static.

## 3.4 The *os::sync* layer

To accomplish the task of combining *what*, *how* and *where*, the aspects of this layer use virtual pointcuts for synchronization advice, which are overridden by the component-specific `IntSync` aspects. The `pcExclude()` pointcut protects *synchronized* but magic code (for example the epilogue itself) from being affected by advice:

```
aspect Sync : Locker {

  pointcut virtual pcSynchronized() = 0;
  pointcut pcToSync() = call(pcSynchronized()
                      && !pcExclude())
                      && !within(pcSynchronized());
  advice pcToSync() : around() {
    enter();
    tjp->proceed();
    leave();
  }
};
```

For *fine-grained* locking as used by the *continuation synchronization* strategy, every component has to be synchronized independently. This is achieved by the fact that an own instantiation of the whole synchronization hierarchy is performed for each (component-specific) `IntSync` aspect, resulting in an own `Mutex` per component. The `Sync` aspect instruments all calls into "foreign" synchronization domains to obtain/release the respective `Mutex` instance around the call.

With *coarse-grained* locking as used by the *hard* and *two-phase* strategies, all components share a single synchronization domain. This is achieved by combining all component-specific `IntSync` aspects with their definitions of the virtual `pcSynchronized()`, `pcBlocked()`, and `pcTransparent()` pointcuts into one single aspect.

## 3.5 Preliminary Evaluation

Table 2 shows first results from our on-going evaluation. It presents the elapsed time [*ns*] from the begin of the hardware interrupt handler to the first *prologue* instruction, *epilogue* instruction, and until interrupt termination (*iret*)[4]. The numbers represent the latency in the optimal case: no other control flow is in the kernel that blocks or delays the execution of prologues and epilogues. It is therefore not surprising that *hard synchronization* performs best in this optimal case as this model involves the lowest ground overhead. With *two phase synchronization* the prologue-activation time is identical, however the potentially delayed execution of the epilogue causes some overhead. As expected, the overhead is highest in the case of *continuation synchronization*. For the later context switch out of interrupt state, additional CPU registers have to be modified before entering the prologue, causing the higher latency for its activation. The context switch to activate the epilogue itself comes at a price, too, even though 1200 *ns* (= 60 clock ticks) can still be considered as a fairly small overhead for the gained flexibility of fine-grained locking.

Even though preliminary, these numbers compare well to those achieved by other embedded OS on this platform. This is shown by the last row, which lists the interrupt handler (prologue) activation latency we measured on a commercial embedded OS widely used in German automotive industry[5]. Hence, an overhead induced by using aspects to apply interrupt synchronization is not observable.

## 4. DISCUSSION

The presented aspect-oriented design and implementation facilitates a good separation of the *what*, *how* and *where* of interrupt synchronization in CiAO. As a result, very different policies can easily and transparently be applied to device drivers and other kernel components. In the following, we discuss some of the interesting issues of the approach.

[4]On a TC1796b running at 50MHz clock speed. Code was compiled with `tricore-gcc` 3.3 using `-O3` optimizations and executed from internal no-wait-state RAM. Measurements were performed with a hardware trace analyzer (Lauterbach). All results were measured (and turned out to be stable) over 10 iterations.

[5]*ProOSEK* from 3SOFT GmbH, http://www.3soft.com. Same compiler settings and setup as above. *ProOSEK* does not provide similar means to split an IRQ handler into prologue and epilogue, so the numbers can be compared only with those of the *hard synchronization* strategy in CiAO.

| [*ns*] | $t_{\text{prologue}}$ | $t_{\text{epilogue}}$ | $t_{\text{iret}}$ |
|---|---|---|---|
| hard | 160 | 160 | 320 |
| two-phase | 160 | 800 | 1200 |
| continuation | 320 | 1200 | 2160 |
| ProOSEK ISR-1 | 240 | (240) | 400 |

**Table 2: Latencies for non-delayed interrupts in CiAO and ProOSEK.**

## 4.1 Aspect-aware OS Design

Configurability of fundamental OS concerns like interrupt synchronization often requires a high amount of configurability in other parts of the system as well. Due to a new strategy that executes epilogues as threads, for instance, it became "suddenly" necessary to protect the dispatcher on the interrupt level. Aspects are in general well suited to apply such cross-cutting extra functionality to foreign components. This requires, however, the availability of suitable join-points aspects can bind to. From the work of Åberg *et. al.* on integrating the *Bossa* scheduling framework with the Linux scheduler by means of AOP [1], we could learn that this can not be taken for granted. Their paper points out that the particularities of the Linux scheduler implementation made it necessary to come up with a (sophisticated) problem-specific AOP approach for Linux kernel code instead of using a general purpose aspect weaver. The scheduler state transitions required by Bossa could not be retrieved by typical pointcut functions such as `call`, `execution`, or `cflow` as they where too hidden in the kernel implementation.

Hence, *aspect-awareness* of kernel components is not reached as a side effect, it has to be handled as its own, global design goal.

### *Explicit Join-Points I*

One way this is taken into account in CiAO is by providing *explicit join-points* for all important transitions. An explicit join-point is given as an (usually) empty hook-function that has the only purpose to be advised by aspects. The CiAO dispatcher, for instance, defines four explicit join-points:

```
class Continuation {
  ...
// explicit join-points, to be execution-advised
  void before_CPURelease( Continuation*& to ) {}
  void before_LastCPURelease( Continuation*& to ) {}
  void after_CPUReceive() {}
  void after_FirstCPUReceive() {}
};
```

The `before_...()` hooks are guaranteed to be invoked by the dispatcher in the context of the leaving thread immediately before releasing the CPU, and the `after_...()` hooks are guaranteed to be executed in the context of the receiving thread immediately after the CPU has been assigned. Thereby, the explicit hooks offer a concise and platform-independent semantics for control-flow transitions that would be difficult to reach by giving advice directly to the (platform-dependent) dispatcher functions. As the hook functions can be inlined, this even does not induce an overhead.

### *Separation of* What, How, *and* Where

While the separation of the *how* and *where* of interrupt synchronization into separate aspects works quite well, the current approach to implement the *what* by component-specific `IntSync` aspects is fragile. Assignment of methods to synchronization classes has to be done manually by listing them in named pointcuts. A better solution would be to "tag" methods directly in the component code by means of an annotation mechanism. Currently, neither C++

nor AspectC++ provide support for this. We think, however, that on the long term such feature is crucial for the scalability of the approach.

## 4.2 Aspects for Near-Hardware Code

A recurring challenge for the implementation was the fact that fundamental low-level OS abstractions, such as the invocation of an interrupt handler or dispatching between different control flows, require an amount of control over the resulting machine code that is generally not guaranteed by the semantics of C/C++. A hardware interrupt handler, for instance, has to make sure that all registers are saved and restored and has to be terminated with a specific end-of-interrupt instruction. The typical solution for such cases is to use either assembler or non-standard compiler- and language extensions (such as `__attribute__(interrupt)` in `gcc`) which ensure that the resulting machine code fulfills the hardware-specific constraints. Join-point shadows in assembler code, however, are just not visible for a high-level-language weaver like the static AspectC++ weaver `ac++`. The alternative is to use either a binary-level weaver or to rely on the mentioned compiler-extensions that make it (mostly) unnecessary to program larger parts in assembler. In either case, however, the code has to be considered as fragile; transformations performed by the aspect weaver may easily break the platform-specific constraints.

### *Explicit Join-Points II*

A pragmatic solution is to use, again, explicit hook functions that are safe to be advised and invoked by the fragile parts when the execution context is no longer constrained. As shown in section 3.1, hardware IRQ classes in CiAO provide an empty `handler()` function for the sole purpose of providing a hook that advice can bind to. Depending on the platform-specific constraints and compiler support, the `handler()` function and all advice given to it can even be inlined into the real interrupt handler, resulting in a safe, but very efficient upcall mechanism for IRQs.

### *Forced Inlining*

The `ac++` weaver transforms advice definitions into `inline` member functions. The C++ `inline` keyword is, however, only a suggestion for the compiler; its interpretation depends on compiler-internal heuristics, optimization flags and so on. In general, this is a good thing—modern compilers are a lot better than the average programmer in deciding when it is really beneficial to embed some function into another. On near-hardware level, however, inlining might be critical for the correctness of the code. Consider a `KernelStack` aspect that implements a strategy to execute interrupt handlers on an own, dedicated kernel stack. In this case, the before-advice which performs the actual stack switch has to be embedded into the interrupt handler, as no function call must take place before the stack has been switched.

In CiAO, we currently solve this issue by translation-unit specific compiler options for the back-end `gcc`-compiler. For all source files that may be targeted by such critical advice, `gcc` is "forced" by specific options to interpret the `inline` keyword literally. However, this is only a workaround: Like other C++ compilers, `gcc` provides language means to declare a function as to be inlined under all circumstances (`__attribute__((always_inline))`). A better solution would therefore be to extend the AspectC++ grammar in a way that such back-end-compiler specific declarators can be given to advice definitions as well and are used in the `ac++` generated code.

### *Join-Point Restrictions*

While explicit join-points provide a pragmatic *alternative* for advising fragile parts of the code, they do not actually *prevent* it. Aspect developers can still intentionally or accidentally formulate pointcut expressions which include fragile elements. It would be beneficial, if the aspect language would provide means to specify certain join-point shadows as non-available, thereby causing them to be implicitly excluded from any pointcut evaluation.

## 5. RELATED WORK

A lot of related work has been conducted in the field of applying AOP to operating system kernels such as Linux [14, 7, 1], FreeBSD [5], NetBSD [6] or eCos [9]. An interesting difference with respect to CiAO is that in most of this work AOP is applied as an *ex post* mechanism to *existing* kernels. Bossa [1] and C4 [7], for instance, advocate for *special-purpose AOP languages* to deal with the particularities of existing kernel code, while CiAO aims to come up with an aspect-aware kernel design that provides ideal support for existing AOP language concepts.

Other related work targets infrastructure software for embedded systems in a broader sense, namely aspects of middleware [15, 8] and quality of service in embedded real time database applications [12].

## 6. SUMMARY AND CONCLUSIONS

The goal of the CiAO project is an *aspect-aware* embedded OS that provides easy configurability of fundamental architectural properties by aspects. By a set of aspects, interrupt synchronization was implemented as a configurable strategy, providing a good separation of the *what*, *how*, and *where* of synchronization issues. The available implementations cover a range of coarse- and fine-grained synchronization approaches, each providing specific advantages and disadvantages. Depending on the actual application scenario, an embedded system developer can choose the strategy implementation that fits best. First evaluation results show that this extra flexibility does not come at the price of efficiency. However, interrupt synchronization should not remain the only configurable architectural OS property. Currently, we are working on the memory protection concern.

While the implementation with AspectC++ was successful, we also learned that applying AOP to such low-level concerns implies some very specific difficulties that are not ideally addressed by the current aspect languages we are aware of. Near-hardware programming requires additional control over the resulting machine code semantics, giving advice to fragile C/C++ code may easily break it. Additional means to use back-end-compiler specific declarators or attributes with advice code might reduce this problem. The possibility to hide "dangerous" join-point shadows from the process of pointcut evaluation, as well as an annotation concept, would at least be very helpful. We are working on appropriate extensions for AspectC++.

## 7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. We especially appreciate the extraordinary effort taken by the first reviewer, whose suggestions led to significant improvements on content, language and style of this paper.

This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4 and SP 968/2-1. Daniel Lohmann was supported by the German Academic Exchange Council (DAAD) under grant no. D/06/40386.

## 8. REFERENCES

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

[2] AUTOSAR. Requirements on operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[3] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.

[4] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

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

[6] M. Engel and B. Freisleben. TOSKANA: a toolkit for operating system kernel aspects. In A. Rashid and M. Aksit, editors, *Transactions on AOSD II*, number 4242 in Lecture Notes in Computer Science, pages 182–226. Springer-Verlag, 2006.

[7] M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS '05)*. USENIX Association, 2005.

[8] F. Hunleth and R. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the 2002 Joint Conference on Languages, Compilers and Tools for Embedded Systems & Soft. and Compilers for Embedded Systems (LCTES/SCOPES '02)*, pages 38–45, Berlin, Germany, June 2002. ACM Press.

[9] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the EuroSys 2006 Conference (EuroSys '06)*, pages 191–204. ACM Press, Apr. 2006.

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

[11] O. Spinczyk and D. Lohmann. The design and implementation of AspectC++. In *Journal on Knowledge-Based Systems, Special Issue on Creative Software Design*. Elsevier North-Holland, Inc., 2007. (to appear).

[12] A. Tešanović, M. Amirijoo, and J. Hansson. Providing configurable QoS management in real-time systems with QoS aspect packages. In A. Rashid and M. Aksit, editors, *Transactions on AOSD II*, number 4242 in Lecture Notes in Computer Science, pages 256–288. Springer-Verlag, 2006.

[13] C. Walls. The Perfect RTOS. Keynote at embedded world '04, Nuremberg, Germany, 2004.

[14] Y. Yanagisawa, K. Kourai, S. Chiba, and R. Ishikawa. A dynamic aspect-oriented system for OS kernels. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*. Springer-Verlag, Oct. 2006. (to appear).

[15] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 130–139, New York, NY, USA, 2003. ACM Press.