Implementing OS Components in Hardware using AOP

Tiago Rogério Mück and Antônio Augusto Fröhlich
Software/Hardware Integration Lab
Federal University of Santa Catarina
Florianópolis, Brazil
{tiago,guto}@lisha.ufsc.br

Michael Gernoth and Wolfgang Schröder-Preikschat
Department of Computer Science 4
Friedrich-Alexander University
Erlangen-Nuremberg
Erlangen, Germany
{gernoth,wosch}@cs.fau.de

ABSTRACT

In this paper we propose a SystemC-based design methodology focusing on the implementation of operating system components in hardware by using Aspect-oriented Programming concepts. As a case study to validate our approach, we have designed and implemented a hardware thread scheduler and a debugging aspect program. For comparison purposes, a hand-made scheduler with debugging capabilities was also implemented. The hardware synthesis results shown that Aspect-oriented Programming concepts and techniques can be efficiently applied to digital hardware design in SystemC through the proposed methodology. The observed overhead in terms of area was less than 1% and the increase in the longest path delay for the circuit was less than 3%. Being SystemC an extension of C++, our strategy puts effective hardware implementation of operating system components into reach for many operating system developers.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Hardware description languages; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms

Design, Languages

Keywords

Aspect-oriented Programming, Digital Hardware Design, Hardware Description Languages, SystemC

1. INTRODUCTION

In contrast with general purpose operating systems, embedded operating systems are usually customized to provide only the functionality necessary to support a well-specified target application. Factoring the operating system into reusable components is a good way to model and to design embedded operating systems. However, dedicated systems are often built as an integrated software/hardware design and the resulting components not rarely need to cope with extreme architectural diversity in order to be effectively reused. Some will be initially deployed in the context of a simple 8-bit microcontroller but will eventually end in an *Application-specific Integrated Circuit* (ASIC) or in a high-end multicore CPU. This architectural variability imposes a major challenge to the design of really reusable operating system components.

The integrated design of software and hardware allows features typically found on operating systems to be implemented in hardware, using programmable logic devices or even designing ASICs. In this context, CPU schedulers have been a favorite for operating system designers considering hardware implementation as means to reduce overhead and interference, particularly on real-time tasks. Furthermore, operations such as context switch and preemption put CPU schedulers high on the list of complicated OS components to be implemented in hardware and thus are usually taken as demonstrators for novel design strategies [28, 26, 21, 2, 24, 31, 8, 3].

However, bringing operating system components to hardware is not a trivial task. Currently there is still a considerable gap between the methodologies and languages used in software design and those used in hardware design. Electronic circuits are created from descriptions written in a *Hardware Description Language* (HDL). These descriptions can be used either for design verification or for hardware synthesis (i.e. generate the physical circuit from a description). Differently from most software programming languages, HDLs are intrinsically parallel and provide explicit means to describe timing. VHDL [15] and Verilog [16] are the most widely used HDLs and provide means for designing hardware at *Register Transfer Level* (RTL). In RTL, circuits are described in terms of the operations between storage elements which are synchronized using clock signals. In Verilog/VHDL, a hardware design may be composed by several `modules`, whose instances are interconnected to build the system. However, in contrast to programming languages, the modules communication is data-driven, and occurs through signals defined by the modules input/output interface.

Nevertheless, the ever increasing complexity of digital hardware designs is leading to the unification of hardware and software design methodologies. For example, *Object-oriented Programming* (OOP) is already supported in the hardware domain by some HDLs, such as SystemC [29], a C++ based modeling platform and language supporting design abstractions at both low and high levels of abstraction. However, the advances in software engineering have already shown that OOP still have some limitations in the way it allows a complex problem to be broken up into reusable abstractions. Even though most classes in an object-oriented model will perform a single function, they often share common, secondary elements with other classes. The implementation of these *crosscutting concerns* is scattered among multiple abstractions, thus breaking the encapsulation principle. *Aspect-oriented Programming* (AOP) [20] is an elaboration over OOP to deal with crosscutting concerns. AOP proposes the encapsulation of such concerns in special units called *aspects*. An aspect can alter the behavior of the base code by applying *advices* (small pieces of code defining additional behavior) in specific points of a program called *pointcuts*. Some extensions to OOP languages have been proposed to support these concepts. For example, AspectJ [19] and AspectC++ [32] extend Java and C++ with full support for AOP features. They provide both new language constructs and an *aspect weaver*, a tool responsible for applying the advices to the base code before it is processed by the traditional compiling chain.

Analogous to software, in hardware some system-wide crosscutting concerns cannot be elegantly encapsulated. For example, in complex circuits, interconnection of several entities is realized by introducing buses. A bus physically interacts with other components (e.g. CPU, memory, devices), but it is difficult to use a module or a class to encapsulate the bus because its interface and arbitration method has to be implemented in every attached component [12]. Other examples of crosscutting concerns in hardware designs can be also found in parts of a system related to its overall functionality or to the implementation of non-functional properties such as fault-tolerance, power management, debugging, clock handling, and many others [11]. Even with the introduction of OOP in hardware, this scattered code is hard to maintain and bugs may be easily introduced. The introduction of AOP to hardware design is expected to provide the easy encapsulation of cross-cutting concerns and an increase in the overall design quality.

In this paper we aim to close the gap between the design of hardware and software operating system components. We propose a design methodology which leverages on SystemC features in order to enable the implementation of operating system components in hardware using OOP and AOP concepts. Along with the use of OOP techniques (e.g. inheritance), we propose the use of a domain engineering strategy which yields components whose execution scenario dependencies are isolated and encapsulated as *aspects* and *configurable features*. These artifacts are implemented using standard C++ metaprogramming features within the SystemC synthesizable subset [27], thus yielding *synthesizable components* that can be more easily modified and reused in a wider range of execution scenarios. This method is illustrated by the design and implementation of a task scheduler.

The remaining of this paper is organized as follows: Section 2 presents a discussion about works related to both the implementation of operating system features in hardware and the implementation of hardware components using AOP; Sections 3 and 4 present our methodology and the design and implementation of our task scheduler in hardware; Section 5 discuss our experimental results; and Section 6 closes the paper with our conclusions.

2. RELATED WORK

In this section we provide an overview of previous works related to the implementation of operating system features in hardware. In the subsequent session we branch to a comprehensive discussion about works related to the deployment of AOP techniques in hardware design.

2.1 HW-based Operating Systems

Several research groups have explored hardware/software codesign for real-time systems in the last years. Hardware support for task schedulers was proposed, among others, by Mooney, who has implemented a cyclical scheduler [26], and by Kuacharoen, who has implemented the RM and EDF priority algorithms [28]. Beyond the support for tasks scheduling, Kohout has developed hardware support for time and event management, taking advantage of the fact that these activities are very often present in real-time systems and have a high intrinsic parallelism [21]. However, this support is limited to fixed priority scheduling and the hardware implementation of such features does not follow a specific design methodology that could be reused to bring another components to hardware as well.

Other works focus on the unification of the interface between software and hardware. This approach is followed by the *HThread* project [2], the ReconOS [24] and the BORPH [31] operating system. In these works a task performed in hardware is also abstracted as an OS thread, and a system call interface is provided between them. In order to allow the interaction of hardware and software threads, these works propose the implementation of schedulers and synchronization devices on both domains (hardware and software). However, despite providing this unified interface, an enormous gap still exists between the way the hardware and software threads themselves are implemented.

The HW-RTOS [8] follows a different approach to bring operating systems features to hardware. It leverages on behavioral synthesis in order to implement a hardware unit responsible for task scheduling and inter-process communication. HW-RTOS was described in C and synthesized with a behavioral synthesis tool, which allowed features to be implemented in hardware by extracting pieces of code directly from software RTOS kernel.

Agkul has implemented the priority inheritance protocol with a hardware resource manager in order to prevent deadlocks and unlimited task blocking [1]. Rafla and Gauba have proposed to implement the context switch in multithread operating systems inside the processor. They proposed the creation of extra register files dedicated for saving the context of specific threads, thus allowing very fast context switches in real-time environments [30].

2.2 AOP Applied to Hardware Design

Several works have already addressed the use of AOP concepts in hardware design. Engel and Spinczyk discussed the nature of crosscutting concerns in VHDL-based hardware design and proposed a hypothetical AOP extension to VHDL [12]. However, the work lacks a concrete implementation so that the impact of AOP in the design can be consistently evaluated. Bainbridge-Smith and Park discussed how the separation of concerns may relate to different levels of algorithmic abstraction. They have mentioned the development of ADH, a new HDL based on AOP, but further details about ADH are not mentioned [5]. Burapathana and others proposed the use of AOP concepts to sequential logic design. Nevertheless, they focused on very simple and low level examples like flip-flops and logic gates [6].

There are also several works that proposed the use of AOP concepts mostly for hardware verification. Kallel and others proposed the use of SystemC and AspectC++ to implement assertion checkers [18]. They focused on the verification of *Transaction-level Models* (TLM) [7] in which transaction state updates are taken as pointcuts. They provide a framework in which the user's verification classes extend base aspect classes that implement the pointcuts and the verification primitives. Vachharajani and others have developed the *Liberty Structural Specification Language* (LSS) [33]. In LSS, each module can declare instances which emit certain events at runtime. These events behave like pointcuts of AOP. Each time a certain state is reached or a value is computed, the instance will emit the corresponding event and user-defined aspects will perform statistics calculation and reporting. Liu and others also proposed AOP-based instrumentation, but focusing high-level power estimation [23]. They have developed a methodology based on SystemC in which AspectC++ is used to define special power-aware aspects. These aspects are used as configuration files to link power aware libraries with SystemC models.

Other works provide AOP features not only for verification, but also for actual hardware design. Déharbe and Medeiros presented and assessed possible applications of AOP in the context of integrated system design by using SystemC with AspectC++ [10]. Differently from the works discussed previously, they showed how AOP can be used to encapsulate some functional characteristics of hardware components. They modeled as aspects the replacement policy of a cache, the data type of an FFT, and the communication protocol between modules. However, only simulation results are shown and they do not compare the implementation of aspect-based components against components with all the functionalities hand-coded. In a similar work, Liu and others implemented a SystemC model for a 128-bit floating-point adder and described the implementation of the same model using AOP techniques [22]. But, synthesis results are not provided and the two models are compared only in terms of functionality to show that the AOP design works like the original SystemC-only design. ASystemC [11] also extends SystemC in a similar fashion, but, instead of using AspectC++, the authors developed their own aspect weaver. The new aspect language was introduced through different case studies involving high-level estimation of circuit size, feature-configurable products, and assertion-based verification. However, the evaluation of ASystemC has the same flaws of the works discussed above.

Other works in this area follow different approaches. The *E* programming language [34] was designed for modeling and verification of electronic systems and some of its mechanisms can be used to support AOP features. Apart from its OOP features, *E* has some constructs to define the execution order of overloaded methods in inherited classes, which can be used to define pointcuts and implement aspects. Indeed, this can be used to implement the behavior of hardware components, but *E* is more focused in high-level specification and there is not any tool support for synthesis. Jun and others have analyzed the application of *Aspectual Feature Module* (AFM) [4] to HDLs. They have implemented a RISC processor using SystemC and FeatureC++ [4], and showed how AFM enables the incremental development of hardware through the modularization of code fragments for the implementation of a function [17]. However, AOP is used only for encapsulation of verification code and the authors do not provide synthesis results of the resulting code.

In summary, several of the previous works have focused on high-level specification and AOP features are used mostly for code instrumentation and verification. Also, there are not any related work aiming at using AOP for the actual design of synthesizable hardware, since, as discussed above, all works present experiments only at the simulation level and lack a more comprehensive discussion about the overheads related to the use of AOP.

3. DESIGNING HARDWARE OS COMPONENTS USING AOP

Application-driven Embedded System Design (ADESD) [13], the design method used in this work, is an elaboration over techniques which have been used in the software domain to develop component-based systems. The methodology elaborates on commonality and variability analysis—the well-known domain decomposition strategy behind OOP—to add the concept of aspect identification and separation at early stages of design. It defines a domain engineering strategy focused on the production of families of scenario-independent components. Dependencies observed during domain engineering are captured as *scenario aspects*, thus enabling components to be reused on a variety of execution scenarios by the application of the respective *scenario aspects*. This aspect weaving is performed by constructs called *Scenario Adapters* [14].

The design artifacts proposed in ADESD were implemented and validated on the *Embedded Parallel Operating System* (EPOS) [13]. EPOS aims to automate the development of dedicated computing systems, and features a set of tools to select, adapt, and plug components into an application-specific framework, thus enabling the automatic generation of an application-oriented system instance. EPOS is implemented in C++ and leverages on *Generative Programming* [9] techniques such as *Static Metaprogramming* in order to achieve high reusability with low overhead.

Whether such guidelines can also be defined for designing operating system hardware components has not yet been investigated, but nonetheless, SystemC enables the introduction of convenient C++ constructs to increase the quality of hardware designs. This will be demonstrated in the next sections.

3.1 Scenario Adapters

Scenario adapters were developed around the idea of components getting in and out of an execution scenario, allowing actions to be executed at these points, therefore, a scenario must define at least two different operations: `enter` and `leave`. These actions must take place respectively before and after each of the component's operation in order to setup the conditions required by the scenario. For example, in a compressed scenario, enter would be responsible to decompress the component's input data, while leave would compress its outputs.

In the software domain, components are objects which communicate using method invocation (considering an OOP-based approach) and the execution of all operations are naturally sequential, so the scenario adapters were originally developed to provide means to just efficiently wrap the method calls to an object with enter and leave operations. However, in the hardware domain, components have input and output signals instead of a method or function interface, and all operations are intrinsically parallel. These different characteristics required some modifications of the original scenario adapter. The new scenario adapter is shown in Figure 1.

SystemC defines hardware components by the specialization of the `sc_module` class. Components communicate using special objects called `channels`. SystemC channels can be used to encapsulate complex communication protocols at register transfer or higher levels of abstraction. However, these complex channels lie outside the SystemC synthesizable subset, so we use only `sc_in` and `sc_out`, which define simple input and output ports for components. Methods which implement the component's behavior must be defined as SystemC processes. In our examples we use SystemC clocked threads (`SC_CTHREAD`), in which all operations are synchronous to a clock signal. The implementation of the `Component::controller` method in Figure 1 shows the common behavior of a `SC_CTHREAD`. SystemC `wait()` statements must be used to synchronize the operations with the clock, in other words, all operations defined between two `wait()` statements occur in the same clock cycle.

Using these constructs, we define each aspect as a single and independent hardware component (`Aspect` class). `enter()` and `leave()` operations are defined using a simple handshaking protocol (`op_rdy_out` and `op_req_in` signals) to trigger its execution. The remaining input/output ports define which operation are being triggered (this is specific of each aspect). With this kind of handshaking communication protocol we can produce more reusable components, since the number of clock cycles it requires for each operation is hidden by the protocol, thus making it easier to synchronize component execution with the rest of the design.

The `Scenario` class incorporates, via aggregation, all of the aspects which define its characteristics. It defines `enter()` and `leave()` methods to encapsulate the implementation of the handshaking protocol which trigger the aspects. Figure 1 shows how the scenario's `enter()` operation is implemented. All aspects are triggered at the same time and executes in parallel, however, if required by the scenario, this can be modified in order to execute each aspect sequentially at the cost of additional clock cycles.

The adaptation of the component to the scenario is performed by the `Scenario Adapter` class via inheritance. This adaptation is possible through the separation of the component's input/output protocol from the implementation of its behavior. A SystemC process (`controller` method) handles the input/output protocol (`behavior` method) and calls the requested operations, which are each implemented in its own methods. These methods are overridden in the `Scenario Adapter` class. Notice that, although scattered through a class hierarchy and different methods, all operations (from the handling of the component's input/output protocol, to the triggering of the aspects) executes inside the `controller SC_CTHREAD` process. For the proposed scheme to work, `wait()` statements are also used to schedule the operations among the clock cycles, instead of defining explicit state machines. If the latter is used, it would not be possible to elegantly implement the structure described in Figure 1, since a state machine would require manual intervention to add the operation defined by the scenario.

3.2 Configurable Features

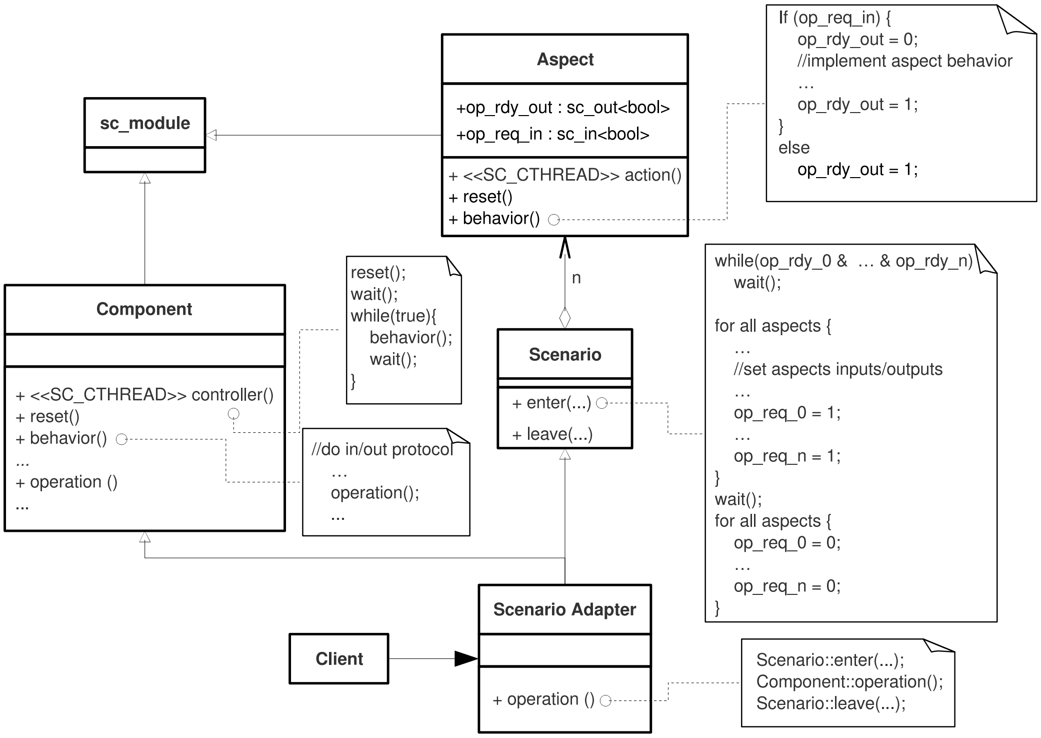
Additionally to the analysis and domain engineering process, several characteristics can be identified as configurable features of the components. In fact, such characteristics represent fine variations within a component, which can be set in order to change slightly its behavior or structure. Figure 2 shows how this features can be implemented using *Static Metaprogramming* [9] techniques. Special template classes called *Traits* are used to define which characteristics of each component is activated. Metaprograms are then used to conditionally modify the component behavior or modify its structure through inheritance.

3.3 ADESD and Classic AOP

Several previous works have already discussed aspect-oriented hardware design using SystemC and proposed solutions based on classic AOP concepts using the well-known AspectC++ language. Indeed, AspectC++ provides more powerful mechanisms for aspect implementation then ADESD, especially when it comes to the definition of the pointcut, however, this additional mechanisms are usually either unnecessary or can be efficiently replaced. For example, the aspects implemented by Déharbe and Medeiros [10] (Section 2) could be more elegantly implemented using other standard C++ features like inheritance and templates parameters. In the scope of ADESD, we can say that scenario adapters can be used to implement *homogeneous crosscutting* [17] (the process of adding the same behavior for all classes). *Heterogeneous crosscutting* [17](when concern is specific to a certain component or family of components) can be easily implemented with standard OOP (e.g. inheritance). Additionally the implementation of ADESD's mechanisms can be realized using only standard SystemC features. Previous works focus on tools and languages which were deployed only for software development (e.g. AspectC++), which limits its use for the generation of synthesizable hardware.

4. CASE STUDY: A HARDWARE SCHEDULER

Our case study is based on a previous implementation of the EPOS scheduler described by [25], which described a task scheduling suitable for hardware and software implementa-



```
If (op_req_in) {
    op_rdy_out = 0;
    //implement aspect behavior
    …
    op_rdy_out = 1;
}
else
    op_rdy_out = 1;
```

```
reset();
wait();
while(true){
    behavior();
    wait();
}
```

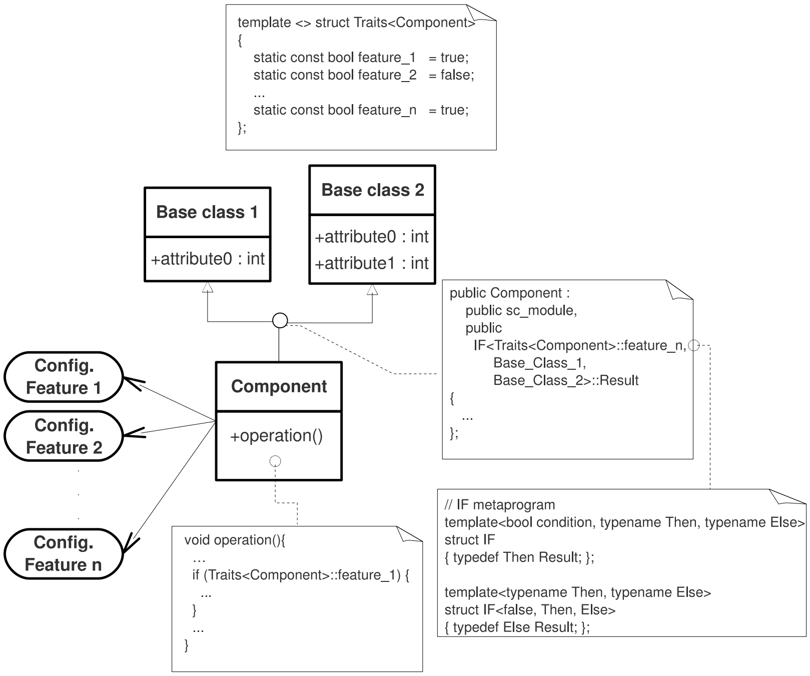
```
//do in/out protocol
    …
    operation();
    …
```

```
while(op_rdy_0 &  … & op_rdy_n)
    wait();

for all aspects {
    …
    //set aspects inputs/outputs
    …
    op_req_0 = 1;
    …
    op_req_n = 1;
}
wait();
for all aspects {
    op_req_0 = 0;
    …
    op_req_n = 0;
}
```

```
Scenario::enter(...);
Component::operation();
Scenario::leave(...);
```

Figure 1: UML class diagram showing the general structure and behavior of a scenario adapter.



```
template <> struct Traits<Component>
{
    static const bool feature_1   = true;
    static const bool feature_2   = false;
    ...
    static const bool feature_n   = true;
};
```

```
public Component :
    public sc_module,
    public
      IF<Traits<Component>::feature_n,
           Base_Class_1,
           Base_Class_2>::Result
{
   ...
};
```

```
void operation(){
  …
  if (Traits<Component>::feature_1) {
    ...
  }
  ...
}
```

```
// IF metaprogram
template<bool condition, typename Then, typename Else>
struct IF
{ typedef Then Result; };

template<typename Then, typename Else>
struct IF<false, Then, Else>
{ typedef Else Result; };
```

Figure 2: Components behavior and structure modified by configurable features.

tion. However, the original VHDL implementation was not susceptible to the same mechanisms that render its software counterpart flexible and reusable. The new System-based hardware scheduler is described below.

Figure 3 shows a simplified view of the task scheduling model. In this design, the task is represented by the class `Thread` and defines the execution flow of the task, implementing the traditional functionality (e.g. suspend and resume operations). This class models only aperiodic tasks. Periodic tasks, a common abstraction of real-time systems, are in fact a specialization of the `Thread` class which aggregates the mechanisms related to the re-execution of the task periodically, responsible for reactivating the task when a new period expires.

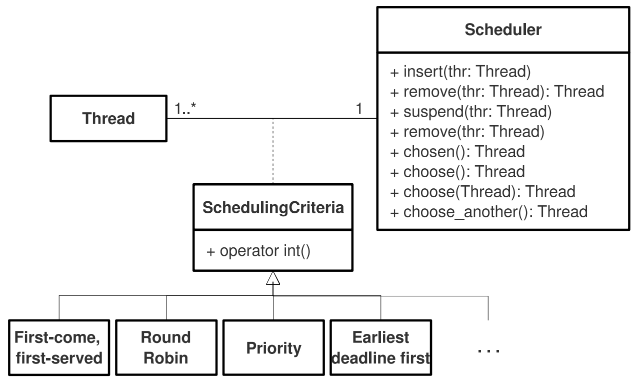


Figure 3: Simplified UML view of the task scheduling model.

The classes `Scheduler` and `SchedulingCriteria` define the structure that realizes the task scheduling. Traditional design and implementations of scheduling algorithms are usually done by a hierarchy of specialized classes of an abstract scheduler class, which can be further specialized to bring new scheduling policies to the system. In order to reduce the complexity of maintenance of the code (generally present in such hierarchy of specialized classes), as well as to promote its reuse, the design detaches the scheduling policy (criteria) from its mechanisms (lists implementations) and also detaches the scheduling criteria from the thread it represents. This is achieved by the isolation of the element's comparison algorithm of the scheduler in the criteria.

4.1 Hardware Implementation

The separation of the mechanism from the scheduling policy was fundamental for the construction of the scheduler in hardware. The hardware scheduler component implements only the mechanisms that realize the ordering of the tasks, based on the selected policy. In this sense, the same hardware component can realize distinct policies.

The implementation of the scheduler in hardware follows a well-defined structure. It has an internal memory that implements an ordered list. One process (`Controller`) is responsible for interpreting all the data received by the interface of the component in hardware and then to activate the process responsible for implementing the functionality requested by the user (through the command interface register). This implementation, as the software counterpart, realizes the insertion of its elements already in order, that is, the queue is always maintained ordered, following the information that the `SchedulingCriteria` provides. In the memory of the component, a double-linked list is implemented.

It worth's highlight two aspects of the implementation of this component regarding its implementation on hardware, especially for programmable logic devices. Both of these aspects are related to the constraints in terms of resources of such devices. Ideally, a hardware scheduler should exploit as most the inherent parallelism of the hardware resources. However, such resources are very expensive, especially when the internal resources are used to implement several parallel bit comparators in order to search elements on the queue, as well as to find the insertion position of an element in queue.

Moreover, the use of 32 bits pointers to reference the elements stored on the list (in this case `Threads`) becomes extremely costly for implementing the comparators to search such elements. On the other side, the maximum number of tasks in an embedded system is usually known at design time, and for that reason, the resources usage of this component could be optimized by implementing a mapping between the system pointer (32 bits) and an internal representation that uses only the necessary number of bits, taking into account the maximum number of tasks running on the system.

Another aspect is related to the search of the position of insertion of the element on the queue. Ideally, such searching could be implemented through a parallel comparison between all elements on the queue, in order to find the insertion point in only one clock cycle. However, such approach, besides increasing the consumption of the resources, as the number of tasks increase it could lead to a very high critical path delay on the synthesized circuit, and thus, to reduce the operating frequency of the component.

By this reason, the insertion of elements was implemented doing a sequential search of the insertion position of the element, which will take N cycles in the worst-case. Besides in this approach, the insertion time could the variable, such variation is hidden by the effect that the insertion could be realized in parallel to the software running on the CPU.

4.2 Aspects Implementation

We have implemented aspects for debugging. Unlike previous works, which focused on simulation-time tracing and logging [33, 23], we have focused on *Design for Testability* [35] and implemented aspects for on-chip debugging using a JTAG scan chain. Figure 4 shows the debugged family of hardware aspects. The class `DebuggedCommon` defines common ports for all aspects. Besides the ports used for clock and reset, it defines outputs for a JTAG debug protocol (`trigger_out` and `data_out`) and for the enter/leave protocol (`op_rdy_out` and `op_req_in`). The input values for the ports defined by the subclasses determine which operation will be triggered.

The aspects implemented define the following debugging functionality: `Watched` causes the state of a component to be dumped every time it is modified; `Traced` causes every operation execution to be signalized; and `Profiled` counts

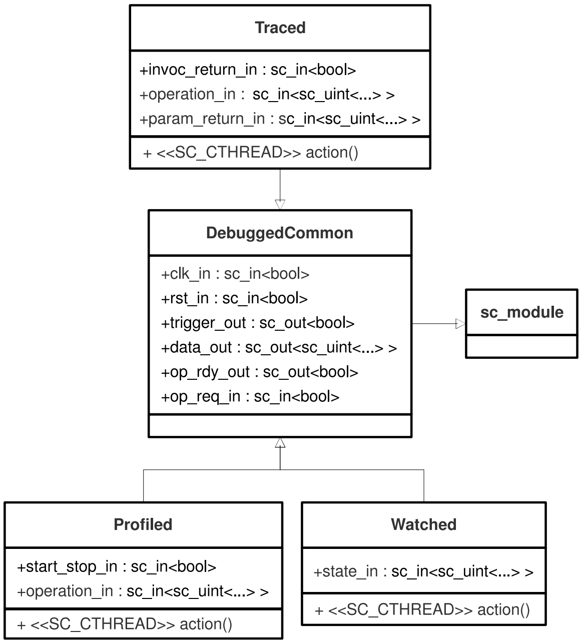
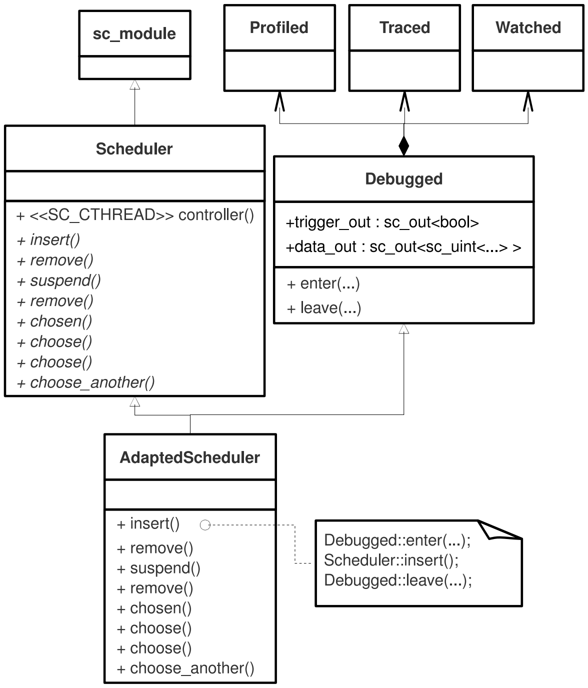

Traced
+invoc_return_in : sc_in<bool>
+operation_in : sc_in<sc_uint<...> >
+param_return_in : sc_in<sc_uint<...> >
+ <<SC_CTHREAD>> action()
DebuggedCommon
+clk_in : sc_in<bool>
+rst_in : sc_in<bool>
+trigger_out : sc_out<bool>
+data_out : sc_out<sc_uint<...> >
+op_rdy_out : sc_out<bool>
+op_req_in : sc_in<bool>
sc_module
Profiled
+start_stop_in : sc_in<bool>
+operation_in : sc_in<sc_uint<...> >
+ <<SC_CTHREAD>> action()
Watched
+state_in : sc_in<sc_uint<...> >
+ <<SC_CTHREAD>> action()

Figure 4: The debugged family of hardware aspects.



```
Debugged::enter(...);
Scheduler::insert();
Debugged::leave(...);
```

Figure 5: Scheduler modified by the scenario adapter.

the number of clock cycles used by the component for each operation.

4.3 Scenario Adapter Implementation

Figure 5 shows how we applied the aspects to the scheduler using a scenario adapter (for simplicity, some details, such as methods, ports, and hierarchies, are omitted). The implementation follows the guidelines depicted in Figure 1. The class `Scheduler` defines the scheduler component. The `controller` SystemC process is responsible for reading the component inputs and calling the method which implements the corresponding operation. The class `AdaptedScheduler` implements the scenario adapter. It inherits from the `Scheduler` and `Debugged` classes, and redefines the operation methods by adding calls to the `enter()` and `leave()` methods of `Debugged`. The `Debugged` class defines the scenario and its methods implement the handshaking protocol that triggers the aspects components.

5. EXPERIMENTAL RESULTS

We have evaluated the efficiency aspect-oriented implementation described previously with an object-oriented-only implementation in which the aspects behavior is *hand-coded* in the core components. We have synthesized our design to physical circuits targeting a *Field-programmable Gate Array* (FPGA) and analyzed their performance and size (area). Figure 6 shows the synthesis flow. The SystemC designs are first converted to VHDL descriptions using Celoxica's Agility 1.3. Then, Xilinx ISE 13.1 is used for both logic synthesis and place-and-route (the process of fitting a circuit for a specific FPGA device). As our target device we have chosen a Xilinx XC3S2000 FPGA. All the synthesis processes described in the flow were executed with all the optimizations enabled.

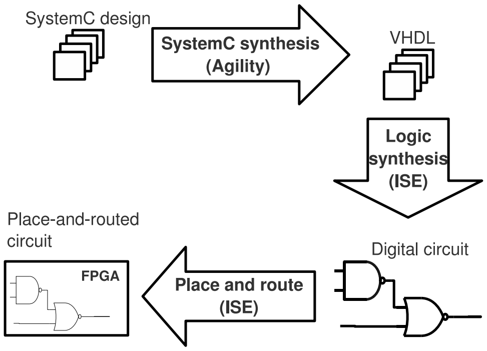


Figure 6: Design synthesis flow targeting FPGAs.

Tables 1 and 2 show the number of *slices* used (a configurable logic element in Xilinx's FPGA) and the *longest path delay* (LPD). The LPD represents the performance of the circuit, while the number of slices is used to evaluate the area. *Standard scheduler* is the scheduler component without any modification, *Debugged scheduler—scenario adapter* is the scheduler modified with the scenario adapter while *Debugged scheduler—hand coded* is the scheduler with the aspects functionalities hand coded. Table 2 also shows the debugged family synthesized in isolation.

The results show that the use of scenario adapters yields

Table 1: Hardware resources estimated by Agility

Parameter	Normal scheduler	Debugged scheduler hand coded	Debugged scheduler scenario adapter
4-input LUTs	2887	2978	3037
Flip Flops	663	754	842
Longest path delay (ns)	32.76	32.76	32.76

Table 2: Hardware resources used after placing and routing Agility's netlists

Parameter	Normal scheduler	Debugged scheduler hand coded	Debugged scheduler scenario adapter	Profiled	Traced	Watched
4-input LUTs	2942	3042	3097	30	40	11
Flip Flops	663	754	842	28	41	12
Occupied Slices	1563	1668	1685	21	22	8
Longest path delay (ns)	23.28	22.58	23.28	4.79	6.00	4.70

a very low overhead in terms of both resource consumption and performance. For the scenario-adapted scheduler, the number of occupied slices is about 1% higher than the hand-coded scheduler. This overhead comes basically from the additional signal and registers required by the handshaking protocol that is used to trigger the aspects, which is not required when everything is coded within a single SystemC process. The difference in performance (given by the longest path delay) is about 3%. Curiously, in the final place-and-routed designs, the hand-coded scheduler has the smaller longest path delay. This may be the result of some optimization algorithm applied in the place-and-route back-end.

6. CONCLUSION

In this paper we have shown how the ADESD domain engineering strategy and AOP techniques can be applied to design and implement flexible operating system components in hardware. As a case study we have implemented a task scheduler in SystemC. The scheduler's dependencies from a debugging execution scenario were encapsulated in aspects and further applied to the core component through the use of a scenario adapter, thus providing a better separation of concerns.

In comparison with other approaches, we have focused in the design of synthesizable hardware components, rather than verification and simulation-only models. The results showed that our design artifacts can be synthesized without introducing significant overhead in the generated components.

Acknowledgments

This work was partially supported by the *Coordination for Improvement of Higher Level Personnel* (CAPES) grant, projects RH-TVD 006/2008 and 240/2008, and by the *German Research Council* (DFG) under grant no. SCHR 603/7-1.

7. REFERENCES

[1] B. Akgul. Hardware support for priority inheritance. In K. A. Publishers, editor, *24th IEEE International Real-Time Systems Symposium*, 2003.

[2] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. Enabling a uniform programming model across the software/hardware boundary. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 89–98, 2006.

[3] J. ao Paulo Pizani Flor, T. R. Mück, and A. A. Fröhlich. High-level design and synthesis of a resource scheduler. In *18th IEEE International Conference on Electronics, Circuits, and Systems*, Beirut, Lebanon, dec 2011.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34:162–180, March 2008.

[5] A. Bainbridge-Smith and S.-H. Park. ADH: an aspect described hardware programming language. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 283–284, dec 2005.

[6] P. Burapathana, P. Pitsatorn, and B. Sowanwanichkul. An Applying Aspect-Oriented Concept to Sequential Logic Design. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 819–820, Washington, DC, USA, 2005. IEEE Computer Society.

[7] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[8] S. Chandra, F. Regazzoni, and M. Lajolo. Hardware/software partitioning of operating systems: a behavioral synthesis approach. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, GLSVLSI '06, pages 324–329, New York, NY, USA, 2006. ACM.

[9] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[10] D. Déharbe and S. Medeiros. Aspect-oriented design in systemC: implementation and applications. In *Proceedings of the 19th annual symposium on*

Integrated circuits and systems design, SBCCI '06, pages 119–124, New York, NY, USA, 2006. ACM.

[11] Y. Endoh. ASystemC: an AOP extension for hardware description language. In *Proceedings of the tenth international conference on Aspect-oriented software development companion*, AOSD '11, pages 19–28, New York, NY, USA, 2011. ACM.

[12] M. Engel and O. Spinczyk. Aspects in hardware: what do they look like? In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '08, pages 5:1–5:6, New York, NY, USA, 2008. ACM.

[13] A. A. Fröhlich. *Application-Oriented Operating Systems.* Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, aug 2001.

[14] A. A. Fröhlich and W. Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA, 2000.

[15] IEEE. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*, 2000.

[16] IEEE. *Std 1364-2001: IEEE Standard Verilog Hardware Description Language*, 2001.

[17] Y. Jun, L. Tun, and T. Qingping. The application of Aspectual Feature Module in the development and verification of SystemC models. In *Specification Design Languages, 2009. FDL 2009. Forum on*, pages 1 –6, sep 2009.

[18] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne. Verification of systemc transaction level models using an aspect-oriented and generic approach. In *Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2010 5th International Conference on*, pages 1 –6, mar 2010.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvï¿½skylï¿½, Finland, June 1997. Springer.

[21] P. Kohout and B. Jacob. Hardware support for real-time operating systems. In *Proceedings of CODES - ISSS'03*, Newport Beach, CA - USA, 2003.

[22] F. Liu, O. A. Mohamed, X. Song, and Q. Tan. A case study on system-level modeling by aspect-oriented programming. In *Proceedings of the 2009 10th International Symposium on Quality of Electronic Design*, pages 345–349, Washington, DC, USA, 2009. IEEE Computer Society.

[23] F. Liu, Q. Tan, X. Song, and N. Abbasi. AOP-based high-level power estimation in SystemC. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, GLSVLSI '10, pages 353–356, New York, NY, USA, 2010. ACM.

[24] E. Lubbers and M. Platzner. A portable abstraction layer for hardware threads. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 17 –22, sept. 2008.

[25] H. Marcondes, R. Cancian, M. Stemmer, and A. A. Fröhlich. On the Design of Flexible Real-Time Schedulers for Embedded Systems. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 02*, CSE '09, pages 382–387, Washington, DC, USA, 2009. IEEE Computer Society.

[26] V. Mooney and G. D. Micheli. Hardware/software codesign of run-time schedulers for real-time systems. In *Proceedings of Design Automation of Embedded Systems*, pages 89–144, 2000.

[27] OSCI. Systemc synthesizable subset draft 1.3, 2010.

[28] M. S. P. Kuacharoen and V. Mooney. A configurable hardware scheduler for real-time systems. In *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA'03*, 2003.

[29] P. R. Panda. SystemC: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[30] N. Rafla and D. Gauba. Hardware implementation of context switching for hard real-time operating systems. In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pages 1 –4, aug 2011.

[31] H. K.-H. So and R. Brodersen. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Trans. Embed. Comput. Syst.*, 7:14:1–14:28, January 2008.

[32] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[33] M. Vachharajani, N. Vachharajani, and D. I. August. The liberty structural specification language: a high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 195–206, New York, NY, USA, 2004. ACM.

[34] M. Vax. Conservative aspect-orientated programming with the e language. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 149–160, New York, NY, USA, 2007. ACM.

[35] T. Williams and K. Parker. Design for testability–A survey. *Proceedings of the IEEE*, 71(1):98–112, 1983.