SoCs under the Perspective of AOSD

Fauze Valério Polpeta, Antônio Augusto Fröhlich and Arliones Hoeller Jr
UFSC/CTC/LISHA
PO Box 476
88049-900 Florianópolis - SC, Brazil
{fauze,guto,arliones}@lisha.ufsc.br
http://www.lisha.ufsc.br/~{fauze,guto,arliones}

Abstract

This paper outlines a strategy for automating the genesis of embedded systems in the context of hardware and software components by generating automatically customized System-On-Chip*'s and the associated run-time support system for dedicated applications.*

We concentrate in the Hardware Mediator *construct, a portability artifact that relies on the* Application-Oriented System Design *to enable the port of component-based operating systems to very distinct architectures. Besides giving rise to a low-overhead system-hardware interface, hardware mediators can be used to dictate which* Intellectual Property *components could be instantiated into an* System-on-Chip *to satisfy, in terms of hardware devices, the requirements of a given application.*

The deployment of hardware mediators in the EPOS *system corroborates the portability claims associated to this software artifact and in which way it could enable the composing of* Application-Oriented System-on-Chip*'s in different application backgrounds.*

Keywords: AOSD - *Application-Oriented System Design, Portability, IP - Intellectual Property,* SOC - *System-on-Chip,* EPOS - *Embedded Parallel Operating System*

1. Introduction

Embedded systems are becoming more and more complex, yet, there is no room for development strategies that incur in extended time-to-market in this extremely competitive sector. In this context, *Programmable Logic Devices* (PLD) define a compromise between system complexity and development costs. Furthermore, recent advances in programmable logic devices are enabling developers to integrate complex designs into a single silicon pastille. These *Systems-On-Chip* (SOC) have several advantages over traditional circuit board implementations, including the level of integration, power consumption, maintainability and most other development metrics.

Nevertheless, getting a SOC out off a *Field-Programmable Gate Array* (FPGA) is not a trivial task and requires an intricate engineering process. Several strategies have been proposed for this purpose, having in common a methodology to devise components, usually referred to as *Intellectual Property* (IP), and to assembly these components together in a functioning system. The reusability of such soft IPs can be considerably increased by isolating their core functionalities from interfacing aspects, thus enabling the creation of libraries of pre-verified IPs that can be integrated as a SOC with the use of additional "glue logic".

In order to automate the development of SOCs, much effort has been payed to supporting tools that assist designers in selecting and configuring the most adequate IPs and also in generating the necessary glue logic. These tools usually presuppose an standardized interconnect, such as WISHBONE [10], AMBA [1] or CORECONNECT [4], and thus can be very effective. For instance, the CORAL [3] tool from IBM uses the concept of *Virtual Design* [8] to provide the designer with simplified IP descriptions that hide many implementation details.

Some *embedded systems* can be completely implemented in hardware using this approach, but the more complex the application, the greater is the probability it will need some kind of *run-time support system* and an *application program*. This is, after all, the reason why so many groups are concentrating efforts to develop processor soft cores [7, 9]. Nevertheless, run-time support systems are often neglected by currently available SOC development methodologies and tools, being mostly restricted to simple processor scheduling routines and the definition of a `software/hardware interface` [2, 5]. The gap between software and hardware gets even bigger when we recall that one of the primary goals of an operating system is to grant the portability of applications, since ordinary oper-

ating systems cannot go with the dynamics of SoCs.

In this paper, we consider the deployment of APPLICATION-ORIENTED SYSTEM DESIGN (AOSD) [15], a design methodology originally proposed for run-time support systems, to guide the complete development of embedded systems, including software and hardware components as well as their integration as a SoC. The results obtained so far are encouraging and will be discussed in details in the subsequent sections.

2. Application-Oriented System Design

The idea of building run-time support systems through the aggregation of independent software components is being used with claimed success in a series of projects [12, 14, 22, 11]. However, software component engineering brings about several new issues, for instance: how to partition the problem domain so as to model really reusable software components? how to select the components from the repository that should be included on an application-specific system instance? how to configure each selected component and the system as a whole so as to approach an optimal system?

Application-Oriented System Design (AOSD) [15] proposes some alternatives to proceed the engineering of a domain towards software components. In principle, an application-oriented decomposition of the problem domain can be obtained following the guidelines of *Object-Oriented Decomposition* [13]. However, some subtle yet important differences must be considered. First, object-oriented decomposition gathers objects with similar behavior in class hierarchies by applying variability analysis to identify how one entity specializes the other. Besides leading to the famous "fragile base class" problem [19], this policy assumes that specializations of an abstraction (i.e. *subclasses*) are only deployed in presence of their more generic versions (i.e. *superclasses*).

Applying variability analysis in the sense of *Family-Based Design* [21] to produce independently deployable abstractions, modeled as members of a family, can avoid this restriction and improve on application-orientation. Certainly, some family members will still be modeled as specializations of others, as in *Incremental System Design* [17], but this is no longer an imperative rule. For example, instead of modeling connection-oriented as a specialization of connectionless communication (or vice-versa), what would misuse a network that natively operates in the opposite mode, one could model both as autonomous members of a family.

A second important difference between application-oriented and object-oriented decomposition concerns environmental dependencies. Variability analysis, as carried out in object-oriented decomposition, does not emphasizes the differentiation of variations that belong to the essence of an abstraction from those that emanate from the execution scenarios being considered for it. Abstractions that incorporate environmental dependencies have a smaller chance of being reused in new scenarios, and, given that an application-oriented operating system will be confronted with a new scenario virtually every time a new application is defined, allowing such dependencies could severely hamper the system.

Nevertheless, one can reduce such dependencies by applying the key concept of *Aspect-Oriented Programming* [18], i.e. aspect separation, to the decomposition process. By doing so, one can tell variations that will shape new family members from those that will yield scenario aspects. For example, instead of modeling a new member for a family of communication mechanisms that is able to operate in the presence of multiple threads, one could model multithreading as a scenario aspect that, when activated, would lock the communication mechanism (or some of its operations) in a critical section.

Based on these premises, Application-Oriented Systems Design guides a domain engineering procedure (see figure 1 that models software components with the aid of three major constructs: families of scenario-independent abstractions, scenario adapters and inflated interfaces.

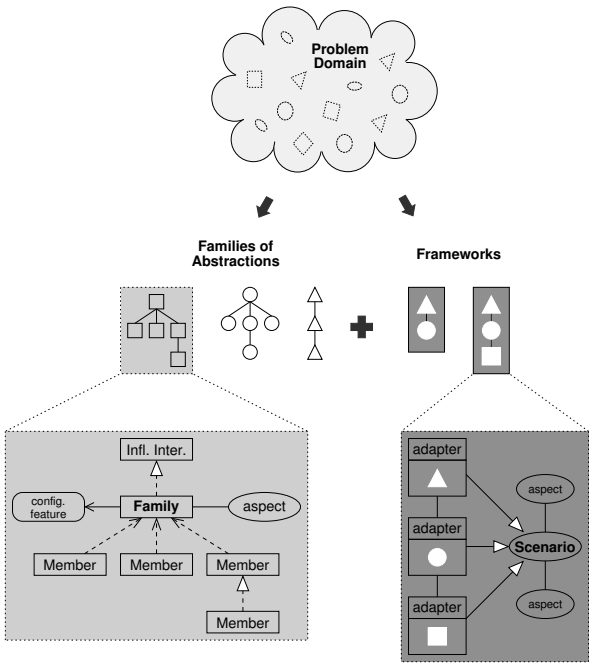

Figure 1. Overview of application-oriented domain decomposition.

Families of scenario independent abstractions

During domain decomposition, abstractions are identified from domain entities and grouped in families according to their commonalities. Yet during this phase, aspect separation is used to shape scenario-independent abstractions, thus enabling them to be reused in a variety of scenarios. These abstractions are subsequently implemented to give rise to the actual software components.

Scenario adapters

As explained earlier, AOSD dictates that scenario dependencies must be factored out as *aspects*, thus keeping abstractions scenario-independent. However, for this strategy to work, means must be provided to apply factored aspects to abstractions in a transparent way. The traditional approach to do this would be deploying an *aspect weaver*, though the *scenario adapter* construct [16] has the same potentialities without requiring an external tool. A scenario adapter wraps an abstraction, intermediating its communication with scenario-dependent clients to perform the necessary scenario adaptations.

Inflated interfaces

Inflated interfaces summarize the features of all members of a family, creating a unique view of the family as a "super component". It allows application programmers to write their applications based on well-know, comprehensive interfaces, postponing the decision about which member of the family shall be use until enough configuration knowledge is acquired. The binding of an inflated interface to one of the members of a family can thus be made by automatic configuration tools that identify which features of the family were used in order to choose the simplest realization that implements the requested interface subset at compile-time.

Hardware mediators

Hardware mediators are software constructs that mediate the interaction between abstractions and hardware components. The main idea behind hardware mediators is not to build universal hardware abstraction layers and virtual machines, but sustaining an *interface contract* between system and machine. Differently from ordinary HALs, hardware mediators do not build a monolithic layer encapsulating the resources available in the hardware platform. Each hardware component is mediated via its own mediator, thus granting the portability of abstractions that use it without creating unnecessary dependencies. Indeed, hardware mediators are intended to be mostly metaprogrammed and therefore dissolve themselves in the abstractions as soon as the interface contract is met. In other words, a hardware mediator delivers the functionality of the corresponding hardware component through a system-oriented interface.

Configurable features

Another important element of AOSD are *configurable features*, which designate features of components that can be switched on and off according to the requirements dictated by abstractions. A configurable feature is not restricted to a flag indicating whether a preexisting feature must be activated or not. Usually, it also incorporates a *Generic Programmed* [20] implementation of the algorithms and data structures that are necessary to implement that feature when the component itself does not provide it. An example of configurable feature is the generation of CRC codes in an Ethernet mediator.

Component framework

A *component framework* captures a reusable software architecture by defining how abstractions can be arranged together in a functioning system. Reusable system architectures are usually defined considering the domain knowledge acquired during the design of previous systems. After having developed some systems, or some versions of a system, for a certain problem domain, developers begin to agree on how to implement the abstractions that build up the domain; how they interact with each other, with the environment, and with applications; and how the implied non-functional requirements can be accomplished. Such an expertise can be captured in an architectural specification to be reused in upcoming systems.

Capturing reusable system architectures in a component-based system is fundamental, since a real component-based system is only achieved when components can be arranged together in an assemblage of predictable behavior. In AOSD, reusable architectures begin to be modeled yet during domain decomposition with the identification of relationships between families of abstractions. These relationships are latter enriched by scenario constraints during the specification of scenario aspects to yield a component framework in the form of a collection of interrelated scenario adapters. Each scenario adapter acts as a "socket" for components of the corresponding family. Plugging components into the framework is accomplished by binding the inflated interface of every used family to the desired family member. The way scenario adapters are arranged in the framework would define the basic architecture of resultant systems, while architectural elements that do not concern components could be hard-coded in the framework.

3. AOSD and SoCs

Although originally devised to guide the development of software components of the operating system domain, the main concepts behind the *Application-Oriented System Design* methodology can also be deployed on the context of hardware components. More specifically, on the context of programmable logic, where hardware components are described using high-level languages such as VHDL, VERILOG and SYSTEMC in order to abstract the physical elements of the underling hardware platform technology. When instantiated, these components give rise to a program *bitstream*, which in its turn represents a substantial advance when compared to the lithography masks used to build hard ASIC components.

The creation of a software component considering the AOSD methodology is drived by four main stages: domain analysis, domain decomposition, component design and its implementation. Indeed, all the stages can be used without any modification to develop hardware components. For comparison, it is perfectly feasible to think about modeling a set of hardware devices as a family of hardware components, and, these components are nothing more than a set of HDL (Hardware Description Language) files, i. e., a *software*.

In a future not so far, as an outcome of the consolidation of higher-level languages for hardware description such as SYSTEMC, there will also be possible to employ, in the implementation of the hardware components, techniques that allow these components to be much more organized and offer easier maintainability. There can also be provided, through the use of advanced techniques such as Aspect-Oriented Programming, Static Meta-Programming, Subject-Oriented Programming and Generic-Programming, an enhanced configurability, what will make application-orientation much more feasible than now. Another important outcome of this consolidation is a possible easier integration between software and hardware source-code.

4. SoCs in EPOS

Until this moment, we conjectured about portability in component-based operating systems, focusing in the *hardware mediator* construct, which, differently from hardware abstraction layers and virtual machines, have the ability to establish an interface contract between the hardware and the operating system components and yet generate virtually no overhead.

The use of hardware mediators in the `Epos` system corroborated the portability claims associated to the techniques explained and enabled EPOS to be easily ported across very distinct architectures without any modification in its software components. For each architecture, only the respective set of hardware mediators has to be implemented. However, the deployment of *Application Oriented System Design* on the context in which hardware mediators are inserted—software-hardware interfacing—fosters this portability artifact to a new perspective on the genesis of embedded systems.

Under the scope of AOSD, hardware mediators, when instantiated, can be viewed as a summary of which hardware components are necessary to support an application. In this way, by extending the concept of hardware operating system interface to a level that would enable hardware generation, mediators are also figurated as pointers for generating a "machine description" that matches, as does the operating system, the application's requirements. Besides implementing hardware interfacing components, mediators can be associated with pre-designed and pre-verified hardware description components. Such mediators, when instantiated, will give rise not only to a system-hardware interface, but to an application-oriented hardware instance in the form of a *System-on-a-Chip*. [1].

In order to evaluate this new approach, the `Epos` system was taken again. The strategy used in EPOS to describe software components in a repository and their dependencies plays a key role in making the process of generating application-oriented SoCs possible. The description of the components holds the information necessary to identify which abstractions and hardware mediators satisfy the application's requirements and more: which of these mediators are associated with hardware components. A typical description of a family of mediators is depicted in the figure 2.

Perhaps, considering now that the mediator related to the `OR1200 CPU` core can be associated with a hardware component or, in other words, an *IP* [2], which can be instantiated in a programmable logic device to realize the same functionality that a `OR1200` core chip does. To ratify this peculiarity in the mediator description, a non-functional property named *synthesizable* is specified. As illustrated in the figure 3 this property, classified as a *feature*, can designate other non-functional properties. In this case two *dependences*, which in turn indicate the hardware components that compose the `OR1200` CPU core.

As regards to the description of this class of components, it follows the same principles of describing system abstractions and hardware mediators. However, due to the very own nature of the traditional *hardware description lan-*

1 A *System-on-a-Chip*, also referred as SoC, is an apparatus able to deliver the same functionality that a traditional hardware platform does when connecting several electronic components together on a circuit board, but, differently from the traditional ones, it is encapsulated in a single-chip pastille.

2 IP is the abbreviation for *Intellectual Property*. It refers to a description of a eletronic circuit that can later be reduced to an on-chip hardware device through a synthesis process.

```
<family name="CPU" type="mediator" class="dissociated">
 <interface>
  <constructor/>
  <method name="clock" return="Hertz" qualifiers="class"/>
  <method name="int_enable" qualifiers="class"/>
  <method name="int_disable" qualifiers="class"/>
  <method name="switch_context" qualifiers="class">
   <parameter name="current" type="Context * volatile *"/>
   <parameter name="next" type="Context * volatile"/>
   ...
 </interface>
 <common>
  <type name="Reg8" type="synonym"
        value="unsigned char"/>
  <type name="Reg16" type="synonym"
        value="unsigned short"/>
  <type name="Reg32" type="synonym"
        value="unsigned long"/>
  <type name="Reg64" type="synonym"
        value="unsigned long long"/>
 </common>
 <member name="IA32" type="exclusive">
  <method name="clock" return="Hertz"/>
  <method name="int_enable"/>
  <method name="int_disable"/>
  <method name="switch_context" qualifiers="class">
   <parameter name="current"
        type="Context * volatile *"/>
   <parameter name="next" type="Context * volatile"/>
  </method>
  ...
 <member/>
 <member name="OR1200" type="exclusive">
  <method name="clock" return="Hertz"/>
  <method name="int_enable"/>
  <method name="int_disable"/>
  <method name="switch_context" qualifiers="class">
   <parameter name="current"
        type="Context * volatile *"/>
   <parameter name="next" type="Context * volatile"/>
  </method>
  ...
 <member/>
</family>
```

Figure 2. A fragment of the CPU hardware mediator family description.

guages in not implementing the object oriented paradigm [3], the description of hardware component is essencially focused in non-functional properties and configurable features. The figure 4 presents a component family that features, among other CPU components, those that compose the OR1200 core. As illustrated, both the OR1200_I and

3 The traditional hardware description languages are focused in the VHDL and VERILOG approachs. Otherwise, there is a substantial effort to bring to the context of these languages the design paradigms used in software programming.

```
<family name="CPU" type="mediator"
        class="dissociated">
  ...
  <member name="OR1200">
    <method name="clock" return="Hertz"/>
    <method name="int_enable"/>
    <method name="int_disable"/>
    <method name="switch_context" qualifiers="class">
      <parameter name="current"
           type="Context * volatile *"/>
      <parameter name="next" type="Context * volatile"/>
    </method>
    ...
    <feature name="synthesizable" value="true">
      <dependency name="::CPU_IP::OR1200_I"
           value="true"/>
      <dependency name="::CPU_IP::OR1200_D"
           value="true"/>
    </feature>
  </member>
  ...
</family>
```

Figure 3. A fragment of the hardware mediator for the synthesizable OR1200 CPU core.

OR1200_D have a feature named *interconnection*. This feature specifies the on-chip bus(es) on which the component can be compatibly connected. The subsequently configurable features, in the form of *traits*, aggregate the necessary information that will be passed to a SoC building tool in order to perfectly enable the integration of each IP when it is tailored into a SoC.

Nevertheless, new features or dependencies could be specified inside the IP descriptions. The purpose of non-functional properties have been specifying features that cannot be directly deduced from the component's interface. Consider, for instance, that all the IPs that will be instantiated for a given application have to realize the same on-chip bus standard to be glued together into a SoC. The already mentioned feature *interconnection* enables us to ensure this peculiarity yet at system configuration time.

However, classifying a hardware mediator as *synthesizable* does not implies that it will always instantiate the associated IP components. It depends on a *machine description* to determine, as soon as the application's requirements are matched, which soft IPs can be synthesized and which are already instantiated as hard cores in hardware. Indeed, this machine description figurates an complete overview of the hardware platform, including the devices that it compromises and how these devices are disposed on it. The figure 5 depicts a description of a hardware platform that is all based on a programmable logic device, which means that each one of the system's mediators is associated with an IP that will be instantiated as soon as the application requires

```
<family name="CPU_IP" type="hardware"
        class="dissociated">
  <interface>
  </interface>
  <common>
  </common>

  <member name="OR1200_I">
    <feature name="interconnection" value="wishbone"/>
    < trait  name="readwrite/" type="bool" value="false"/>
    < trait  name="lock_o"     type="int"   value="0"/>
    < trait  name="tga_o"      type="int"   value="1"/>
    < trait  name="tgc_o"      type="int"   value="1"/>
     ...
  </member>

  <member name="OR1200_D">
    <feature name="interconnection" value="wishbone"/>
    < trait  name="readwrite " type="bool" value="true"/>
    < trait  name="lock_o"     type="int"   value="0"/>
    < trait  name="tga_o"      type="int"   value="1"/>
    < trait  name="tgc_o"      type="int"   value="1"/>
     ...
  </member>
  ...
</family>
```

Figure 4. A fragment of a family of CPU hardware description components

it.

Yet in respect to the machine description illustrated in the figure 5. The element named *synthesis* in the machine description has a crucial importance in the configuration process. It summarizes all the global synthesis properties that must be matched as features in the hardware descriptions components that can be isntantiated as hardware devices of this machine. Whithout this summary, the features of each component would need to be matched among all the IPs. The mentioned compatibility of on-chip bus between the IPs figurates a good example of it. Another feature could be language used to describe the hardware components (VHDL, Verilog...), specially if one wants to keep the homogeneity in the implementation of the hardware components that compose the machine. However considering that the synthesis process will reduce the hardware components to a gate-level description, this feature becomes optional and therefore, an own user decision.

Additionally, besides featuring a wide-ranging specification strategy to describe software components from the operating system and hardware domain, EPOS counts with a development environment from which many different application-oriented instances can virtually arises. An overview of this infrastructure is depicted in the figure 6.

```
<machine name="ORSoC">
  <synthesis interconnection="wishbone"/>

  <processor name="OR1200" clock="300000000"
      word_size="32" endianess="big"
      synthesizable="true"/>

  <memory name="sram" base="0" size="0x100000"
      synthesizable="true">
    < trait  name="readwrite" type="bool" value="true"/>
    < trait  name="adr_i_hi" type="int"  value="11"/>
    < trait  name="adr_i_lo" type="int"  value="2"/>
    < trait  name="tga_i" type="int"  value="0"/>
    < trait  name="tgc_i" type="int"  value="0"/>
    < trait  name="tgd_i" type="int"  value="0"/>
    < trait  name="lock_i" type="int"  value="0"/>
    < trait  name="err_o" type="int" value="0"/>
    < trait  name="rty_o" type="int" value="0"/>
  </memory>

  <bus name="wishbone" synthesizable="true">
    < trait  name="endofburst" type="int" value="111"/>
    < trait  name="dat_size" type="int" value="32"/>
    < trait  name="adr_size" type="int" value="32"/>
    < trait  name="muxtype" type="int" value="ANDOR"/>
    < trait  name="classic" type="int"  value="000"/>
    < trait  name="tga_bits" type="int"  value="2"/>
    < trait  name="tgc_bits" type="int"  value="3"/>
    < trait  name="tgd_bits" type="int"  value="0"/>
    < trait  name="rename_tga" type="int" value="BTE"/>
    < trait  name="rename_tgc" type="int" value="CTI"/>
    < trait  name="rename_tgd" type="int" value="TGD"/>
    < trait  name="signal_groups" type="int" value="1"/>
  </bus>

  <device name="UART" bus=""wishbone" class="Serial"
      synthesizable="true"/>
</machine>
```

Figure 5.

As depicted in the figure 6, the primary specification produced is resulted from the application's analysis. The `analyzer` scans the application searching for references to the inflated interfaces [4] and outputs an specification of requirements in the form of partial component interface declarations, including methods, types and constants that were used by the application. This specification feeds a second tool, the `configurator`, which consults a component description database to build a dependency three. The output of this configuration process consists of a set of keys that define the binding of inflated interfaces to abstractions and the definition of which IPs (if the application and the platform enabled it) will be instantiated.

The last step in the generation process is accomplished

4 Inflated interfaces summarize the features of all members of a family, creating a unique view of the family as a "super component"

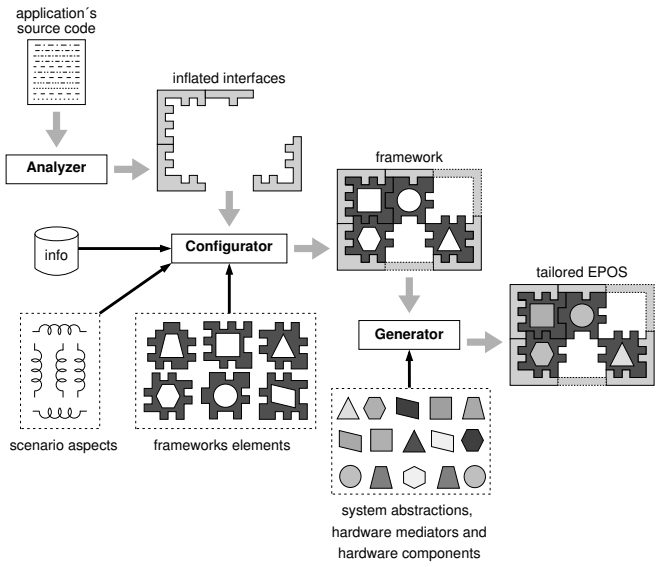

Figure 6. An overview of the Epos development environment.

by the `generator`. This tool translates the keys produced by the `configurator` into parameters for a statically metaprogrammed component framework and causes the compilation of a tailored operating system instance. On the side of the hardware, if a SoC needs also to be tailored, the `generator` counts with a *SoC Builder* module that, using the configurable features of the selected IPs, will logically glue these components into a single synthesizable description that later can be instantiated in a programmable logic device such as a FPGA.

5. Case Study: OpenRISC 1200 Memory System

Aiming the validation of these concepts, an analysis was made to identify the effects of using this methodology on adapting the processor's cache size and the number of TLB (Translation Lookaside Buffer) entries to the application needs. The chosen architecture was the OPENRISC 1200, a 32-bit scalar RISC processor based on the Harvard microarchitecture, very similar to a POWERPC [7]. This soft core is organized in a way that allows the programmer to change the sizes of the processor's caches and TLBS.

Experimental results showed that this core, configured with 8 KBytes of cache memory (4 KBytes for instructions and 4 KBytes for data), needs 772 more LUTS (Look-Up Tables) than the another one with the same configuration just without any cache. This space can be very useful when dealing with small FPGA (Field-Programmable Gate-Array) devices, once the saved gates can be used to build, for example, another microcontroller, or can even allow the use of a smaller device, making the system cheaper. However, the definition of the optimum size for the `CPU` cache must be done through a workflow analysis, so it cannot be achieved automatically.

Nevertheless, the EPOS configuration tools will always know the size of the software which will run over this SoC, so it can configure the system's MMU (Memory Management Unit) to have only the amount of TLB entries needed by the application. It is known that the number of LUTS that will be saved through this configuration is not so expressive as the ones saved by changing the cache sizes, but what must be taken into account here is which gates we are saving. The TLB implementation uses special purpose gates that are built to make comparisons faster. Most FPGA devices have a limited amount of this kind of gates, and some don't even implement it, so avoid using them is a good design improvement.

Another complex issue to deal within this context is the software support for a dynamically configurable MMU. Most operating systems implements monolithic drivers to handle this devices. The memory manager of the EPOS system is not implemented this way. The use of several resources such as Aspect-Oriented Programming and Static Meta-Programming, and the development following the AOSD methodology generated a very efficient and easily portable software support for a wide variety of memory management strategies. This memory system is better described in the respective work [6].

The table 1 shows the obtained numbers of the built SoCs. All of them were designed supposing a system with a 32-bit processor (OPENRISC 1200) and a serial port (UART16550). The first prototype had exactly this configuration, i. e., it didn't have any cache memory or MMU. The other two configurations imposed the use of a cache memory and implemented a MMU with different TLB sizes (64 and 128 entries). The gain on changing this configuration isn't too good (54 LUTS), but it is enough to build, for instance, another serial port. Another important issue to consider is that, as explained above, saving these gates is important to allow other devices to use the comparison gates.

Prototype	LUTs
BASIC	5,866
BASIC + 8KB CACHE + 64 ENTRIES TLB	6,638
BASIC + 8KB CACHE + 128 ENTRIES TLB	6,692

Table 1. The size (in LUTs) of the synthesized prototypes.

6. Conclusions and Future Work

This paper presented the results obtained until this moment from the deployment of APPLICATION-ORIENTED SYSTEM DESIGN (AOSD) as a design methodology to develop complete embedded systems, including software and hardware components, for it generating a hardware instance in the form of a SOC and the correspondent run-time software support, the both fulfilling the application's requirements.

The experiments were concentrated on instantiating a customized OpenRISC-based SOC under different application contexts. The EPOS system provided the appropriated support for these experiments by enabling the description of the system components and their respective integration.

The results are so good that we decided to extend the project to a level where we can, not only statically, but also dynamically adapt the hardware and software instances in according with the applications demand.

References

[1] Advanced RISC Machines Limited. *The de facto Standard for On-Chip Bus*, online edition, 2003. [http://www.arm.com/products/solutions/AMBAHomePage.html].

[2] A. Corporation. Sopc builder. Technical report, Altera Corporation, http://www.altera.com/sopcbuilder, may 2004.

[3] R. A. B. et. Al. Coral - automating the design of systems-on-chip using cores. In *IEEE Custom Integrated Circuits Conference*.

[4] IBM Microeletronics. *CoreConnect Bus Architecture*, online edition, 1999. [http://www.ibm.com/chips/products/coreconnect/].

[5] X. Inc. Xilinx ise 6 software manuals and help. Technical report, Xilinx Inc., http://www.xilinx.com/support/sw_manuals/xilinx6, may 2004.

[6] A. S. Hoeller Jr. Famílias de abstrações de gerência de memória para o EPOS. B. Sc. Thesis, Universidade Federal de Santa Catarina, feb 2004.

[7] D. Lampret. *OpenRISC 1200 IP Core Specification*. OpenCores, 0.6 edition, jun 2001.

[8] B. Reinaldo A. and W. R. Lee. Designing system-on-chip using cores. In *Embedded Tutorial Paper at the 37th ACM/IEEE Design Automation Conference*.

[9] G. Research. *LEON2 XST User's Manual*. Gaisler Research Laboratory, 1.0.23 edition, may 2004.

[10] Silicore Corporation. *The WISHBONE Service Center*, online edition, 2003. [http://www.silicore.net/wishbone.htm/].

[11] L. Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Germany, June 1999.

[12] 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*, St Malo, France, May 1999.

[13] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[14] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

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

[16] 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, U.S.A., July 2000.

[17] A. N. Habermann, L. Flon, and L. W. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[18] 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, Jyväskylä, Finland, June 1997. Springer.

[19] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer.

[20] D. R. Musser and A. A. Stepanov. Generic Programming. In *Proceedings of the First International Joint Conference of ISSAC and AAECC*, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.

[21] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Mar. 1976.

[22] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, U.S.A., Oct. 2000.