On AOP techniques for C++-based HW/SW component implementation

Tiago Rogério Mück and Antônio Augusto Fröhlich
Federal University of Santa Catarina
Florianópolis, Brazil
E-mail: {tiago,guto}@lisha.ufsc.br

***Abstract*—The increasing complexity of embedded system applications is leading to a convergence between hardware and software development. In this paper we aim to close the gap between hardware and software implementation by proposing guidelines for handling both domains in a unified fashion. We leverage on aspect-oriented programming (AOP) concepts to provide unified C++ descriptions that can be both compiled to software or synthesized to dedicated hardware using high-level synthesis tools. Our results show that our strategy leads to reusable components at the cost of a small overhead when compared to software-only and hardware-only C++ implementations.**

***Index Terms*—System-level design, HW/SW co-design, High-level synthesis, Aspect-oriented system design.**

I. Introduction

Current embedded applications are becoming more sophisticated as the advances of the semiconductor industry allows the use of an increasingly amount of computational resources. Also, strict time-to-market requirements of most applications demand a better productivity, pushing embedded system designs to the *system-level*. In this scenario, a convergence between hardware and software design methodologies is desirable, since a unified modeling approach would enable one to take decisions about hardware/software partitioning later in the design process, maybe even automatically. In the last few years, advances in *electronic design automation* (EDA) tools are allowing hardware synthesis from high-level, software-like descriptions. This process is known as *high-level synthesis* (HLS) and allows designers to describe hardware components using languages like C++, and higher-level techniques, such as *Object-Oriented Programming* (OOP). The focus of these tools, however, is hardware synthesis, and they do not provide a clear design methodology for developing components that could be reused across the hardware and software domains.

Aiming to narrow this gap, in this paper we describe some design guidelines and mechanisms to support the implementation of both hardware and software components from a single C++ description. Our guidelines are built upon the *Application-driven Embedded System Design* (ADESD) methodology [1]. ADESD leverages on OOP and *aspect-oriented programming* (AOP) concepts, defining a domain engineering strategy which allows us to clearly separate the core behavior and the structure of a component from aspects that must be handled differently whether a component is implemented as hardware or software. In order to generate descriptions that can be efficiently synthesized by HLS tools or compiled to a software binary, the implementation of both the components and the mechanisms which adapt them make extensive use of C++ *generative programming* [2] techniques such as *static metaprogramming*.

To evaluate the feasibility of our approach, we use a *Private Automatic Branch Exchange* (PABX) application of which some components were reimplemented using our unified design strategy.

II. Related work

Several design methodologies and tools were proposed in order to provide more tightly coupled hardware and software design flows. Most of these methodologies were initially based on the concept of building a system by assembling pre-validated components [3], [4], [5], which imposed limitations on terms of hardware/software partitioning. Current design flows leverage on state-of-the-art EDA tools to support hardware synthesis from high-level C++ constructs. On this track, the OSSS+R methodology [6] uses cycle-accurate SystemC and adds new language constructs to support synthesizable polymorphism and high-level communication. However, hardware/software partitioning must still be done early in the design process[7], and the inclusion of non-standard language constructs reduces compatibility with available compilers and synthesis tools. The *Saturn* [8] design flow also contributes in this scenario, but follows a different approach. It aims to close the gap between UML-based modeling and the execution of the models for their verification. The authors have elaborated over *SysML*, an extension of UML for system-level design, and developed a tool which generates C++ for software and RTL SystemC for hardware. SystemCoDesigner [9] is a tool which integrates a HLS flow with design space exploration. The design entry of SystemCoDesigner is an actor-based data flow model implemented using SystemC. After design space exploration, actors of this model can be converted to synthesizable SystemC or to C++ for software compilation. However, as the authors themselves claim, SystemCoDesigner targets mostly data-flow-based applications, and they do not provide directions towards a more general deployment. The System-on-chip environment (SCE) [10] takes SpecC models as input and provides a refinement-based tool flow. Guided by the designer, the SCE automatically generates a set of

Transaction-level models (TLM) [11] that are further refined to pin- and cycle-accurate system implementation.

The ADESD [1] methodology elaborates on commonality and variability analysis 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. Characteristics of specific execution scenarios are modeled in special constructs called *aspects* and are applied to the components only during the final system assembly. In a previous work [12], we have already shown how the ADESD methodology was used to provide a C++ description of a resource scheduler suitable for HLS. Throughout the design process, two basic aspects were identified that required a different approach in the final hardware/software implementation of the scheduler: *resource allocation* and *communication interface*. In the next sections we show how one can further leverage on ADESD's artifacts to provide C++ unified descriptions and efficient hardware/software aspects separation.

III. HW/SW ASPECT ENCAPSULATION

C++ code unified and suitable for automatic implementation in both hardware and software must follow a careful design process so it will not contain characteristics specific of hardware or software. Throughout the domain decomposition process in our previous work [12] we have identified two main features that distinguish hardware from software: *1)storage allocation*. In hardware, dynamic features such as dynamic memory allocation are not available, therefore, in synthesizable C++ code all data structures must reside in statically allocated memory. And *2) communication interface*. The top-level interface of the resulting hardware block is usually inferred by HLS tool from a single method or function that serves as an entry-point, whereas in software, components operation can be requested directly through their method call interface.

Such differences can be efficiently encapsulated using the *aspect* concept of AOP, and weaved with the unified C++ descriptions in order to obtain an implementation specific for a hardware or software implementation flow. ADESD proposes the use of constructs called *Scenario adapters* [13] to perform aspect weaving. 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. Figure 1 shows how this is achieved. The *SW Scenario* and *HW Scenario* classes represent the software and hardware execution scenarios, respectively, and incorporate, via aggregation, all aspect programs that are needed to characterize them. The adaptation of the component to the scenario is performed by the *Scenario_Adapter* class via a conditional inheritance implemented using static metaprogramming.

The *HW Scenario* is composed by the aspects *Static Alloc* and *Dispatch*, whom are responsible, respectively, for storage allocation and method call dispatch. The former is a storage allocator used to deal with the absence of dynamic memory allocation in hardware. All components operations go through

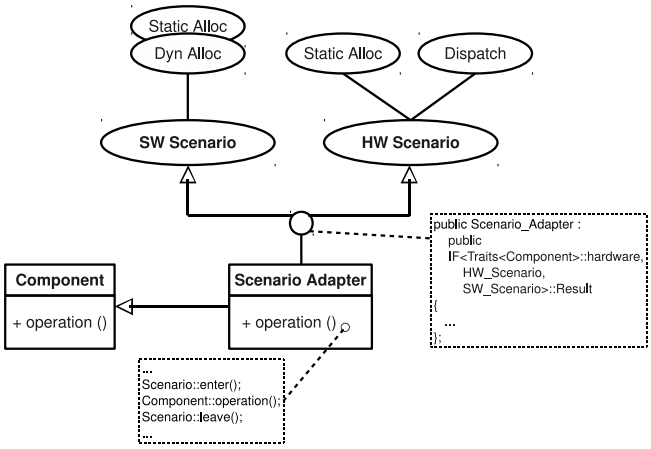

Figure 1. UML diagram showing component adaptation using scenario adapters

the allocator which reserves and releases storage space on demand. It is important to mention that handling memory allocation externally is only possible if the component follows a careful design process that removes this feature from its core implementation. For example, the resource scheduler described in our previous work [12] uses a linked list to implement its core behavior. This list is designed in a way it is not responsible for the allocation of space for links and the objects it stores. It implements only the list algorithms and deals with references to such elements. The example below shows how the *insert* and *remove* methods of the scheduler are redefined inside the scenario adapter using this approach:

```
Link insert(Object obj) {
    Link link = Scenario::allocate(obj);
    Component::insert(link);
    return link;
}
Object remove(Link link) {
    Object obj = Scenario::get(link);
    Component::remove(link);
    Scenario::free(link);
    return obj;
}
```

In hardware implementations, *Scenario::allocate(obj)*, *Scenario::get(link)*, and *Scenario::free(link)* map to operations implemented in the *Static Alloc* aspect. In this scenario, the number of storage slots for links and objects is defined at synthesis-time and allocation requests are just mapped to a free slot. In a software implementation, dynamic memory allocation is available, thus, storage allocation can be handled by either the *Static Alloc* or the *Dynamic Alloc* aspect, as shown in Figure 1.

The *Dispatch* aspect is used, in *HW Scenario*, to define an entry point for the component so it will be compliant with HLS tools requirements. For the tool used in our case studies (Calypto's CatapultC [14]), the top-level interface of the resulting hardware block (port directions and sizes) is inferred from a *single function signature*. This function is defined by *Dispatch* and receives a *method id* as its first parameter, interprets its value, performs the necessary type conversions and calls the

appropriate method of the component. The value returned by the called method is also inspected, converted if necessary and assigned to one of the dispatcher output parameters. A dispatch mechanism is not necessary in the software scenario since operations are requested using direct method calls.

Finally, it is worth mentioning that aspects, scenario, and adapters are implemented using static metaprogramming and are highly generic. The scenario to which the component is adapted is, in fact, defined using metaprogramming, as shown in Figure 1. Special template classes called *Traits* are used to define which characteristics of each component is activated. The code sample below shows an example of a *Trait* class:

```
template <> struct Traits<Component> {
    static const bool hardware = true;
};
```

It defines that the component *Component* has a *hardware* characteristic that is used to define which domain the component will be on (hardware or software). In Figure 1, this characteristic is used to conditionally modify the scenario adapter's base scenario. This decision is statically determined using a metaprogram. The implementation of *IF* metaprogram is shown below:

```
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; };
```

Other aspects concerning hardware generation using HLS are related to the synthesis process. The same high-level algorithm can span several different hardware implementations. For instance, loops can have each iteration executed in a clock cycle, or can be fully unrolled in order to increase throughput at the cost of additional silicon area. This kind of synthesis decision is usually taken based on directives which are provided separately from the algorithm descriptions. The definition and fine tuning of these directives is part of the design space exploration process and is not in the scope of this work.

IV. Experimental results

In order to evaluate our approach, we have applied its mechanisms in the implementation of the following components of a PABX application: the thread scheduler of EPOS [15] operating system, which has been mentioned in Section III and described in our previous work [12]; a 16-bit IMA ADPCM encoder/decoder that is used to reduce the traffic of voice data transmitted through the system; and a *Dual-Tone Multi-Frequency* (DTMF) detector that uses the *goertzel algorithm* to check if a sample frame contains specific frequency components.

In order to demonstrate that unified implementations can be compared to dedicated ones in terms of efficiency, we compare software scenario-adapted components against the original C++ implementations, and hardware scenario-adapted components against components manually tailored for high-level synthesis. The scenario-adapted implementations are feed to their respective software/hardware flows as shown in Figure 2. In the software implementation flow, we have used *gcc 4.0.2* targeting the *Plasma softcore* (a MIPS32 implementation) and using *level 2* optimizations. In the hardware flow, the scenario-adapted code is feed to *Calypto's CatapultC* [14] HLS tool in order to obtain RTL descriptions of the components. These descriptions were then synthesized using *Xilinx's ISE 13.4* targeting a *Virtex6 XC6VLX240T* FPGA. CatapultC and ISE were configured to minimize circuit area considering a target operating frequency of 100 MHz.

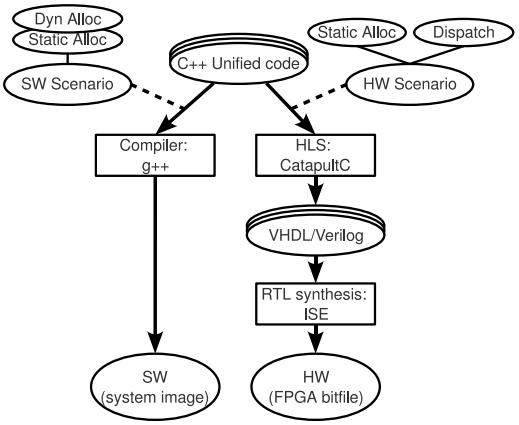

Figure 2. Implementation steps and tools used, for both hardware and software flows

Table I compares the software/hardware-only C++ with the software/hardware scenario-adapted C++ in terms of the *worst case execution time* (WCET) of each operation. The software WCET of the Scheduler and the ADPCM codec is about 2.5% higher in the unified implementation. For the DTMF detector, the difference increases to 5%. The original DTMF detector requires a single call to *do_dtmf* to analyze a frame of samples, while the in the refactored unified DTMF detector, several calls to an *add_sample* method are required before performing the same task, which results in a more significant increase in the execution time. For the hardware side, the *Dispatch* aspect created an overhead which is proportional to the number of arguments in the operation (*e.g.* 2 for *Scheduler::insert(thread,priority)* and 1 for *ADPCM_Codec::encode(sample)*). Analogous to its software counterpart, the high overhead of the unified DTMF detector also comes from the additional method invocations required to fill its internal buffer. This operation is implemented in a stream-like fashion in the HW-only detector.

Table II shows the memory footprint for software implementations and the amount of FPGA resource required by the hardware implementations. There is an average increase of about 4.9% in the total memory footprint. In the case of the Scheduler, the overhead comes from the generalization of storage allocation (the *Static/Dyn Alloc* aspect). For the remaining case studies, most of the overhead comes from additional code required to encapsulate the behavior into more reusable OOP classes with a clear method interface. In the hardware implementations, there is an absolute increase in the number *look-up tables* (LUTs) and *flip-flops* (FFs) which

is proportional to the number of operations that must go through the *Dispatch* aspect. This aspect implement a generic mechanism for parsing and issuing operation requests, while the hardware-only descriptions focused on more specific and optimized interfaces. The amount of other dedicated FPGA resources (*e.g.* DSP and RAM blocks) are not shown since the introduction of scenario adapters affect only the number of LUTs and FFs.

Table I
WCET OF SOFTWARE/HARDWARE-ONLY C++ *vs.* UNIFIED C++ ADAPTED TO SOFTWARE/HARDWARE.

Component		Software (μs)		Hardware (cycles)	
		SW-only	Unified	HW-only	Unified
Scheduler	insert	6.0	6.0	19	21
	remove	2.9	3.3	10	11
	suspend	3.0	3.1	10	11
	resume	6.0	6.0	19	20
	choose	8.4	8.5	10	11
ADPCM	encode	4.2	4.2	2	3
	decode	3.4	3.6	2	3
DTMF	do_dtmf	5878.3	6182.9	7041	7741

Table II
AREA FOOTPRINT OF SOFTWARE/HARDWARE-ONLY C++ *vs.* UNIFIED C++ ADAPTED TO SOFTWARE/HARDWARE.

Component	Software (code/data/total)		Hardware (LUT/FF)	
	SW-only	Unified	HW-only	Unified
Scheduler	2344 / 144 / 2488	2436 / 192 / 2628	2119 / 1849	2540 / 2766
ADPCM	632 / 436 / 1068	708 / 440 / 1148	524 / 208	615 / 368
DTMF	508 / 3144 / 3652	564 / 3152 / 3716	387 / 331	443 / 431

V. CONCLUSION

In this paper we have explored a methodology based on AOP and OOP concepts in order to produce unified descriptions of hardware and software components. We have shown that components designed following the principles presented in this work are susceptible to both software and hardware generation using standard compilers and HLS tools. This is possible through the isolation of specific hardware and software characteristics (resource allocation and communication interface) into aspect programs which are weaved with the unified descriptions only during the final implementation stages of the design process. Furthermore, our mechanisms are implemented using only standard C++ features, thus facilitating compatibility with different C++/C-based HLS tools.

Finally, we have demonstrated our methods by redesigning some functional blocks of a PABX system. The resulting components confirmed that, at an acceptable cost in area and performance, we can use C++ as a unified language to implement both hardware and software in an straightforward way, thus reducing the costs of design cycles and time-to-market, and contributing to the progress of embedded system design towards system-level methodologies.

ACKNOWLEDGMENTS

The authors would like to thank João Paulo Pizani Flor and Marcelo Daniel Berejuck for providing the original implementations of the some of the case studies used in this work. This work is partially supported by CAPES, under grants RH-TVD 006/2008 and 240/2008.

REFERENCES

[1] F. V. Polpeta and A. A. Fröhlich, "On the Automatic Generation of SoC-based Embedded Systems," in *Proc. of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, Sep 2005.

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

[3] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design & Test of Computers*, vol. 18, pp. 23–33, 2001.

[4] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "PeaCE: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 24:1–24:25, May 2008.

[5] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu, "A Next-Generation Design Framework for Platform-based Design," in *Proc. of the Design & Verification Conference & Exhibition*, February 2007.

[6] A. Schallenberg, W. Nebel, A. Herrholz, P. A. Hartmann, and F. Oppenheimer, "OSSS+R: a framework for application level modelling and synthesis of reconfigurable systems," in *Proc. of the Conference on Design, Automation and Test in Europe*, Nice, France, 2009, pp. 970–975.

[7] K. Grüttner, F. Oppenheimer, W. Nebel, F. Colas-Bigey, and A.-M. Fouilliart, "SystemC-based modelling, seamless refinement, and synthesis of a JPEG 2000 decoder," in *Proc. of the conference on Design, automation and test in Europe*, Munich, Germany, 2008, pp. 128–133.

[8] F. Mischkalla, D. He, and W. Mueller, "Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems," in *Proc. of the Conference on Design, Automation and Test in Europe*, Dresden, Germany, 2010, pp. 1201–1206.

[9] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 1:1–1:23, January 2009.

[10] D. Rainer, G. Andreas, P. Junyu, S. Dongwan, C. Lukai, Y. Haobo, A. Samar, D. Daniel *et al.*, "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP Journal on Embedded Systems*, vol. 2008, 2008.

[11] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Newport Beach, USA, 2003, pp. 19–24.

[12] J. P. P. Flor, T. R. Mück, and A. A. Fröhlich, "High-level Design and Synthesis of a Resource Scheduler," in *Proc. of the 18th IEEE International Conference on Electronics, Circuits, and Systems*, Beirut, Lebanon, 2011, pp. 736–739.

[13] T. R. Mück, M. Gernoth, W. Schröder-Preikschat, and A. A. Fröhlich, "Implementing OS Components in Hardware using AOP," *SIGOPS Operating Systems Review*, vol. 46, no. 1, pp. 64–72, 2012.

[14] Calypto Design Systems, "CatapultC Synthesis," 2011, http://www.calypto.com/.

[15] The EPOS Project, "Embedded Parallel Operating System," 2011, http://epos.lisha.ufsc.br/.