High-level Design and Synthesis of a Resource Scheduler

João Paulo Pizani Flor, Tiago Rogério Mück, Antônio Augusto Fröhlich
Software/Hardware Integration Lab
Federal University of Santa Catarina
Florianópolis, Brazil
Email: {joaopizani,tiago,guto}@lisha.ufsc.br

***Abstract*—Given the increasing complexity of current embedded systems, hardware design is being pushed to a higher level of abstraction, with High-Level Synthesis tools enabling hardware synthesis from untimed C++. Still, HLS technology does not provide a clear methodology to derive both hardware and software implementations from a single high-level code. This paper describes the design, implementation and evaluation of a resource scheduler that has a single C++ description and is automatically implementable in both software and hardware.**

***Keywords*-High-level synthesis; system-level design; resource scheduling; reconfigurable computing**

I. Introduction

Embedded systems are becoming increasingly complex as the advances of the semiconductor industry allow the use of sophisticated computational resources in a wider range of applications. Often, the development of such systems encompasses an integrated hardware/software design that can be realized by several computer architectures, ranging from 8-bit microcontrollers to complex *Multiprocessor Systems-on-Chip* (MPSoCs). In order to deal with this complexity, embedded system designs are being pushed to higher levels of abstraction, such as *system-level design*. In this scenario, a convergence between hardware and software modeling is desirable, since a unified approach would allow one to decide 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 descriptions at the behavioral level [1]. These tools allow designers to describe hardware components using languages like C++, and with untimed constructs. The focus of these tools, however, is hardware synthesis, and they do not provide a clear methodology for developing components implementable in hardware and software.

In this paper, we aim to advance in the direction of unified hardware/software design. Previous works [2] defined guidelines for translating operating system components – such as timers, schedulers, and synchronizers – from software to hardware and vice versa. These components were designed to allow their implementation to migrate between the hardware and software domains without the need to modify their client application. Nevertheless, these *hybrid components* are defined only in terms of their interface and communication structure, and the implementation of their behavior still follows different methodologies. To narrow this gap, we describe and evaluate a unified (single-code) resource scheduler. The choice of such component as a case study is motivated by its complex behavior. A scheduler may perform operations both synchronously (upon request by another component) or asynchronously (by preempting the execution of another component). Furthermore, a hardware-implemented scheduler has (in one possible microarchitecture) deterministic execution time, eliminating jitter and improving the support of real time applications.

Our design follows the principles defined by the *Application-driven Embedded System Design* (ADESD) [3] methodology. ADESD elaborates on OOP and *Aspect-Oriented Programming* concepts, defining a domain engineering strategy focused on the production of scenario-independent components. The C++ code of our scheduler leverages on *generic programming* [4] techniques such as *static metaprogramming* in order to provide an efficient implementation for both hardware and software.

II. Related work

Several design methodologies and tools were proposed in order to provide more tightly coupled hardware and software design flows [5]–[7]. Most of these methodologies are based on the concept of building a system by assembling pre-validated components. One example is *Metro-II* [7]. This framework follows the *Platform-based design* (PDB) [5] methodology, and proposes the use of a metamodel with formal semantics that developers can use to capture designs. However, these methodologies do not define clear guidelines to design new components which can be reused in a wide range of applications. Also, the hardware/software partitioning is defined in the early design phases and is limited by the platform specification.

In order to overcome these issues, one must focus on closing the gap between software and hardware design. State-of-the-art EDA tools like Catapult C [1] support hardware synthesis from high-level C++ constructs, and several works have already demonstrated the applicability of high-level synthesis for implementing computation-intense hardware components [8]. The OSSS+R methodology [9] advances further and aims at generating both hardware and software from the same description. It raises the level of abstraction of RTL SystemC by adding language constructs to support polymorphism and high-level communication. However, hardware/software partitioning must still be done early in the design process [10], and the inclusion of non-standard language constructs reduces compatibility with available compilers and synthesis tools. The *Saturn* [11] design flow also contributes in this direction, but follows a different approach. The authors propose *SysML*, an extension of UML for system-level design, and a tool which generates C++ for software and RTL SystemC for hardware. However, Saturn has the same limitations described previously: hardware and software cannot be generated from the same specification. Additionally, it is not clear whether their tool generates code only for the interface and integration of components, or the behavior is also implemented in SysML.

The ADESD [3] 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. Dependencies observed during domain engineering are captured as different aspects, thus enabling components to be reused on a variety of execution scenarios with the application of proper aspects. In [2] ADESD's design artifacts were explored to define guidelines for the development of components which could migrate between the hardware and software domains, and a common architecture for communication between components living in hardware and/or software. These *hybrid components* provide more flexibility and allow for the postponement of the hardware/software partitioning. However, hybrid components still require duplicated descriptions (in a software programming language and in a hardware description language). In the next sections we demonstrate how ADESD's design artifacts can be used to overcome such limitations through the unified (single code) design of a hybrid hardware/software resource scheduler.

III. Development and implementation of a hybrid HW/SW resource scheduler

As a case study we chose to modify the scheduler of the EPOS operating system [3], making its C++ code unified and suitable for automatic implementation in both hardware and software. EPOS's domain engineering simplified our work by providing a good *separation of concerns* around the scheduler. Figure 1 shows a simplified class diagram of the scheduling-related classes in EPOS.

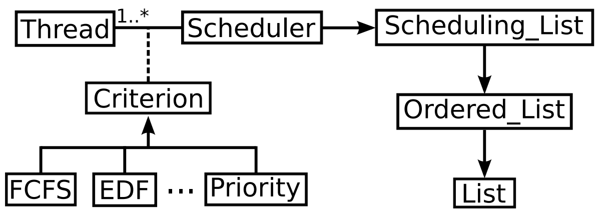

Figure 1. Simplified class diagram of scheduling-related classes in EPOS.

The Scheduler class is responsible only for keeping ordered queues, with the ordering defined by the scheduling criterion. An object of the Scheduler class also allows its callers to insert and remove clients from the queues, as well as to get the current and next owners of the resource. In our case study, we implemented a scheduler for threads. However, the Scheduler code is largely independent of the type of resource being scheduled (in fact, Scheduler is a *class template*). Furthermore, concerns such as timing interrupt generation and context switching are handled by other classes (Alarm and CPU, respectively).

Among all classes in figure 1, we adapted the code of the Scheduler class (also of its base classes), so that it can now serve as input for both hardware and software implementation flows. Figure 2 summarizes both flows.

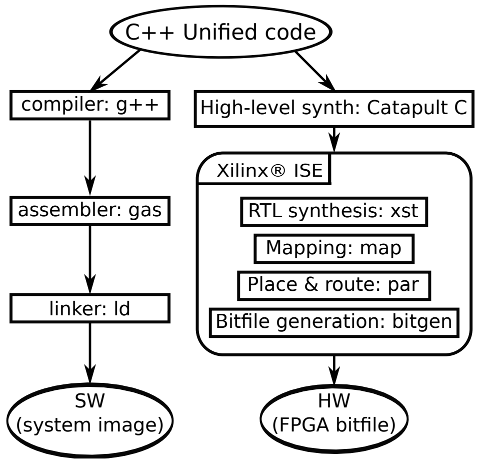

Figure 2. Implementation steps and tools used, for both HW and SW flows.

An important consideration about figure 2 is that the usage of a unified code for the scheduler did not affect the software implementation flow. The toolchain consists only of standard open-source tools, and is *exactly the same* as the one used with the software-only code. In the hardware implementation flow, the only notable addition is the *High-level synthesis* step, done by Catapult C of Mentor Graphics®. Catapult C takes untimed C++ as input, performs technology mapping, resource allocation and scheduling, producing as its result a RTL (VHDL or Verilog) description of the hardware block.

During our initial attempts at high-level synthesis we were faced with three fundamental limitations imposed by the toolchain, and these limitations have guided our development from the very beginning. They are, namely:

1) Pointers have no intrinsic meaning in hardware. They are mapped to indices of the storage structures to which they point. Thus, *no null or otherwise invalid pointers* are allowed in the source code.
2) There is no feature similar to dynamic memory allocation in hardware. Therefore, in a *synthesizable* code, all data structures must reside in statically allocated memory.
3) The top-level interface of the resulting hardware block (port directions and sizes) is inferred by Catapult C

from a *single function signature*. There must be only one function in the code with the pragma “hls_design top”.

Limitation number 1 influenced our development deeply, mainly because several methods of the Scheduler class used null pointers to report failures. To overcome this limitation, we changed the code of the scheduler (and of its internal data structures) to utilize an *option type*.

An option type is a container for a generic value, and has an internal state which represents the presence or absence of this value. Option types are very popular in functional programming as the return type of functions that can fail. We implemented an option type in the C++ class template *Maybe<T>*, which has the following constructors:

```
Maybe(): _exists(false), _thing(T()){}
Maybe(T obj): _exists(true), _thing(obj){}
```

One constructor represents the absence of a value in the container while the other represents its presence. By replacing all occurrences of simple pointers (T*) in the scheduler code with Maybe<T*>values, we completely avoided passing invalid pointers around. Still, the modified code was not significantly larger, neither did it run significantly slower, as we clarify in section IV.

The other two limitations of the hardware flow (no dynamic memory and a single function as interface) were overcome by *wrapping* the scheduler code with two C++ class templates: they are responsible, respectively, for storage allocation and method call dispatch. The functionality and architecture of these wrappers are summarized in figure 3.

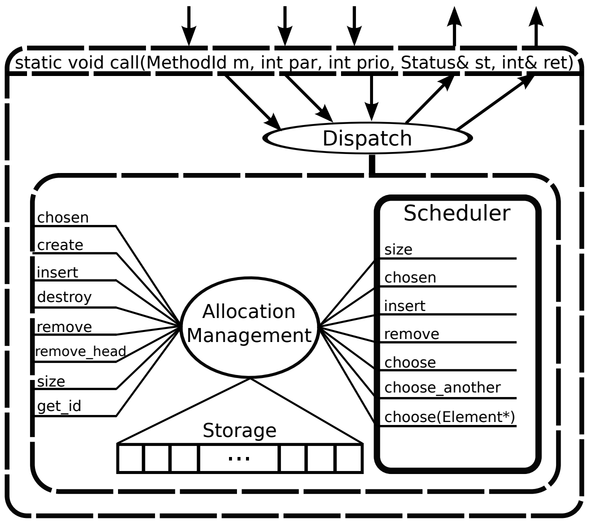

Figure 3. Wrappers that adapt the Scheduler class to make it synthesizable.

The innermost block in the “onion-like” architecture of figure 3 is the Scheduler C++ class template. This exact code serves as input to the software implementation flow. The two wrappers mentioned beforehand are represented by the two outermost blocks in the diagram, with dashed bounding boxes.

In figure 3, we can also see that the interface of the storage allocation wrapper is largely the same as the interface of the Scheduler class. The only responsibility of the storage allocator is, therefore, to provide storage space – reserving and releasing it on demand – to the wrapped Scheduler instance.

The interface of the dispatch wrapper is radically different, however, and has a single function with all input and output parameters used by the wrapped class. One notable parameter of type *MethodId* is present in the call function of the dispatch wrapper. The responsibility of the dispatch wrapper is to interpret the value of this parameter, perform the necessary type conversions and call the appropriate method of the wrapped class. The value returned by the called method is also inspected, converted if necessary and assigned to one of the wrapper’s output parameters.

Both wrappers developed are highly generic and were purposefully designed to be able to wrap other components. In fact, they are C++ class templates, parametrized by the wrapped class and the size of the pre-allocated storage space.

IV. Results

Our hybrid implementation of a resource scheduler was evaluated in both software and hardware-based scenarios. The high-level synthesis of the scheduler was performed using Catapult-C®, from Mentor Graphics, and for the RTL synthesis and translation we used the ISE toolchain, from Xilinx®. The following three hardware microarchitectures were investigated:

1) *Fully automatic* microarchitecture: The complete synthesis process, from untimed C++ to FPGA bitfile, happened with no human interaction. All directives and flags were left at their default values.
2) *Fully serial* microarchitecture: Catapult C was instructed to map data structures in the C++ code to memories (BRAMs). Our goal in this case was to save device area.
3) *Fully parallel* microarchitecture: Data structures were mapped to registers and all loops were unrolled, in order to fully harness the available parallelism in the algorithms.

We evaluated the performance of all three automatically-generated microarchitectures by comparing them (both in terms of area and critical path delay) with a handwritten RTL scheduler implementation [12]. All implementations were synthesized targeting a Virtex6 (XC6VLX240T) device from Xilinx®, and with a target clock frequency of 50MHz. Table I summarizes the performance of all evaluated hardware microarchitectures, both generated and handwritten.

Particularly interesting is the fully serial generated microarchitecture, which gets close to the handwritten RTL, both in terms of area (+32%) and of maximum delay (+19%). Furthermore, despite having greater area and maximum delay, the fully parallel microarchitecture also has its advantage: all operations (search, insert, remove, etc.) are done in parallel, and therefore within a constant number n of cycles, which is the same for all operations. This eliminates jitter in scheduling, improving the support of hard real-time applications.

Scenario	LUTs	Device occupation	Max. delay
Handwritten RTL	1250	0.83%	5.598 ns
Fully serial	1654	1.10%	6.672 ns
Fully automatic	3392	2.25%	7.341 ns
Fully parallel	5121	3.40%	10.597 ns

Table I
PERFORMANCE SUMMARY OF ALL EVALUATED HW MICROARCHITECTURES

As already mentioned in section III, we changed the original C++ source code of the scheduler (by introducing the option type) in order to make it synthesizable. The introduction of the option type happened in the unified code, affecting, therefore, also the software implementation.

To measure the effect of these changes, we compared the old (software-only) C++ scheduler code with the unified one, in terms of code size and execution time. We chose MIPS32 as the target architecture for compilation, and measured the resulting EPOS system image size and the execution time of some methods for both scheduler implementations. Table II summarizes these results.

Metric	SW-only	Unified	difference
Code size	29580 B	29700 B	0.41%
insert execution time	$51 \times 10^{-1} \mu s$	$53 \times 10^{-1} \mu s$	3.92%
remove execution time	$27 \times 10^{-1} \mu s$	$29 \times 10^{-1} \mu s$	7.41%
suspend execution time	$32 \times 10^{-1} \mu s$	$34 \times 10^{-1} \mu s$	6.25%
resume execution time	$42 \times 10^{-1} \mu s$	$43 \times 10^{-1} \mu s$	2.38%
choose execution time	$61 \times 10^{-1} \mu s$	$67 \times 10^{-1} \mu s$	9.84%

Table II
COMPARISON OF GENERATED CODE SIZE AND EXECUTION TIME BETWEEN SW-ONLY AND UNIFIED SCHEDULER VERSIONS

The execution time measurements were done by instrumenting Scheduler methods with calls to a *timestamp counter* that runs at the same frequency as the CPU. The resolution, therefore, is 20 ns, and our measurements are in tenths of microseconds. A dining philosophers application was used to request the scheduler's services. We took 100 sample runs of this application, and table II shows the average execution times of the methods.

V. CONCLUSION

In this paper we presented the unified design and implementation of a generic resource scheduler. The exact same C++ source code serves as input for both the hardware and software implementation flows. Furthermore, the performance of this unified code in software and hardware execution scenarios is close to the performance of software-only and hardware-only implementations, respectively. No additional tools were necessary to generate a software system image from the unified code, and the adaptions necessary to hardware synthesis were organized in a layered architecture, facilitating automatic application.

Careful domain engineering and system design leads to algorithm implementations that are largely separated from execution scenario details. Our case study shows that code following these guidelines (specially the ADESD methodology, as presented in section II) is suitable for automatic implementation in both hardware and software scenarios. During the development of the hardware scenario adapters, we came across the hypothesis that these adapters could be modeled as aspects [4]. The further investigation of this hypothesis constitutes important future work related to the research exposed in this paper.

As a last remark we emphasize the fact that programming algorithms using unified source code, as we proposed, facilitates design space exploration through two mechanisms:

- Design choices regarding execution scenario (whether hardware or software) can be *postponed*, with two advantages: duplicated testbenches are avoided, and the fine-tuning of parameters can be done in the unified code.
- The automatic derivation of hardware from the unified code is guided by *synthesis directives*, which can be optimized by design space exploration frameworks.

REFERENCES

[1] Mentor Graphics, "CatapultC Synthesis," http://www.mentor.com/esl/catapult.

[2] H. Marcondes and A. A. Fröhlich, "A Hybrid Hardware and Software Component Architecture for Embedded System Design," in *IESS '09*, Langenargen, Germany, 2009, pp. 259–270.

[3] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

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

[5] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design & Test of Computers*, vol. 18, no. 6, pp. 23–33, 2001.

[6] W. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, L. Gauthier, and M. Diaz-Nava, "Multiprocessor SoC platforms: a component-based design approach," *Design & Test of Computers, IEEE*, vol. 19, no. 6, pp. 52–63, 2002.

[7] 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 *DVCon 2007*, February 2007.

[8] M. Fingeroff, *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[9] E. Grimpe and F. Oppenheimer, "Extending the SystemC synthesis subset by object-oriented features," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 25–30.

[10] 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 *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 128–133. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403409

[11] W. Mueller, D. He, F. Mischkalla, A. Wegele, P. Whiston, P. Penil, E. Villar, N. Mitas, D. Kritharidis, F. Azcarate, and M. Carballeda, "The SATURN Approach to SysML-Based HW/SW Codesign," in *Proc. of the 2010 IEEE Annual Symposium on VLSI*, ser. ISVLSI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 506–511.

[12] 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*, ser. CSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 382–387.