A Unified HW/SW Interface Model to Remove Discontinuities between HW and SW Design

Aimen Bouchhima, Xi Chen, Frédéric Pétrot, Wander O. Cesário, Ahmed A. Jerraya
TIMA Laboratory
46 Avenue Félix Viallet
38031 Grenoble CEDEX, France
+33 476 57 43 01
Aimen.Bouchhima@imag.fr

ABSTRACT
One major challenge in System-on-Chip (SoC) design is the definition and design of interfaces between hardware and software. Traditional ASIC designer and software designer model HW/SW interface twice. Using two separate models introduces a discontinuity between hardware and software. This paper introduces a unified HW/SW component model to describe different parts of HW/SW interface at different abstraction levels. The benefits of using the proposed model are two fold: first, it provides a single model to present system design from abstract specification to mixed HW/SW implementation and second, it enables full system simulation at different abstraction level during refinement flow.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features

B.4.2 [**Input/Output and Data Communications**]: Input/Output Devices

General Terms
Algorithms, Design, Standardization, Languages

Keywords
Hardware/Software Interfaces, Hardware dependent Software, Embedded Systems

1. INTRODUCTION
90% of new ASICs already include a CPU in 130nm technology. Multimedia platforms (e.g. Nomadik and Nexperia) are already multi-processor system on chip (SoC) using different kinds of programmable processors (e.g. DSPs and microcontrollers) [1]. Heterogenfeous cores are exploited to meet the tight performance and cost constraints. This trend of building heterogeneous multi-processor SoC will be even accelerated. SoCs will be composed of multiple, possibly highly parallel processors for applications such as mobile terminals, set top boxes, game processors, video processors, and network processors.

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

A major challenge to effectively design such systems is to master their inherent complexity within an ever shrinking time-to market window while meeting stringent resource constraints (cost, power, area etc).

Face to this challenge, classic SoC design flows seem to reach their limits. Such flows rely on a sequential hardware/software design approach where complete hardware architecture should first be developed before software could be programmed on top of it. This implies several limitations, which could be summarized in the three following points:

(1) An inherently long design cycle especially if redesign loops have to be performed before reaching acceptable design.

(2) Because software is developed --at a low abstraction level-- in a hardware dependent way, sharing such software across several designs is considerably limited.

(3) Since hardware/software integration is performed late in the design flow, the exploration of architectural trade-offs turns to be a very tedious and time consuming process. Actually this late integration denotes a gap in the design of such hardware/software systems. This is mainly due to the absence of a unified model that continuously capture hardware and software at different abstraction levels during the design flow.

Based on these observations, an "ideal" design flow that targets multiprocessor SoC should allow:

(1) Concurrent HW/SW design to shorten the design cycle.

(2) Software component reuse to master software complexity and reduce development cost and effort.

(3) HW/SW integration at multiple abstraction levels to allow effective exploration of HW/SW architecture trade-offs.

The solution for the two first requirements clearly goes through relaxing the tight dependency between hardware and software design. Such practice is already a rule of thumb in general purpose computer design where hardware and software seem as to belong to completely different worlds. In fact, in order to design their applications, software programmers usually rely on thick abstraction layers that make hardware appear as a perfect machine. On the other hand, hardware may be independently developed as long as it complies with a set of fixed, software-defined conventions.

This software-centric design approach, completely decouple hardware and software development in favour of productivity and

extensive component reuse. These are clearly not the unique issues in embedded context, where satisfying the resource constraints and meeting application performance are equally important. Applying the computer design approach to the SoC domain -without considering SoC specificities - will result in lower-quality, over sized (expensive) and non-competitive designs.

Recently, the platform based design (PBD) paradigm emerged as the solution that basically allows to adapt the general purpose computer design approach to the SoC context [2] [3]. This is achieved mainly through the concept of application domains or classes. An application domain is a set of applications that share similar characteristics. For each application domain, corresponds a system platform that may be considered as a generic architecture family or template. A physical architecture is then viewed as a particular instance of this platform, targeting a specific application inside the application domain.

The joint generalization/differentiation concept (platform/ instance) allows a trade-off between design reuse and productivity on one hand, and efficiency on the other hand. Reuse is achieved via the common features shared between applications belonging to the same domain, while efficiency is ensured by customizing the architecture instance to the particular application needs.

Although they succeed to meet the two first requirements (design concurrency and component reuse), to the best of our knowledge, conventional platform based design flows still have not proposed any solution to the discontinuity problem between hardware and software design. We believe that bridging such design gap is a key issue in order to make full benefit of concurrent hardware software design flow and to further reduce design costs and efforts.

Bridging the design gap means considering hardware/software design at several abstraction levels, starting from abstract hardware/software specification and arriving to detailed low level implementation. The validation of the entire system at each abstraction level is a key enabler for both efficient architecture exploration and early error/bug detection/correction. For instance, evaluating the effect of one particular RTOS scheduling policy while taking into account the on-chip network routing algorithm is an example of typical design decision that should be performed as early as possible. Similarly, choosing to delegate a given operating system functionality to a dedicated hardware component is better done at early design stage, i.e. before a detailed implementation of either architectures is necessary. The late evaluation/validation of such architectural decisions – that is, once a particular architecture instance is completely developed - is a tedious and time consuming practice and may results into complete redesign cycles and/or lower-quality designs.

In this paper, a unified HW/SW interface component model is advocated to remove the discontinuity between hardware and software sides. The proposed model allows capturing the hardware software interface at different abstractions level during the whole design flow and provides an executable environment to perform global design space exploration. The component based concept is used to promote design reuse at each abstraction level within the context of a concurrent HW/SW design flow. We show how this could effectively address the reuse of embedded software components in a platform based context without incurring the excessive overhead of thick software abstraction layers.

The rest of the paper is organized as follows: section 2 discusses some in-depth issues related to the concept of hardware dependent software in SoC design. The proposed design flow is introduced in section 3. Section 4, describes the proposed unified HW/SW model and shows an application of this concept to model a complete HW/SW system.

2. HARDWARE-DEPENDENT SOFTWARE IN SOC DESIGN

The increasing complexity of multiprocessor SoC has put many constraints on the way embedded software is being developed. In fact, unlike general purpose platforms with regular and homogeneous architectures, multiprocessor SoCs rather exhibit heterogeneous and irregular architectures [1]. As a matter of fact, programming such devices generally turns out to be a low level programming, where a deep knowledge of the underlying hardware architecture, in its smallest details, is required in order to achieve the desired performance. Of course, from a software perspective, such a strong dependency on the the underlying hardware architecture has many disadvantages. First, it implies a long and sequential design cycle as software designers are forced to wait until complete hardware architecture is made available. This gets even worse if modifications to this initial architecture are to be made, generally leading to a redesign cycle of a major part of the developed software. Second, this makes the validation and debug of the developed software a tedious and error-prone process due to its intricate dependency upon subtle hardware features. Last, and not least, the reuse of software components is considerably limited as different software IPs must be adapted to different target architectures.

At the heart of the problem, lays this low level programming abstraction that serves as basis for embedded software development. What we mean by low level programming abstraction is not necessarily the use of assembly languages (although this still represents a significant part in current embedded software designs). It is rather the way application programmers are exposed to the bare hardware when designing their systems. This may include, for instance, some low level C code that manipulates few bits in a particular register of a memory mapped I/O device.

The Hardware dependent software (or HdS) concept is introduced to exactly tackle the disadvantages of such low level programming practice. The exact meaning of HdS depends on the context where it is used.

In general purpose computer domain, HdS is already a well known concept. Examples include windows NT Hardware Abstraction Layer (HAL), Linux Universal Device interface (UDI), Simple DirectMedia Layer (SDL) etc. This generally represents a thick software layer that completely hides the underlying hardware through a fixed standard application programmer interface (API). Furthermore, since it generally already implements many design decisions (policies), such abstraction layer used to be tightly coupled to the operating system.

In platform based design, each application domain may have it own HdS API, reflecting the specificities of each domain (platform). The HdS layer includes those “low level” software functionalities whose implementations depend directly upon the underlying hardware architecture instance. This may include for

instance device drivers, boot code, DSP-specific algorithms, and possibly parts of the operating system (interrupt management, context related operations etc). Inside the same domain, software designers could rely on the fixed HdS API to develop their applications, possibly reusing pre-designed elements at the operating system or/and higher levels. This basically structures embedded software into two main layers (Figure 1.a): one is hardware dependent (HdS), the other is hardware independent (HiS). Such fixed partitioning of software within platform based design is likely to be a major source of inefficiency and may therefore seriously limit the effectiveness of the approach, especially in most demanding applications.

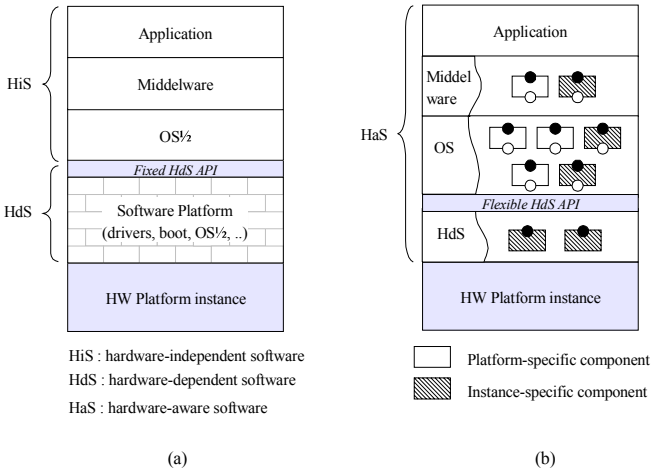


Figure 1. HdS concept (a) in classic platform based design (b) in the proposed approach

To tackle the inefficiency problem while preserving software reuse, the HdS API has to be made flexible, i.e. customizable through specific services that are dependent upon the underlying *hardware platform instance.* In Figure 1b, this is allowed by using a component model across the different software layers up to the application. In the figure, dashed elements correspond to components that are tightly dependent on the specific architecture instance. Note that such components may be used not only inside the HdS layer but also in upper software layers. We talk about hardware aware software (HaS) instead of hardware independent software. The mechanism used to ensure the coherency of the obtained software design is discussed in subsequent sections.

3. HW/SW INTERFACE ABSTRACTION

3.1 HW/SW interface concept

In order to allow for concurrent HW/SW design, we need abstract models of both software and hardware components. Ideally one would like to start with a set of SW tasks communicating with a set of HW subsystems (figure 2a). This could be viewed as an abstract HW/SW specification of the application. Because software components run on processors, the abstraction needed to describe the interconnection between software and hardware components is totally different from the existing abstraction of wires between hardware components as well as the function call abstraction used to describe hierarchy in software. We simply refer to it as abstract HW/SW interface (figure 2b). The HW/SW interface needs to handle two different interfaces: one on the software side using API and one on the hardware side using wires. This heterogeneity makes HW/SW interface design very difficult and time-consuming because the design requires the knowledge of both software and hardware and their interaction. Once refined, the abstract HW/SW interface eventually results in a set of heterogenous subsystems including CPU subsystem, HdS and HW adaptation (figure 2c).

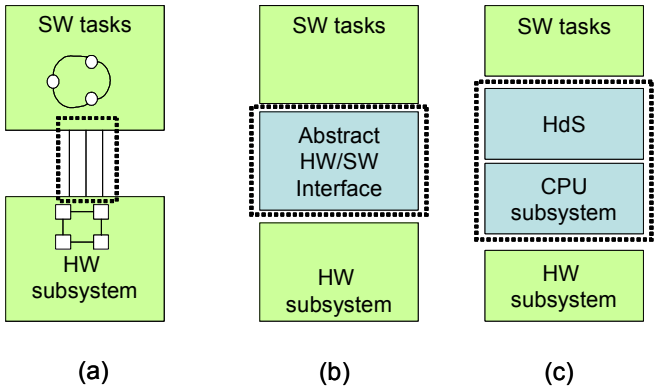


Figure 2. Evolution of abstraction levels in chip design (a) Implicit HW/SW interfaces, (b) Explicit abstract HW/SW interfaces, (c) HW/SW interfaces implementation

3.2 Removing discontinuities is SoC design

As stated previously, the discontinuity observed in classic SoC design flows is due to the separate models for hardware and software components. This discontinuity prevents from performing global design space exploration and validation since early design stages and leads to low quality designs and unacceptable time-to-market delays.

Using a unified HW/SW model during different steps of design flow removes this discontinuity and enables seamless design process for both hardware and software from functional specification to RTL implementation. Additionally, being executable, the unified model allows for multi-level validation and exploration of the entire design.

Figure 3 illustrates the proposed design methodology using the unified hardware software approach. This flow starts with a system specification using some functional model, e.g. Task Graph (TG). The functional model describes the algorithm for the system behavior. Instead of designing the SoC directly from the functional model, a two-step method is widely accepted [4]. The first step includes high-level architecture design and task level HW/SW partitioning, and the result is an abstract architecture model of SoC. In this model, there are three basic elements: modules (software and hardware), global communication interconnects and abstract interfaces. It is a high-level SoC model, in which the details of HW/SW interface are abstracted. The second step is to implement each element described in this abstract model. The benefits of the two-step SoC design method include enabling system architecture exploration and dividing the whole SoC design into several simpler independent design steps.

Hardware and software modules are separated by abstract interfaces. These need to be refined through embedded software design and hardware design. Some of the modules can use pre-designed implementations according to the task level partitioning.

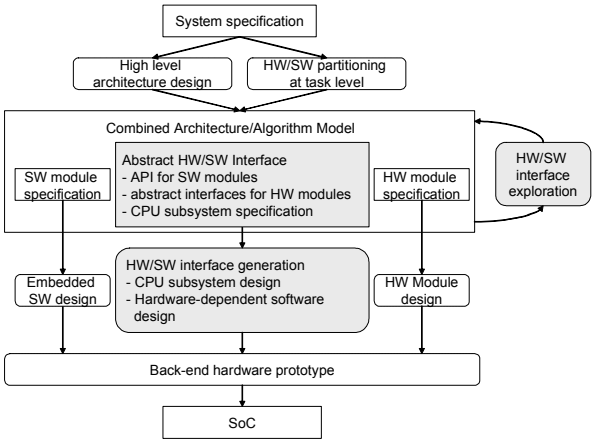


Figure 3. Proposed HW/SW interface design flow

With the support of a component library, HW/SW interfaces can be designed using a systematic method and a HW/SW interface generation tool (see Figure 4). The proposed method generates the HW/SW interface by selecting, configuring and integrating components in the component library. Hardware-dependent software and CPU subsystem are thus generated automatically.

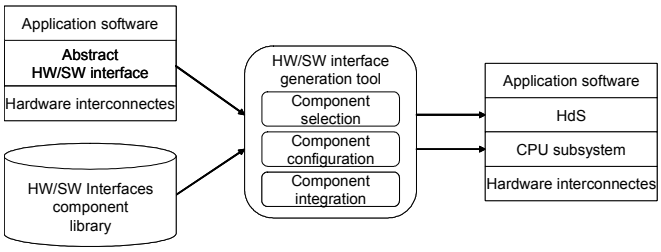


Figure 4. Needed HW/SW interface generation tool.

The key issue for the success of such a model is the definition of a unified model able to represent both the HdS and the CPU subsystem. A combined HdS-CPU refinement method is presented in [9].

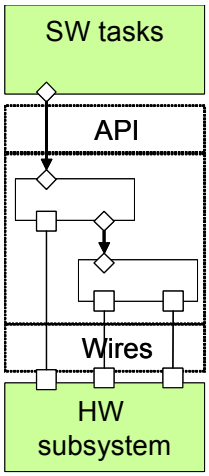
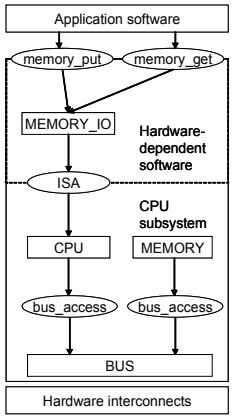


Figure 5. Unified HW/SW Model using the service-based model

4. MODELING HW/SW INTERFACES USING SERVICE BASED COMPONENT MODEL

In a service based component model, the basic concepts are components, services and service dependency. Figure 5 shows an example of service based component model. In this figure, rectangles, circles and arrows correspond to components, services and service dependency respectively. Extensive description of using this model for HW/SW interfaces is given in [5] [6]. This section will only give an example to show how HW, CPU and software can be described using the same model.

The design of CPU subsystem and hardware-dependent software is tightly coupled. We present a very simple HW/SW interface example shown in Figure 6 to illustrate the link between HdS and the CPU subsystem. As shown in Figure 6(a), the HW/SW interface is modeled using the unified service-based component model [5], in which there are three hardware components, i.e. *CPU*, *MEMORY*, *BUS*, and one software component, i.e. *MEMORY_IO*. The *BUS* is the communication component whose service ports connect only with hardware components. The *MEMORY* represents a special kind of peripheral that provides service ports for data store/retrieve. The *CPU* provides its instruction-set as service port to connect with software components. The *MEMORY_IO* HdS component provides two software services, i.e. *memory_put* and *memory_get*. Figure 6(b) shows a part of the XML-based description for this component. Its implementation requires the instructions provided by the CPU. Unlike the load/store CPU instructions, which get/put a fixed data type in the memory, the service provided by *MEMORY_IO* can use a configurable data type when accessing the data in memory. This is done by specifying the date type in the *MEMORY_IO_IMPLEMENTATION* definition.



(a) HW/SW interface

```
<itemdef name="MEMORY_IO">
  <typeref base="COMPONENT"/>
  <value>
    <value name="declararion" access="ref"
           data="MEMORY_IO_DECLARATION"/>
    <value name="implementation" access="ref"
           data="MEMORY_IO_IMPLEMENTATION"/>
  </value>
</itemdef>
<itemdef name=" MEMORY_IO_DECLARATION ">
  <typeref base="DECLARATION"/>
  <value>
    <value name="description" access="value" data=""/>
    <value name="required">
      <value access="value" data="ISA"/>
    </value>
    <value name="provided">
      <value access="value" data="memory_put"/>
      <value access="value" data="memory_get"/>
    </value>
  </value>
</itemdef>
<itemdef name=" MEMORY_IO_IMPLEMENTATION ">
…
```

(b) XML-based description of MEMORY_IO

Figure 6. Example of hardware-dependent software and a simple CPU subsystem

5. CONCLUSION

In conventional HW/SW codesign approaches [7] [8], designers start from a system specification that captures the functionality of their design in a formal way. This formal specification is then (automatically) refined to a final architecture, generally composed of hardware and software elements. Although they have the advantage of being fully automated and guaranteeing the correctness of the generated architecture with respect to the initial specification, such approaches suffer from limitations that restrict their effective use. These limitations are mainly related to the restrictive assumptions that have to be satisfied by both applications and target architectures, which make these approaches hardly scalable to complex, real-life applications.

Having a unified model to describe both hardware and software components at different stages of a design flow is, in our view, a key enabler toward an effective reuse of both hardware and software components. Moreover, we believe that this mixed level component integration, put in a concurrent HW/SW design flow context, is able to achieve higher quality designs while considerably shortening the total design cycle.

A key feature of the proposed methodology is the ability to perform global validation and design space exploration of the entire HW/SW design at different abstraction levels [5].

Another key feature is the ability of automatically selecting, configuring and composing different components (from different libraries) in order to build up the complete system [9].

6. REFERENCES

[1] A. A. Jerraya, W. Wolf, « Multiprocessor Systems-on-Chips », Morgan Kaufmann Publishers, ISBN 0-12-385251-X, September 2004.

[2] A. Sangiovanni Vincentelli, «Platform-based Design», EEDesign of EETimes, February 2002.

[3] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based de-sign. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[4] Keutzer, K., Malik, S., Newton, A. R., System Level Design: Orthogonolization of Concerns and Platform-Based Design, *IEEE Trans. Computer-Aided Design of Integrated Circuit and Systems*, vol.19, no. 12, Dec.2000.

[5] A. Sarmento, L. Kriaa, A. Grasset, M.-W. Youssef, A. Bouchhima, F. Rousseau, W. Cesario, A.A. Jerraya, "Service Dependency Graph, an Efficient Model for Hardware/ Software Interfaces Modeling and Generation for SoC Design", ISSS 2005, New York, USA, 19-21 September 2005.

[6] M.Zitterbart, "A Model for Flexible High performance Communication Subsystems", IEEE Journal on selected areas in communication, VOL. 11, NO, 4, MAY 1993.

[7] G. De Micheli, R.K. Gupta, Hardware-Software Codesign , Proceedings of the IEEE, V85, No3, 1997.

[8] W. Wolf, A Decade of Hardware/Software Codesign , IEEE Computer, 2003.

[9] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, A.A. Jerraya, "Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration", ASP-DAC 2005 proceedings, 18-21 January 2005, Shanghai, China, 2005.