Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration

Aimen Bouchhima Iuliana Bacivarov Wassim Youssef Marius Bonaciu Ahmed A. Jerraya

System Level Synthesis Group, TIMA laboratory
46, Av. Felix Viallet, 38031 Grenoble, France
{Forename.Name@imag.fr}

Abstract - Current and future SoC will contain an increasing number of heterogeneous multiprocessor subsystems combined with a complex communication architecture to meet flexibility, performance and cost constraints. The early validation of such complex MP-SoC architectures is a key enabler to manage this complexity and thus to enhance design productivity.

In this paper, we describe an abstract, high level CPU subsystem model that captures the specificities of such MP-SoC architectures, along with a timed co-simulation environment to perform early exploration of the entire HW/SW design. The model is based on the Hardware Abstraction Layer (HAL) concept allowing the validation of complex applications written on top of real-life operating systems. Experimentation with a MPEG4 application proves the interest of the proposed methodology.

I. Introduction

Heterogeneous multiprocessor systems on-chip may be represented, without loss of generality, as a set of processing nodes or components which interact via a communication network (fig. 1). Depending on the nature of the components and the way interaction between them is designed, different classes of SoC architectures may be obtained.

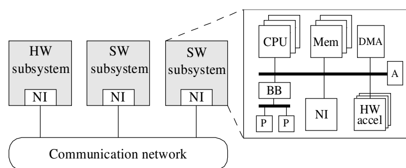

Figure 1 : a generic heterogeneous MP-SoC architecture

Realizing the complete potential of such SoC designs depends heavily on the ability to perform early validation of the entire design to explore different system-level trade-offs.

However, while a large body of research have focused on designing environments for early HW/SW co-validation, it turned out that the validation of the SW subsystems in the context of the overall design is a major bottleneck against efficient design space exploration.

The main problem behind this difficulty is the abstraction at witch a SW subsystem is considered. In fact, in most current approaches, SW is viewed at the CPU instruction level, which assumes complete knowledge of the CPU subsystem down to lowest details such as local peripherals address maps and interrupt registers bits assignment. This implies that the CPU subsystem architecture is fully designed at least at the RT level and that low level SW is also available to drive it. The validation of such subsystem relies on the classical approach including instruction set simulator(s) (ISS) of the target processor(s) and hardware models of peripherals (fig. 2-a).

In this paper we focus on a higher SW abstraction level : the Hardware Abstraction Layer (HAL) concept [1], where the entire CPU subsystem is viewed as an homogeneous entity providing a set of services to system programmers (HAL API) .

The major contribution of this paper is to provide an abstract CPU subsystem model based on the HAL concept. The proposed model supports irregular, custom architectures that feature (1) massive, sophisticated data transfer, (2) efficient synchronization schemes and (3) complex computation.

A simulation model based on the SystemC environment is also described allowing early, fast and time accurate cosimulation of the global design. Compared to a conventional ISS based simulation model (fig. 2-a), the proposed abstract CPU subsystem model provides higher interface levels to both HW and SW sides (fig. 2-b).

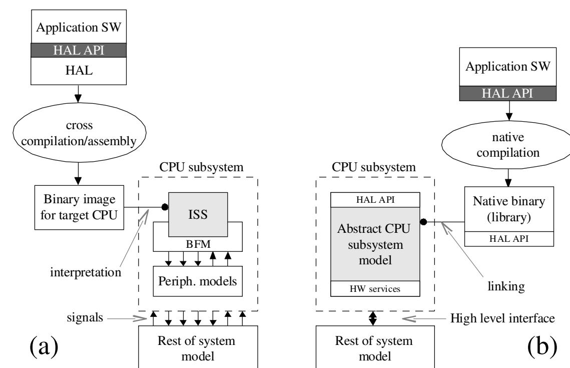

Figure 2 : (a) conventional ISS based simulation model (b) the proposed HAL based simulation model

The rest of the paper is organized as follows : after discussing further related work in section 2, the CPU subsystem abstract model is introduced in section 3, while section 4 details the underlying simulation environment. A case study on design space exploration of an MPEG4 real-time encoder is presented in section 5 while section 6 concludes the paper.

II. Related Work

Several works have addressed the modeling of processor architectures. Most of them used a language based approach to capture processor internal data-path and associated instruction set architecture (ISA). Such languages are referred to as architecture description languages (ADL). Examples include nML[2], LISA[3] etc. While having played an important role in the field of retargetable SW tool generation (compiler, assembler, simulators etc) and HW synthesis, such modeling languages are not intended to capture the whole CPU subsystem architecture. Besides, they operate at the ISA level and thus come late in the design flow.

Recently, some research activities have focused on providing high level codesign environments for early HW/SW co-exploration.

SoCOS [4] is an example of such environment including abstract OS model that can be simulated with timed HW models. In [5], a method of building OS simulation model is also presented. All, these approaches have focused on providing an abstract OS execution environment that captures the dynamic behavior of software. However they don't allow to model the underlying execution machine, that is the CPU subsystem architecture.

III. Abstract CPU subsystem Model

A. Overview

Current approaches to model a CPU subsystem view it as a set of HW components communicating via physical wires (buses, control signals, interrupt lines etc.). This “HW view” of the CPU subsystem is a natural one as long as software is considered at the ISA level. However, it becomes inconvenient if we would like to raise software abstraction up to the HAL level. At this level, the entire CPU subsystem should be considered as a functional entity that offers a set of services corresponding to application needs.
The proposed HAL level CPU subsystem abstraction is depicted in fig. 3. It is important to note that the various elements of the figure do not correspond to physical hardware components but rather to software functionalities as seen by system programmers. Similarly, the arrows in the figures are not wires but rather logical relationship between the different functional elements.
Seen globally, the model may be compared to a classic Von Neumann machine with an execution unit and a data or storage unit. However, unlike this basic computational model (which is used by the compiler abstraction), our model includes other system related aspects such as synchronization (which deals with interrupts) and I/O transfer (modeled within the data unit).

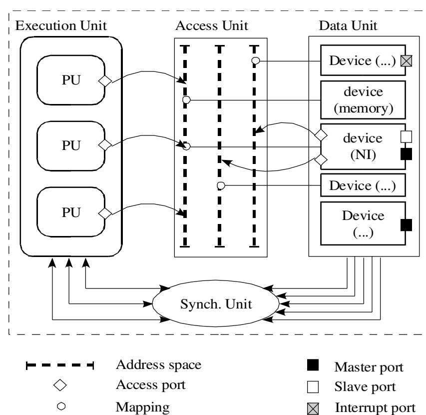

Figure 3 : HAL level abstraction of the CPU subsystem

Furthermore, the model supports parallel computations inside the execution unit. The access unit plays an important role in modeling the coordination of data transfer between entities requesting data access and entities providing such data.
To allow the CPU subsystem model to be used in a codesign environment, the HW interface to the rest of the system is also included. In fact, this HW interface represents the “trace” of the CPU subsystem model inside the overall (hardware) system. The interface itself is abstracted through two types of services: provided services (slave ports) and required services (master port). Special provided services correspond to interrupt requests.

B. Execution unit abstraction

The execution unit (EU) abstracts computation inside the CPU subsystem. It is basically composed of one or more processing units (PU). A processing unit is an independent parallel computation path that has its own execution thread. This decomposition corresponds to a task level parallelism and is explicitly visible to programmers. Examples may range from SMT and SMP architectures to more heterogeneous forms, all known as chip multiprocessing (CMP).
The EU element should provide a boot abstraction that ensures a consistent state at startup time (especially in a multiprocessor context). It is also responsible of providing an identification service as well as services for synchronization (e.g. by means of spin-locks) and inter-PU interruption. Atomic operations which are used to implement spin-locks are provided by the access unit.
The PU element should provide services to manipulate its underlying execution thread (context operations). These services are used by the operating system to implement SW multitasking.

C. Data unit abstraction

The data unit is a placeholder for more basic data entities which are device elements. A device element is an abstraction of any physical device that may hold relevant information from a user perspective. This excludes many other physical devices that don't directly fill a functional role from a high level programmer point of view (e.g. Bus bridges, interrupt controllers etc).
A device element may be of two different types : passive and active. A passive device corresponds to a collection of simple idempotent memory locations (e.g. RAMs, ROMs etc). An active device is composed of memory locations plus an underlying behavior that may modify the content of these locations.
The services provided by the data unit should be considered as per device basis. Generally passive devices just provide read/write services from/to specific addresses. However active devices may provide higher level services that perform more complex functionalities such as resetting a timer, initiating a DMA transfer etc. From an implementation point of view, this eventually corresponds to a sequence of read/write operations on registers. In our model, we assume that whenever a higher level functionality is available for a device, the programmer is supposed to use it rather than performing low level register accesses. This is coherent with the “HAL concept” which relieves the programmer from low level implementation details.

D. Access unit abstraction

The access unit corresponds to the abstraction of a key feature in SoC design: communication and data transfer. It is made of two different entities: address space collection and access ports.
The address space collection is a set of independent address spaces. An address space abstracts a physical addressable domain which may be a bus or a hierarchy of related bus segments.
Each address space entity is associated to a set of device elements. This association constitutes the address map of the considered address space. A device element may be mapped to more than one address space. In such case, the device may be accessed independently from either space (case of a dual port memory). In our model, a device may also have a “virtual map” where actual values of the address range are nor relevant for the considered abstraction level.
An access port is the only place where address space entities may be accessed. Access ports are often associated to PU, but some device elements may also have associated access ports (case of DMA enabled devices for instance).
In terms of services, an address space element should provide basic read/write operations. Other services such as conflict resolution by means of sequential operations (like barriers) as well as atomic operations may also be needed.
In many cases, access ports are transparent to programmers. However, they may hide complex behavior such as caching and dynamic address translation (MMU). In such cases, the associated services must be provided (cache flushing / invalidating, translation table operations etc).

E. Synchronization unit abstraction

In our model, we consider that the interruption process has a global visibility since it concerns different types of elements. Physically, the interrupt management process is often distributed among

several hardware components starting from the processor itself and including special devices such as interrupt controllers. In our model, The whole (complex) interrupt management process is abstracted by the synchronization unit. Interrupt requests are identified by unique IDs and have associated priorities that determine their run-time precedence. Interrupt requests are then processed and delivered on a per PU basis.

The synchronization unit should provide services to attach interrupts to special SW handlers (ISR). It should also provide appropriate services to control the run-time behavior of the interruption mechanism such as enabling/disabling interrupts, and setting/modifying their priorities.

F. The metamodel

The different elements of the above abstract CPU subsystem model and their relationship can be captured, more formally, within a domain specific language (DSL) using the metamodel formalism. The figure below depicts the proposed abstract syntax of the CPU subsystem model

The metamodel description is based on a UML-like notation where different elements of the abstract CPU subsystem model are captured within a class diagram. Each HAL service related to an element corresponds to a member function inside the associated class.

In the figure, the gray area corresponds to what we call “core HAL”. This includes the basic elements that are common to all architectures. Domain specific extensions may then be defined by extending those basic elements.

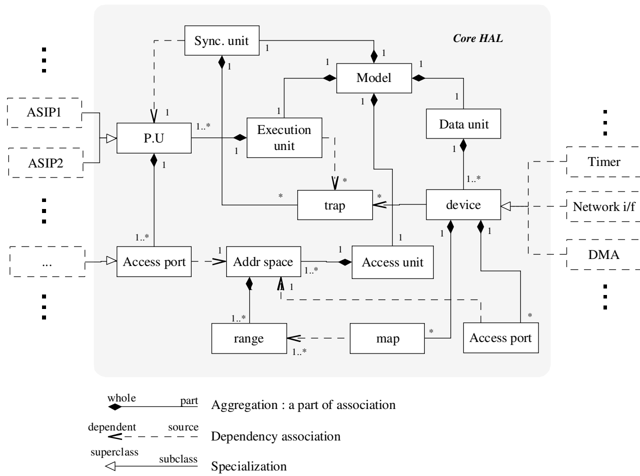

Figure 4: the metamodel view of the abstract CPU subsystem

IV.The simulation environment

The simulation model is used inside a cosimulation environment to perform early design space exploration. As cosimulation platform, we use SystemC that offers a natural context for describing software in addition to many other attractive system level design facilities[6]. To achieve the required simulation speed, we rely on the native execution of SW [7] in contrast to the interpreted, ISS based approach. The native execution approach is made possible since our SW is HW independent. It, therefore, can be compiled for the host simulation machine and run against an appropriate HAL library that emulates the actual target dependent HAL. This approach, augmented by appropriate performance annotations of the original SW code ensures an equivalent functional behavior while giving an acceptable accuracy level.

Fig. 5 gives an overview of the proposed simulation environment. One one side, we have software code written at the HAL level (i.e using HAL API). On the other side, we have the SystemC (hardware) model of the entire design including communication network, HW subsystems and CPU subsystems. For each CPU subsystem module, corresponds an application SW. In our case, different applications are compiled separately as shared library objects and linked to the SystemC executable.

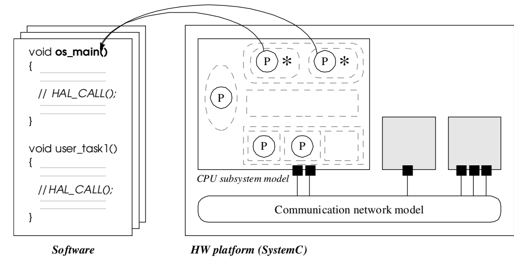

figure 5 : overview of the simulation environment

A CPU subsystem model is represented by a unique SystemC module that provides a set of hardware services throughout its associated ports. The abstract CPU subsystem model elements are represented by dashed lines. They are implemented as C++ objects whose interfaces export the different HAL services. In the figure, the unique SystemC objects within the CPU subsystem module are processes (circles) and events (asterisks).

For each PU, we associate a SystemC process (SC_THREAD) and a dynamic event. The process is responsible of running the application behavior. In the figure, the execution unit is composed of two PU that run the same code in a homogeneous single program multiple data (SPMD) fashion. A SystemC process is also needed to model the run-time behavior of the synchronization unit. Similarly, some device elements (typically those that have active behavior) need to have associated processes.

Modeling SW time

To enable time accurate simulation, SW execution time has to be modeled. This is achieved by performing static annotations within the original SW code. This kind of code instrumentation is well covered in the literature [7]. Given a processor type, the time needed by a SW basic block to execute is estimated. The SW application code is then instrumented accordingly by annotating each basic block using its corresponding delay. These annotations will correspond to “wait” statements.

However, the statically estimated delays do not take into account the possible occurrence of hardware interrupts, nor do they take into consideration the effect of stalls associated with concurrent accesses to the same address space entity. To solve this problem, we use a special annotating function (*consume*) which implements an appropriate algorithm based on the dynamic sensitivity of the SystemC *wait(delay,event)* function.

Fig.6 the run-time behavior of the *consume()* function, where two parallel elements, each having its own access port (a PU and a DMA device for instance) concurrently access the same address space entity.

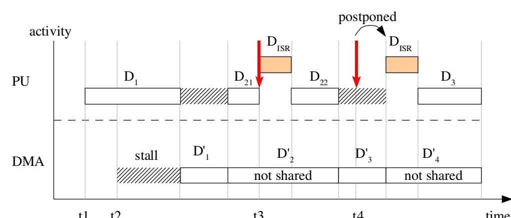

Figure 6: example of “consume” run-time behavior

V. Case study : real-time DivX encoder

Fig. 7 shows the specification of the MPEG4 real-time encoder application. The input module receives a stream of non-compressed video data (QCIF). Each video frame is then split into 4 sub-frames which are sent to 4 parallel encoding modules. The processed data is then sent to a variable length encoding module (VLC) which also reconstructs the entire frame before forwarding it to the output module.

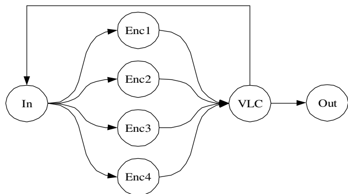

Figure 7: Specification of the DivX application

Target architecture and simulation environment

The target architecture is built around a DMA engine network. Each encoder module consists in a ARM7 based CPU sub-system. The VLC module is mapped on a ARM9 based CPU sub-system, while input and output are mapped on dedicated HW IPs.

Fig. 8 shows the HAL level SystemC simulation model of the DivX application. The DMA engine is modeled at TLM level. For each CPU subsystem, we use a (proprietary) RTOS that was slightly adapted to rely on the same HAL API as that provided by the abstract CPU subsystem simulation model.

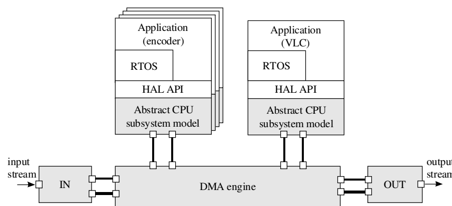

Figure 8 : Simulation model of the DivX application

CPU subsystem model

we experimented two variants of CPU subsystems. In the first one (figure 9a), the input video buffer is made part of the local CPU on-chip memory. Therefore, the network controller has to go through system bus in order to perform a DMA transfer. Access conflicts are resolved using an arbiter implementing a simple FIFO based scheduler. In the second configuration (figure 9b), a dedicated double bank video buffer is used. This allows computation be performed in parallel with external data transfer.

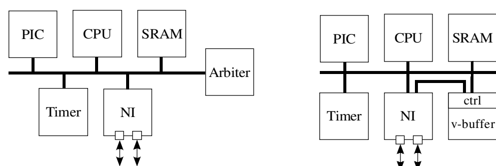

Figure 9 : two CPU subsystems variants

We first run the simulation at HAL level with corresponding CPU subsystem simulation models. Then, we performed a cycle accurate simulation using instruction set simulators (Armulator). This step corresponds to the refinement of the abstract CPU subsystems i.e using real hardware components for the CPU subsystem and associated real HAL software.

The obtained results are summarized in table 1. The test-bench consists in a 1 second video sequence. The different ARM processors are cadenced at 60 MHz

	Execution time (s)		*Average error*	*Simulation time*		*Speed-up*
	ISA	HAL		ISA	HAL	
Enc	1,337	1.378	3%	8h	22s	x1300
VLC	0.840	0.889	6%			
Enc	0.918	0.938	2%	8h	16s	x1800
VLC	0.622	0.653	5%			

Table 1 : experimental results

The upper and lower part of the table correspond to the first and second CPU subsystem variants respectively.

The first column shows the active time consumed by the different CPU's in order to process the 1 second video sequence. This time is measured in both ISA and HAL based simulations.

The second column computes the average error of the HAL level simulation compared to the ISA one (assuming a 100% accuracy for the later). The obtained results show that :

- The first CPU subsystem variant failed to be real time
- The error relative to the VLC module is more important because of the ARM9 pipeline architecture.
- Errors are slightly more important in the case of the first CPU subsystem architecture (without double bank memory) because of the additional inaccuracy introduced by the bus arbiter simulation model.

The last two columns are related to simulation speed. We clearly see that, compared to ISA based simulation, the HAL level simulation achieves a considerable speedup (more than 3 orders of magnitude). The speedup is slightly degraded in the first architecture configuration because of the overhead introduced by the bus scheduling process.

VI. Conclusion

In this paper, we presented an abstract CPU subsystem model targeting MP-SoC design. The proposed model captures most of the inherent SoC specificities in terms of computation, data transfer, and synchronization while offering a high level view of the CPU subsystem. A simulation environment implementing such model was also described on top of SystemC. The obtained results demonstrate a considerable simulation speedup compared to a classic ISS based simulation and a reasonable accuracy, which are key enablers for early design space exploration.

References

[1] S. Yoo, A.A Jerraya "Introduction to Hardware Abstraction Layers for SoC." In Proc. of DATE, Mar. 2003.

[2] A. Fauth and J. Van Praet and M. Freericks. "Describing Instruction Set Processors Using nML". In Proc. of the European Design and Test Conference, Mar. 1995.

[3] S.Pees, A.Hoffman. "LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures". In Proc. Design Automation Conference, 1999.

[4] D. Desmet, D. Verkest and H. De man. "Operating system based SW generation for system-on-chip". In Proc. Design Automation Conference, Jun. 2000.

[5] Andreas Gerstlauer, Haobo Yu, Daniel D. Gajski, "RTOS Modeling for System-Level Design" Proc. of Design, Automation & Test in Europe, Munich, Germany, March 2003.

[6] SystemC. Available : http://www.systemc.org/

[7] J. R. Bammi , W. Kruijtzer , L. Lavagno , E. Harcourt , M. T. Lazarescu, "Software performance estimation strategies in a system-level design tool". In Proc. of the eighth international workshop on Hardware/software codesign, p.82-86, May 2000