Programming models and HW-SW Interfaces Abstraction for Multi-Processor SoC

Ahmed A. Jerraya
TIMA Laboratory
46 Ave Felix Viallet
38031 Grenoble CEDEX, France
+33476574759
Ahmed.Jerraya@imag.fr

Aimen Bouchhima
TIMA Laboratory
46 Ave Felix Viallet
38031 Grenoble CEDEX, France
+33476574301
Aimen.Bouchhima@imag.fr

Frédéric Pétrot
TIMA Laboratory
46 Ave Felix Viallet
38031 Grenoble CEDEX, France
+33476574870
Frederic.Petrot@imag.fr

ABSTRACT

For the design of classic computers the Parallel programming concept is used to abstract HW/SW interfaces during high level specification of application software. The software is then adapted to existing multiprocessor platforms using a low level software layer that implements the programming model. Unlike classic computers, the design of heterogeneous MPSoC includes also building the processors and other kind of hardware components required to execute the software. In this case, the programming model hides both hardware and software refinements. This paper deals with parallel programming models to abstract both hardware and software interfaces in the case of heterogeneous MPSoC design. Different abstraction levels will be needed. For the long term, the use of higher level programming models will open new vistas for optimization and architecture exploration like CPU/RTOS tradeoffs.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms: Algorithms, Design, Standardization, Languages.

Keywords: Programming models, HW/SW interfaces, Heterogeneous MPSoC

1. INTRODUCTION

1.1 Who needs Heterogeneous MPSoC ?

Heterogeneous MPSoCs are no more an advanced research topic for academia [1]. 90% of SoC designed since the start of the 130nm process, include at least one CPU [2]. 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) [3]. Within such architectures, the increasing computation power is addressed by matching the architecture to the application domain rather than by increasingly using powerful general purpose computing engines [4] or duplicating the same core to increase parallelism. The availability of methods for the automatic generation of application specific processors (ASIP) seems to accelerate this tendency. The possibility of generating custom processors to execute specific tasks makes it possible to adapt the cost and the performances of the various computation nodes to a given application. More and more functions, traditionally executed by specific but inflexible hardware, will now be executed by software on dedicated programmable processors [5].

Programming these application specific heterogeneous multiprocessor architectures will be the key issue because of two contradictory requirements: (1) Reducing software development cost and overall design time requires a higher level programming model. This reduces the amount of architecture details that need to be handled by application software designers and then speed up the design process. The use of higher level programming model will also allow concurrent software/hardware design and thus reduces the overall design time. (2) Improving the performance of the overall system requires finding the best matches between hardware and software. This is generally obtained through low level programming. The challenge is then to find a programming model able to satisfy these two contradictory requirements.

This paper deals with parallel programming models to abstract HW-SW Interfaces in the case of heterogeneous MPSoC. The rest of this section analyzes the difference between MPSoC and classical multiprocessor architecture. The next section defines HW/SW interfaces in the case of heterogeneous MPSoC architectures. Section 3 uses the concept of programming model at different abstraction levels. Section 4 presents a new design flow based on the concept of high level programming model.

1.2 MPSoCs vs. classic distributed computers.

The use of heterogeneous ASIPs makes heterogeneous MPSoC architectures fundamentally different from classic general purpose multiprocessor architectures. For the design of classic computers, the parallel programming concept (e.g. MPI) is used as an Application Programming Interface (API) to abstract HW/SW interfaces during high level specification of software applications. The application software can be simulated using an execution platform of the API (e.g. MPICH) or executed on existing multiprocessor architectures that include a low level software layer to implement the programming model. In this case the overall performances obtained after hardware/software integration cannot be guaranteed and will depend on the match between the application and the platform. Unlike classic computers, the design of MPSoC requires a better matching between hardware and software in order to meet performances requirements. In this case, the HW/SW interfaces implementation is not standard; it needs to be customized for a specific application in order to get the required performances.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

This includes customizing the CPUs and all the peripherals required to accelerate communication and computation. In most cases, even the lower software layers need to be customized to reach the required cost and performances constraints. Applying the classical design schemes for those architectures leads to inefficient designs.

Additionally, Classic SoC design flows imply a long design cycle. Such flows rely on a sequential approach where complete hardware architecture should first be developed before software could be designed on top of it. This long design cycle is not acceptable because time to market constraints. The use of high level programming model to abstract hardware/software interfaces is the key enabler for concurrent hardware and software design. This abstraction allows to separate low level implementation issues from high level application programming. It also smoothes the design flow and eases the interaction between hardware and software designers. It acts as a contract between hardware and software teams that may work concurrently [6]. Additionally, this scheme eases the integration phase since both hardware and software have been developed to comply with a well defined interface. The use of a parallel programming model allows reducing the overall system design time and cost in addition to a better handling of complexity.

The use of programming models for the design of heterogeneous MPSoC requires the definition of new design automation methods to enable concurrent design of hardware and software. This will also require new models to deal with non standard application specific hardware/software interfaces at several abstraction levels.

2. HW-SW ARCHITECTURE FOR MPSOC

The literature relates mainly two kinds of organizations for multiprocessor architectures. These are called shared memory and message passing [7]. This classification fixes both hardware and software organizations for each class. The shared memory organization generally assumes a multithreaded application organized as a single software stack and hardware architecture made of multiple identical CPUs. The communication between the different CPUs is made through a global shared memory. The message passing organization assumes multiple software stacks running on non identical subsystems that may include a different CPU and/or a different I/O system in addition to specific local memory architecture. The communication between different subsystems is generally made through message passing. Heterogeneous MPSoC generally combine both models to integrate a massive number of processors on a single chip [8]. Future heterogeneous MPSoC will be made of few heterogeneous subsystems where each may include a massive number of the same processor to run a specific software stack.

2.1 Hardware Architecture

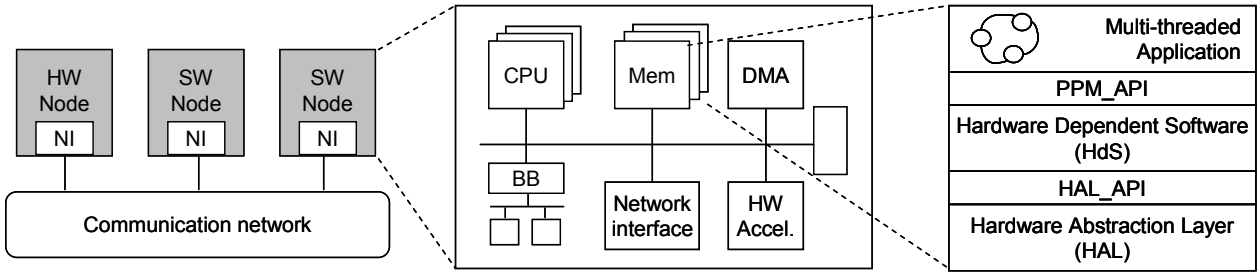
MPSoC architectures may be represented, without loss of generality, as a set of processing nodes or components which interact via a communication network (figure 1-a). Processing nodes may be either hardware or software. Software nodes (figure 1-b) are programmable sub-systems that include one or several identical processing unit. Different kinds of processing unit may be needed for the different subsystems to realize different functionalities (e.g DSP for data oriented operations, GPP for control oriented operations and ASIP for application specific computation). In addition to the CPU, the hardware part of a software node generally includes additional components to speed up communication. This may range from simple bus arbitration to sophisticated memory and parallel I/O architectures.

2.2 Software Architecture

In classical literature, a node is organized into layers for the purposes of standardization and reuse. Unfortunately each layer induces additional cost and performances overheads. In this paper we consider that within a node, embedded software is structured in only three layers as depicted in figure 1-c. The top layer is the software application that may be a multi-threaded description or a single thread function. This model makes use of parallel programming model (PPM_API) to abstract the underlying platform. This separation is required for concurrent software and hardware development. The second layer consists in the Hardware Dependent Software (HdS).

The HdS layer is responsible of providing the necessary services to manage and share resources. This includes for instance the scheduling of application threads on top of the available processing elements, inter-task communication, external communication, and all other kinds of resources management and control. Conventionally these services are provided by the operating system and additional libraries for communication middleware. The federative HdS term underline the fact that, in an embedded context, we are concerned with application specific implementations of these functionalities that strongly depend on the target hardware architecture.

At this level, the hardware dependency is kept functional, i.e. it concerns only high level aspects of the hardware architecture like the type of available resources. Low level details about how to access these resources are then abstracted by the third layer, which is the hardware abstraction layer (HAL). The separation between HDS and HAL makes easier architecture exploration for the design of both the CPU subsystem and the operating system services.



a) MPSoC generic architecture (b) hardware architecture of a software node (c) Embedded Software architecture

Figure 1. Heterogeneous MPSoC Architecture

2.3 Abstracting Heterogeneous MPSoC

Existing SoC design practices are based on all explicit HW/SW interfaces. Figure (2-a) shows the classic view of a SoC. In this view, the ultimate interface between embedded software and the underlying hardware is the processing element (CPU). According to this view, a model of the overall system will be composed of a HDL based model of the hardware parts (generally an RTL model) and an instruction set simulator (ISS) that interpret a set of binary instructions using an explicit memory and I/O architectures obtained by compiling/linking the embedded software.

Ideally one would like to start with a set of SW tasks communicating with a set of HW subsystems (figure 2-d). This could be viewed as a functional specification of the application where HW/SW interfaces are fully abstracted. This model even hides that software components run on processors. Of course, during the design process, the HW/SW interface refinement needs to handle two different interfaces: one on the software side using API and one on the hardware side using wires. Once refined, the abstract HW/SW interface eventually results in a set of heterogeneous subsystems including a CPU subsystem, a Hardware dependent Software (HdS) layer and a Hardware abstraction Layer (figure 2-c). During the design process several intermediate steps will use an abstract model of HW/SW interfaces to separate low level implementation issues from high level application related aspects (Figure 2-b). The key issues in order to make possible the transition between these different models are (1) The existence of a well defined programming model allowing to abstract HW/SW interfaces during the different steps. This will be detailed in the next section. (2) The definition of a new design flow able to handle HW/SW interfaces at different abstraction levels. This will be detailed in Section 4.

3. PROGRAMMING MODELS

Tools exist for automatic mapping of sequential programs on homogeneous multiprocessor architectures. Unfortunately these are not efficient for heterogeneous MPSoC architectures. In order to allow the design of distributed applications, parallel programming models have been introduced and extensively studied by software communities for high level programming of heterogeneous multiprocessor architectures.

3.1 Programming models used in SW

As long as only software is concerned, Skillicorn [10] identifies 6 key concepts that may be hidden by the programming model. These are concurrency, decomposition, mapping, communication and synchronization. These concepts define 6 different abstraction levels for parallel programming models. These are summarized in Table 1 that gives the different levels with typical corresponding parallel programming languages for each of them. All these programming models take into account only the software side. They assume the existence of lower levels of software and a hardware platform able to execute the corresponding model. The mapping of tasks on computation nodes is still implicit.

Table 1: The six programming levels as defined by Skillicorn

Abstraction level	*Typical languages*	*Explicit concepts*
Implicit Concurrency	PPP, Crystal	None
Parallel level	Concurrent Prolog	+Concurrency
Thread level	SDL	+ threading
Agent models	Emerald, CORBA	+ Mapping
Process Network	KPN	+ Communication
Message Passing	MPI, OCCAM	+ Synchronization

3.2 Programming models for SoC design.

In order to allow for concurrent HW/SW design, we need abstract HW/SW interfaces including both software and hardware components. Similar to programming models for SW, HW/SW interfaces may be described at different abstraction levels. The 4 key concepts that we consider are: explicit HW resources, management and control strategies of HW resources, the CPU architecture, and the CPU implementation.

These concepts define four new abstraction levels that we name System level, Virtual Architecture level, Transaction Accurate level and Virtual prototype level. These four levels are presented in Table 2.

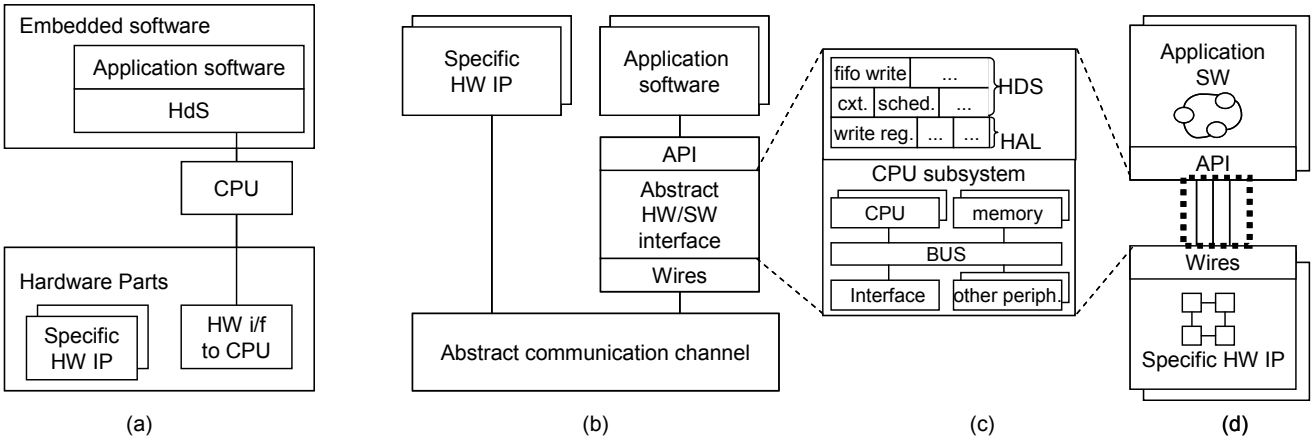


Figure 2. SoC abstraction views: (a) Classic view (b) Abstract HW-SW view (c) Implementation view (d) Functional view

Table 2. Additional Models for SoC Design

Abstraction level	*Typical programming languages*	*Explicit concepts*
System level	MPI, Sumulink	All functional
Virtual Architecture	Untimed SystemC[11]	+Abstract Resources
Transaction Accurate	TLM SystemC [11]	+Resources sharing and Control strategies
Virtual Prototype	Cosimulation with ISS	+ ISA & detailed I/O interrupts
RTL	HDL	+CPU implementation and Reset sequences

At the system level all the hardware is implicit similar to the message passing model used for software. At the Virtual Architecture level, HW/SW partitioning and resources allocation are made explicit. Also the allocation of threads/tasks to subsystem is fixed. This model combines both the specification of the application and the architecture. It is also called Combined Architecture Algorithm Model (CAAM) [12]. At the Transaction Accurate level the resources management and control strategies become explicit. On the software side this fixes the RTOS and on the hardware side a functional model of the bus is defined. The software interface is specified to the HAL level while the hardware communication is defined at the bus transaction level. The virtual prototype level corresponds to the classical co-simulation model with ISS [15]. At this level the architecture of the CPU is fixed but not yet its implementation that remains hidden by an ISS. Finally at the RTL level, the CPU implementation is fixed and all the hardware parts of the system are detailed. SW needs also to be detailed at the binary level in order to allow an execution of the overall system.

3.3 Defining a programming model for SoC

A programming model is made of a set of functions (implicit and/or explicit primitives) that can be used by SW to interact with HW. It should cover all the functionalities of both the HDS and the CPU subsystem. Additionally, the programming model needs to cover the four levels required for SoC refinement.

In order to cover different abstraction levels of both HDS and CPU subsystems, the programming model needs to include three kinds of primitives:

- Communication primitives: these are aimed to exchange data between the HW and the SW.
- Task and resources control primitives: these are aimed to handle task creation, management and sequencing. At system level, these primitives are generally implicit and built in the constructions of the language. The typical scheme is the Module Hierarchy in block structure languages where each module declares implicit execution threads.
- HW access primitives: these are required when the architecture includes specific HW. These include specific primitives to implement specific protocol or I/O schemes, for example a specific memory controller allowing multiple accesses. These will always be considered at lower abstraction layers and cannot be abstracted using the standard communication primitives.

The programming models at different abstraction levels are summarized in Table 3. The different abstraction levels may be expressed by a single and unique parallel programming model that uses the same primitives applicable at different abstraction levels or uses different primitives for each level. In practice mixed level description is required to enable inter-layer optimization when specifying HW/SW interfaces.

3.4 Existing programming models

A number of MP-SoC specific programming models, based on shared memory or message passing, have been defined recently. The Task Transaction Level interface (TTL) proposed in [13] focuses on stream processing applications in which concurrency and communication are explicit. The interaction between tasks is performed through communication primitives with different semantics, allowing blocking or non blocking calls, in order or out of order data access, and direct access to channel data. The TTL API defines three abstraction levels: the *vector_read* and *vector_write* functions are typical system level functions in which synchronization and data transfers are combined, the *reAcquireRoom* and *releaseData* functions (*re* stands for relative) grant/release atomic access to vectors of data that can be loaded or stored out of order, but relative to the last access,*i.e.* with no explicit address. This corresponds to our virtual architecture level. Finally, the *AcquireRoom* and *releaseData* lock and unlock access to scalars, for which explicit addressing schemes must be defined. This corresponds to the transaction accurate level.

The Multiflex approach proposed in [8] targets multimedia and networking applications, with the objective of having good performance even for small granularity tasks. Multiflex supports both the Symmetric Multi Processing approach used on shared-memory multiprocessors and a remote procedure call based programming approach called DSOC (Distributed System Object Component). The SMP functionality is close to the one provided by POSIX, *i.e.* thread creation, mutexes, condition variables, etc, and the DSOC uses a broker to spawn the remote methods.

Table 3. Programming Model API at different abstraction levels

	Communication primitives	*Task and Resources control*	*HW access Primitives*
System Level	data exchange e.g Send/Receive(Data)	Implicit e.g. threads in SystemC.	Functional access to specific resources
Virtual architecture	+synchronization e.g; Posix threads, Lock/unlock (data)	Explicit tasks control e.g. Create/resume(task)	Specific I/O protocols related to architecture.
Transaction accurate	Data access with specific addresses e.g; read/write(data, adr)	HW managements of resources e.g. Context switch and test/set.	Physical access to HW resources
Virtual prototype	HW dependant I/O e.g. DMA and memory mapped I/O.	HW arbitration and address translation e.g. Memory Map	Physical I/Os
RTL	Load/store	Interrupts & Jumps	Physical I/Os

These abstractions make no separation between Virtual Architecture and Transaction levels since they rely on fixed synchronization mechanisms. Hardware support to locks and run queues management is provided by a concurrency engine, and the processors have several hardware contexts to allow context switches in one cycle. DSOC uses a CORBA like approach, but implements hardware accelerators to optimize the performances.

In the previous section (table 3), we showed that a suitable parallel programming model for MPSoC needs to be defined at several abstraction levels corresponding to different design steps. This hierarchical view of the parallel programming model ensures a seamless implementation of higher level APIs on lower level ones. In order to ensure a better match between the parallel programming model and the underlying hardware architecture, the API also has to be made extensible, at each abstraction level, to cope with the broad range of possible hardware components. The existing MPSoC programming models seem either to focus on one aspect or the other. We think that it is important to consider both aspects i.e hierarchy and extensibility when designing an MPSoC oriented parallel programming model. [12] introduces the concept of service dependency graph to represent HW/SW interface at different abstraction levels and handle application specific API. This model represents the HW/SW interface as a set of interdependent components providing and requiring services.

4. NEXT GENERATION DESIGN FLOWS

The use of a programming model to hide HW/SW interfaces at different abstraction levels removes the discontinuity between hardware and software during the different steps of the design process [12].

Figure 3 illustrates the proposed design methodology to refine hardware software interfaces. This flow starts with a system-level specification made of functions using a system-level parallel programming model. This may be an MPI or Simulink functional model that can be simulated using the corresponding environment. A four step method is used to produce a custom HW/SW architecture from this system-level specification using the abstraction levels defined in section 3.2. Each of these steps allows for global validation of the whole system including software and hardware components. The availability of fast high level simulation makes possible the architecture exploration at different abstraction levels. Even if the flow is far from being automated, several experiments have shown that this is a working scheme that accelerates MPSoC design process and still producing reasonable quality design. The overall methodology is introduced in [12] and different experiments are reported in [14,15,16]. The rest of this section gives 2 snapshots of these experiments.

Table 4 shows experiments reported in [14] on simulation at different abstraction levels. All the simulations were made using SystemC. At virtual architecture and transaction accurate level, the simulations are based on native execution of software. Timing annotation was used for both virtual architecture and transaction accurate levels. The time accuracy measures the timing error when compared to RTL simulation and the speedup measures the speed factor gain when compared to untimed system level simulation. It is important to note that virtual architecture simulation model allows quite precise timing estimation (80%) that is good enough to perform architecture exploration at partitioning level. The transaction-level model includes a high-level model of the CPU subsystem and HDS. The simulation at this level validates the partitioning between HDS and the CPU.

Table 4: Simulation at different abstraction levels

Abstraction level	Speedup	Time Accuracy
RTL	10^{-5}	100%
Virtual Prototype	10^{-3}	99%
Transaction Level	10^{-2}	95%
Virtual Architecture	1	80%
System Level	1	10%

Figure 4 shows the results of another experiment, reported in [16] for the design of a heterogeneous MPSoC architecture customized for an MPEG4 application. This architecture includes 2 different kinds of CPU subsystems using different architectures and different HDS.

The experiment details an architecture exploration process to codesign the HDS and the CPU subsystems in order to reach the required performances. The obtained architecture makes use of two kinds of CPU subsystems with two different memory architectures. The HDS is customized for each subsystem.

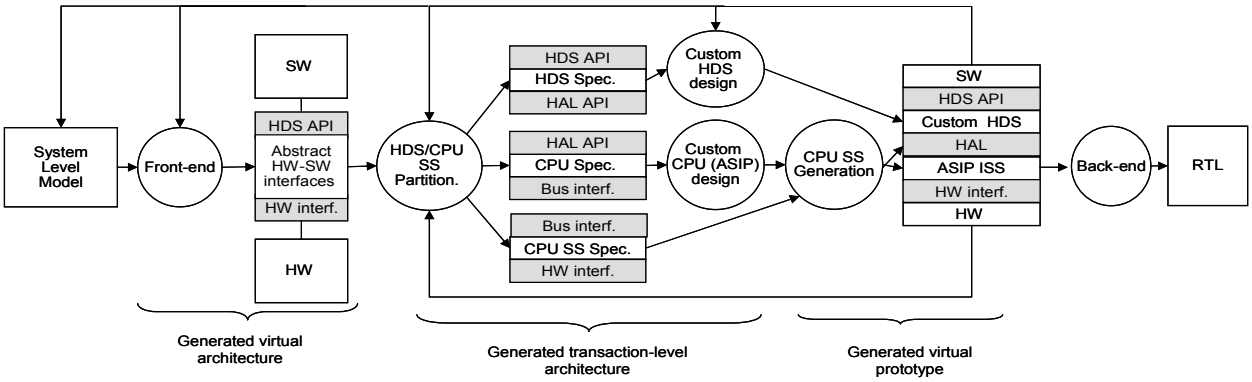


Figure 3. Proposed HW/SW Interfaces design flow

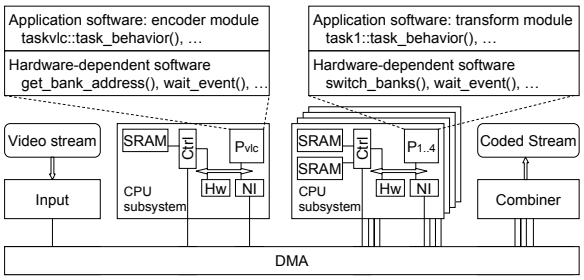


Figure 4. Implementation of MPEG-4 encoder.

This experiment shows that the use of a high level parallel programming model to abstract hardware/software interfaces is a working concept and may be the right approach to build higher than RTL design automation tool. It allows early validation and makes easier classical architecture exploration at partitioning level. Additionally, this scheme opens the design process to several new optimizations that were not possible when using the classical separate HW and SW design scheme. The most obvious optimization is a better adaptation of the CPU to both HW and SW interfaces. For example, new flexible processor technologies such as Tensilica [17] can be used to optimize performances of the HW/SW interfaces by introducing application-specific I/O operations.

5. Conclusions

In this paper we discussed the use of high level programming for the abstraction of HW-SW interfaces. A programming model is made of a set of functions (implicit and/or explicit primitives) that can be used by the SW to interact with HW. We tried to adapt the concept, initially used to coordinate software and hardware communities for the design of classic computers, to heterogeneous MPSoC design. In the case of heterogeneous MPSoC design, the programming model hides both hardware and software refinements. It should cover all the functionalities of both the lower software layers and the CPU subsystem. The use of a high level parallel programming model requires new design flows and opens new vistas for optimization and architecture exploration like CPU/RTOS tradeoffs.

6. ACKNOWLEDGMENTS

This work is supported by European projects SPRINT and SHAPES, MEDEA+ project LOMOSA+ and ITEA project MERCED.

7. REFERENCES

[1] J. Turley. Survey says: Software tools more important than Chips. *Embedded Systems Design Journal*. 4-11-2005.

[2] H. Jones. Analysis of the relationship between EDA Expenditures and Competitive Positioning of IC Vendors for 2003. *http://www.edac.org/resources_profitability.jsp*

[3] A.A. Jerraya, W. Wolf and H. Tenhunen, Guest Editors. *IEEE Computer, Special Issue on MPSoC*. Volume 38 Number 7, pp. 36-40, July 2005.

[4] G. Fettweis, H. Meyr. Applications, Architectures, Design Methodology and Tools for MPSoC. Embedded Tutorial. *DATE'06*, Munich, Germany, March 2006.

[5] W. Wolf. *High-Performance Embedded Computing*. Morgan Kaufman. 2006

[6] C. Berthet. Going Mobile: The Next Horizon for Multi-million Gate Designs in the Semi-Conductor Industry. *In Proceedings of 39th Design Automation Conference*, USA, June 2002.

[7] D. Culler, J.P. Singh, and A. Gupta**.** *Parallel Computer Architecture: A Hardware/Software Approach***.** Morgan Kaufmann, August 1998, ISBN 1558603433

[8] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, G. Nicolescu. Parallel Programming Models for a Multi-Processor SoC Platform Applied to Networking and Multimedia. *IEEE Transactions on VLSI Journal*, 2006.

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

[10] D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, **v**ol. 30, issue 2, pp 123 – 169, 1998.

[11] F. Ghenassia. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, 2005, ISBN 0-387-26232-6.

[12] A. Bouchhima, X. Chen, F. Pétrot, W. Cesario, A.A. Jerraya. A Unified HW/SW Interface Model to Remove Discontinuities between HW and SW Design. In Proceedings of *EMSOFT 2005*, Jersey City NJ, USA, Sept. 18-22, 2005.

[13] P. van der Wolf, E. de Kock, T. Henriksson W Kruijtzer and G. Essink. Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach. Special Session. *In Proceedings of CODES+ISSS 2004***.** Stockholm, Sweden, Sept. 2004.

[14] A. Bouchhima, S. Yoo, A.A. Jerraya. Fast and Accurate Timed Execution of High Level Embedded Software Using HW/SW Interface Simulation Model. In Proceedings of *ASP-DAC 2004*, Yokohama, Japan, January 2004.

[15] S. Yoo, M.W. Youssef, A. Bouchhima, A.A. Jerraya, M. Diaz-Nava. Multi-Processor SoC Design Methodology Using a Concept of Two-Layer Hardware-Dependent Software. In Proceedings of *Design Automation and Test in Europe, DATE'04*, Paris, France, February 2004.

[16] A. Bouchhima, L. Kriaa, W. Youssef, P. Gerin, F. Pétrot, A.A. Jerraya. A Unified HW/SW Interface Refinement Approach for MPSoC Design. In Proceedings of *The 4th International IEEE-NEWCAS Conference NEWCAS 2006*, Gatineau, Canada, June 18-21, 2006.

[17] C. Rowen. *Engineering the Complex SoC*. Prentice Hall, 2004.