Platform-Based Software Design Flow for Heterogeneous MPSoC

KATALIN POPOVICI, XAVIER GUERIN, and FREDERIC ROUSSEAU
TIMA Laboratory
PIER STANISLAO PAOLOCCI
ATMEL Roma Advanced DSP
and
AHMED AMINE JERRAYA
CEA-LETI, MINATEC

Current multimedia applications demand complex heterogeneous multiprocessor architectures with specific communication infrastructure in order to achieve the required performances. Programming these architectures usually results in writing separate low-level code for the different processors (DSP, microcontroller), implying late global validation of the overall application with the hardware platform. We propose a platform-based software design flow able to efficiently use the resources of the architecture and allowing easy experimentation of several mappings of the application onto the platform resources. We use a high-level environment to capture both application and architecture initial representations. An executable software stack is generated automatically for each processor from the initial model. The software generation and validation is performed gradually corresponding to different software abstraction levels. Specific software development platforms (abstract models of the architecture) are generated and used to allow debugging of the different software components with explicit hardware-software interaction. We applied this approach on a multimedia platform, involving a high performance DSP and a RISC processor, to explore communication architecture and generate an efficient executable code for a multimedia application. Based on automatic tools, the proposed flow increases productivity and preserves design quality.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques—Computer-aided software engineering (CASE); D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures; C.3 [Computer Systems Organization]: Special purpose and application-based systems

This research was supported by the SHAPES European Project (FET-FP6-2004-IST-4.2.3.4(viii)-started January 2006).

Authors’ addresses: K. Popovici, X. Guerin, and F. Rousseau, TIMA Laboratory, 46 Avenue Félix Viallet, F38031 Grenoble, France; email: katalin.popovici@imag.fr, xavier.guerin@imag.fr, frederic.rousseau@imag.fr; Pier Stanislas Paolucci, ATMEL Roma Advanced DSP, Via V. G. Galati 91, 00155 Roma, Italy, email: pier.paolucci@atmel.com. Ahmed Amine Jerraya, CEA-LETI, MINATEC, 17 Rue des Martyrs, 38054, Grenoble, France, email: ahmed.jerraya@cea.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.


1. INTRODUCTION

Multimedia applications include heterogeneous functions that require different kinds of processing units (DSP for complex computation, microcontroller for control functions etc.) and different communication schemes (fast links, non-standard memory organization, and access).

To achieve the required computation and communication performances, heterogeneous MPSoC architecture with specific communication components seems to be a promising solution [Meyr 2006]. Typical heterogeneous platforms used in industry are TI OMAP [TI] for cellular phones, Philips Viper, and Nexperia [Nexperia] for consumer products or the Nomadik platform [ST]. They incorporate a DSP processor and a microcontroller, communicating via efficient, but sophisticated, infrastructure. For this type of architecture, classic programming environments do not fit: (1) high-level programming does not efficiently handle specific I/O and communication schemes, while (2) low-level programming explicitly managing specific I/O and communication is a time-consuming and error-prone activity. In practice, programming these heterogeneous architectures is done by developing separate low-level codes for the different processors, without any global validation of the overall application with the hardware platform. The validation can be performed only when all the binary software is produced and can be executed on the hardware platform.

Next-generation MPSoC seems to accelerate this trend in complex hardware architecture more when multiple DSP and microcontrollers will be integrated on a single chip [Turley 2005]. Hence, the key challenge is how to efficiently program such architectures, starting from a high level language. The different types of processors execute different software stacks. An additional difficulty is to debug and validate the lower software layers required to map the high-level application code on the target heterogeneous architecture.

In this article, we use an architecture example, the Shapes MPSoC architecture [Paolucci et al. 2006], which is a multi-tile architecture based on a Diopsis tile (Figure 1). The Diopsis tile is a triple-core system integrating an ATMELaGicV VLIW DSP [Atmel Inc.], an ARM9 RISC microcontroller, and a network processor (DNP). The system combines the flexibility of the ARMc9 controller with the high performance of the DSP and the on-chip/off-chip networking capability of the DNP. The local memories of the DSP and RISC can be accessed by both processing units. In addition, a distributed external memory (DXM) can be used to share data between all the processors. The data transfer between

these processors can follow different schemes based on an AMBA bus, e.g., the DSP can read/write data to the local memory of the ARM by using or not using a DMA transfer.

Efficient programming requires the use of the characteristics of the architecture [Culler et al. 1998]. For instance, a data exchange between two tasks mapped on different processors, in the case of Diopsis tile, may use different schemes through either the shared or the local memory of one of these processors. In addition, different synchronization schemes (polling, interrupts) may be used to coordinate this exchange. Each of these communication schemes has advantages and disadvantages in terms of performance (latency, throughput), resource sharing (multitasking, parallel I/O) and communication overhead (memory size, execution time). The ideal scheme would be able to produce an efficient software code starting from a high-level program using generic communication primitives, such as \texttt{send/recv} provided by MPI [MPI].

In an ideal design flow, the software generation targeting a specific architecture consists of partitioning and mapping, final application software code generation and hardware-dependent software (HdS) code generation (Figure 2a).

The HdS is made of lower software layers that may incorporate an operating system (OS), communication management, and a hardware abstraction layer to allow the OS functions to access the hardware resources of the platform. Unfortunately, we are still missing such an ideal generic flow, able to efficiently map high-level programs on heterogeneous MPSoC architectures. In addition,
the validation and debugging of Hds remains the main bottleneck in MPSoC design [Wolf 2006], because each processor subsystem requires specific Hds implementation to be efficient.

The classical approaches for the software design use programming models to abstract the hardware architecture (Figure 2b). These generally induce discontinuities in the software design, i.e. the software compiler ignores the processor architecture (e.g. interrupts or specific I/Os). To produce efficient code, the software needs to be adapted to the target architecture by using specific libraries, such as system library for the different hardware components or specific memory mapping for the different CPU and memory architectures.

In this article, the design and validation of the software is performed by using a development platform. As shown in Figure 3, the software development platform is an abstract model of the architecture in the form of a run-time library or simulator aimed to execute the software [Sangiovanni-Vincetelli et al. 2001]. The combination of the platform with the software code produces an executable model that emulates the execution of the final system, including hardware and software architecture. This executable model allows simulation of the software with detailed hardware–software interaction, software debug, and eventually performance measurement. The platform and the software may be combined using different schemes.

The key contribution of this article is a platform-based software generation flow, which allows software code generation, software development platform
Platform-Based Software Design Flow for Heterogeneous MPSoC

![Software development platform diagram]

Fig. 3. Software development platform.

generation and simulation models generation for the software debug. The flow is able to efficiently use the resources of the architecture, allowing easy experimentation with several mappings of the communication onto the platform resources. We use Simulink as a high-level environment to capture the initial specification of the application. The proposed flow is based on automatic tools. Thus, it increases design productivity.

The input model captures both application and architecture representations. It makes explicit the software distribution on different processors and the used communication schemes used. The executable software stacks are generated automatically for each processing unit from the initial model. Each software stack is structured into layers to allow flexibility in terms of software components reuse (OS and communication protocol) and portability to other platforms. These different software components are generated and validated incrementally and correspond to four different software abstraction levels: system architecture, virtual architecture, transaction accurate architecture and virtual prototype. To allow the software validation, different development platforms are also automatically generated at each abstraction level.

The rest of the article is organized as follows. Section 2 relates previous work. Section 3 presents an overview of the proposed software design flow. Section 4 presents the combined software/hardware representation at the system architecture level. Section 5 describes the virtual architecture hardware and software models. Section 6 deals with the transaction-accurate architecture model. Section 7 presents experiments with mapping the MJPEG application on the Diopsis platform.

2. RELATED WORK

Previous work related to software generation and validation from a high-level environment can be classified in three categories: software-oriented, hardware-oriented or electronic system level (ESL) design-oriented.

The software-oriented approaches make use of a software model in the form of a runtime library to model the interaction with the hardware [Desoli et al. 2002; Magee et al. 2005]. The application can be written in a high-level

language or generated from an UML description or other model-driven-based specification [Balasubramanian et al. 2006; Kangas et al. 2006; Model driven Architecture]. The software stack construction consists of compiling this code and linking the results with the run-time libraries. The library is defined separately for each processor and can be very sophisticated. Such approaches have been already applied for SoC architectures, e.g., YAPI [De Kock et al. 2000]), such as Trimedia platform with DSP and MIPS. The libraries are difficult to port on other processors, which make this approach unaffordable for heterogeneous MPSoC architectures that require fast time to market and architecture exploration to meet the performance requirements. In fact, library porting is fastidious and implies long design time and high complexity for software debugging.

The hardware-oriented approach executes the final software on a virtual platform and it corresponds to classic hardware–software cosimulation models using instruction-set simulators (ISS) [Rowson 1994; Semeria and Gosh 2000]. These techniques require that all the software and hardware are fully specified. Thus, the software validation occurs too late and the debugging process may be too expensive and fastidious.

The ESL oriented approaches use high-level APIs (application programming interface) to abstract the hardware-software interfaces, e.g. DSOC [Paulin et al. 2006], TTL [Van der Wolf et al. 2004]. This approach enables the automatic generation of a virtual prototype from a system-level model, but the generation of HdS software layer is performed in one step, which generally implies the use of predefined communication schemes. Moreover, the gap between the system-level model and generated code makes the debugging of the generated software stacks more difficult because the identification of the different sources of bugs is not obvious.

The proposed software design flow combines all the benefits of the first three previously described methods while mitigating new drawbacks. Our approach is derived from a platform-based design method [Sangiovanni-Vincentelli et al. 2004]. Platform-based design focuses on the creation of abstraction layers in the design flow and investigates the semantic properties of mapping across these layers. It lies on a structured methodology for developing economically feasible software design flows.

The proposed approach starts with a high-level model of the application in allowing fast simulation of the application model. It automatically generates the software code and the corresponding software development platform. The abstract models of the architecture capture the special features of the final hardware platform allowing accurate estimation of performances. Our proposed flow abstracts the hardware–software interfaces by using high-level APIs that hide the architecture details when describing the application. Even if the partitioning and mapping are explicit in our model, the flow still provides enough abstraction to produce a significant gain in productivity. Another key advantage of the approach presented in this article is the incremental generation and validation of the different components of the software stack. This makes the debugging of both application and HdS easier in addition to opening new opportunities like communication mapping exploration.

3. SOFTWARE DESIGN FLOW

The following paragraphs detail the proposed software design flow. First, several software-abstraction levels are defined; the main design steps are then presented.

3.1 Software-Abstraction Levels

In order to allow separate generation and validation of different software components, we use a structured model to represent heterogeneous MPSoC architectures [Jerraya et al. 2006]. Each processor subsystem executes a specific software stack organized in two layers: the application and the hardware-dependent software (HdS) layers. The application layer is associated with the high-level behavior of the heterogeneous functions composing the target application. The HdS layer is associated with the hardware-dependent low-level software behavior, such as interrupts routine services, context switch, and specific I/O control. In fact, the HdS layer includes three components: operating system (OS), specific I/O communication, and hardware-abstraction layer (HAL).

These different components and layers of the software stack correspond to different abstraction levels of the software [Popovici et al. 2007]. At each abstraction level, the architecture platform has to be also abstracted in a format where the software sees a high level interface of the hardware, namely an application programming interface (API). With an API well-defined between the different software layers, the generated software components can be validated on the abstract architecture model and can be reused for the validation of the components generated at the next levels. The highest abstraction level is called system architecture; the lowest one is called virtual prototype. To reduce the large gap between these two marginal levels, we defined two intermediate levels: the virtual architecture and the transaction-accurate architecture level, as it will be detailed in the following paragraph.

Figure 4 illustrates the software abstraction levels for a simplified application made of three tasks (T1, T2, and T3), that need to be mapped on an architecture composed of two processing units and several memory hardware subsystems.

The highest level is the system architecture level (SA) (Figure 4a). In this case, the software is composed of a set of functions grouped into tasks. The function is an abstract view of the behavior of an aspect of the application. Several tasks may be mapped on the same software subsystem. The communication between functions, tasks, and subsystems make use of abstract communication links, e.g., standard Simulink links or explicit communication units that correspond to specific communication paths of the target platform. The corresponding development platform consists of the set of the subsystems. The simulation at this level allows validation of the application’s functionality. In this article we use the standard Simulink environment to represent the system architecture model, but other high-level environments with simulation capabilities are also feasible.

The next level is called virtual architecture level (VA) (Figure 4b). The software is refined to task C code that contains the final application code.
and makes use of HdS API. The communication primitives of the HdS API access explicit communication components. Each data transfer specifies an end-to-end communication path. For example, the functional primitives \texttt{send\_mem(ch,src,size)/recv\_mem(ch,dst,size)} may be used to transfer data between the two processors using a global memory connected to the system bus, where \texttt{ch} represents the communication channel used for the data transfer, \texttt{src/dst} the source/destination buffer, and \texttt{size} the number of words to be exchanged. The software is executed using an abstract model of the hardware architecture that provides an emulation of the HdS API. The software development platform is composed of these abstract subsystems, explicit interconnection component, and storage resources. The simulation at this level allows validation of the final code of tasks and may give useful statistics about the communication requirements. Thanks to the HdS APIs, the tasks code remains unchanged for the following levels.

The next level is called the transaction-accurate architecture level (TA) (Figure 4c). The software is linked with an explicit OS and specific I/O software to implement the communication units. The resulting software makes
use of hardware-abstraction layer primitives (HAL API). The data transfers use explicit addresses, e.g., $\text{read\_mem}(\text{addr, dst, size})/\text{write\_mem}(\text{addr, src, size})$, where $\text{addr}$ represents, the source, respectively, the destination address, and $\text{src/dst}$ represents the local address and $\text{size}$ the size of the data. The software is executed using a more detailed development platform to emulate the network component, the explicit peripherals used by the HAL API, and an abstract computation model of the processor. The simulation at this level allows validating the integration of the application with the OS and the communication layer. It may also provide precise information about the communication performances.

Finally, the HAL API and processor are implemented through the use of a HAL software layer and the corresponding processor part for each software subsystem. This represents the virtual prototype level (VP) (Figure 4d). At the virtual prototype level the communication consists of physical I/Os, e.g., \text{load/store}. The simulation at this level allows performance validation and it corresponds to classical hardware/software cosimulation models with instruction-set simulators [De Kock et al. 2000; Rowson 1994] for the processors and RTL components for the hardware resources.

### 3.2 Software Design Steps

As shown in Figure 5, the proposed platform-based software design flow starts with a hierarchical model. This input model represents a functional description of the application specification, combined with the partitioning and mapping information. Aspects related to the architecture model (e.g., processing units available in the target hardware platform) are combined into the application model (i.e., multiple tasks executed on the processing units). To represent this model we use Simulink. The Simulink model is manually built. It makes explicit the communication units to abstract the intrasubsystem (the communication between the tasks inside a subsystem) and the intersubsystem communication protocols (the communication between different subsystems).

Besides the Simulink model, the software design flow uses also an abstract model of the architecture to generate the software development platform and the global hardware software cosimulation model for each refinement step. The abstract architecture model hides details of the underlying implementation of the hardware platform, but ensures a sufficient level of control that the software code can be validated in terms of performance, efficiency, and reliable functionality [Popovici et al. 2007]. Both initial application and abstract architecture are inputs of the system.

The proposed software design flow generates incrementally the task codes, the \text{HdS} and the final binary code for each processor. Each of these different software components corresponds to the virtual architecture, transaction-accurate architecture, and, respectively, to the virtual prototype abstraction levels. At each step, the previously generated software code is reused and integrated with the newly generated software components.

The proposed design flow also automatically generates the software-development platforms to build an executable model that allows simulating and validating the resulted software component at each abstraction level. The
debug of the resulting software is an iterative process. Different software components need different detail levels of the hardware architecture in order to be validated. For example, the debug of the application tasks code is independent from the synchronization mechanism used on the target hardware architecture, while the debug of the HdS requires an explicit implementation of the communication protocol in the platform. All these requirements are considered during the abstraction of the architecture and generation of the software development platforms at each abstraction level. The software-development platform includes only the set of hardware resources that are required for the validation of the software component at the corresponding abstraction level.

In order to validate the software during the different design steps, we use different execution models adapted to each software abstraction level. The initial simulation at system-architecture level is made in Simulink, while all others are made in SystemC.
3.3 Reduced Diopsis tile

In this article, we consider as target hardware architecture a reduced Diopsis tile. We selected the key components that are important for the software generation and validation. Even after the simplification, the selection of the hardware resources still captures all the possible intratile communication schemes and the specific I/O components. The subset is illustrated in Figure 6.

The architecture incorporates an ARM and DSP subsystem. The ARM subsystem includes the processor core and local memories (SRAM, ROM), while the DSP subsystem includes the DSP core, data memory (DMEM), program memory (PMEM), registers (REG), DMA, interrupt controller, and synchronization component. The hardware nodes consist of external distributed memory subsystem (DXM) and peripherals on tile (POT) subsystem. The POT includes the system peripherals of the RISC processor, e.g., timer, interrupts controller (AIC), but also I/O components, such as serial peripheral interface (SPI). The interconnection between these subsystems is made through an AMBA bus.

For performance reasons, the ARM processor can directly access the data memory and control/status registers of the DSP processor via the AMBA slave interface of the DSP subsystem. In the same way, the DSP core can read/write directly from/to the local memory of the RISC processor by initiating a DMA transfer. Moreover, the processors can store and load data to/from the DXM connected to the AMBA bus or exchange different application data through the communication buffers mapped in the DXM.

The rest of the article will detail the proposed software design flow for programming the Diopsis architecture.
4. SYSTEM ARCHITECTURE

The system-architecture model represents a high level application model annotated with partitioning and mapping information. We use Simulink environment to represent the combined application/architecture initial model. The next sections will detail the system-architecture model.

4.1 System Architecture Platform of Diopsis

The system architecture of the Diopsis tile is illustrated in Figure 7. This model includes three subsystems: two software subsystems (the ARM and the DSP subsystem) and a hardware subsystem (the POT). The Simulink hierarchy is able to capture the mapping of the application on the architecture at a high abstraction level through the decomposition of the system into tasks and subsystems. The Simulink model also includes explicit communication units to capture different communication protocols and resources provided by the architecture. In this case, we use software FIFO (SWFIFO) for intrasubsystem communication for the tasks mapped on the same subsystem; for intersubsystem communication between the different subsystems, we use DSP data memory (DMEM), DSP registers (REG), ARM local memory (SRAM), and distributed external memory (DXM). Each of the communication units can be mapped on a specific communication path and protocol of the final architecture. The number of communication units depends on the application partitioning and mapping decisions on the target architecture.

The system architecture model is annotated with software and hardware architecture parameters to allow the automatic generation and validation of the software stack and development platform.

Examples of software architecture parameters are: OSType, which specifies the name of the OS running on the target processor (Linux, Mutek, eCos or the
Platform-Based Software Design Flow for Heterogeneous MPSoC

in-house DwarfOS), CommType, which identifies the type of the communication library used during the HdS integration, SchedulerType to identify the type of the scheduler (preemptive, cooperative), and SchedulerAlgorithm to define the algorithm used for the tasks management by the OS (round-robin, priority based) etc.

Examples of hardware architecture parameters that annotate the system architecture model are: ResourceType which specifies the type of the hardware resource in case of a subsystem (ARM, DSP, POT) or the type of the communication protocol in case of a communication unit (DXM, SRAM, DMEM, REG, SWFIFO); AccessType which identifies whether the memory access is directly performed by the processor or by using a DMA mechanism, NetworkType, which specifies the type of network used to interconnect the different subsystems, ResourceName, which identifies the hardware resource, etc. These parameters annotating the model are used for the automatic generation of the development platform with different detail levels.

4.2 System Architecture Simulation

The Simulink model is used as a reference model for debugging the application’s algorithm. In the system architecture level, we perform a discrete-time simulation to validate the functionality of the application. For the inter- and intrasubsystem communication units, Simulink uses an abstract simulation model for each of these units based on generic Simulink I/Os.

5. VIRTUAL ARCHITECTURE

At the virtual architecture level, the application functions of the system architecture model are refined to a set of tasks code that makes use of HdS APIs including communication primitives. The following section will detail the automatic multitask code generator, the corresponding software development platform generation, and finally the resulting executable model.

5.1 Automatic Task Code Generation

The multitask code generator transforms the Simulink functions of the application into the C program code for each task. This step is very similar to the code generation performed by real-time workshop (RTW) [MathWorks Inc.; Han et al. 2006].

The generated task C code is made of two parts: computation and communication. The computation part describes the behavior of various Simulink functions that are grouped in the task, including local memory declaration. The Simulink blocks within a task are scheduled statically according to their data dependency and generated into a task C code. The communication between functions inside the tasks is translated into local memory elements. To implement the external communication between tasks, the task generator instantiates the function calls of the communication primitives from an HdS API template library, preserving the invocation order of the blocks, and maps the allocated memory space on the arguments of the functions. Before inserting the communication
primitives, the task code generator checks the data dependency between the
tasks in order to perform deadlock prevention.

The multitask C code generator can handle a large subset of predefined
Simulink blocks, such as mathematical operations (sum, multiplication, division, modulo, etc.),
logical operations (AND, OR, XOR), discrete blocks (delay, mux, demux, merge),
conditional structures (if-then-else), and repetitive structures (for-loop, while-condition-loop). The generator also supports user-defined C codes integrated in the Simulink model as S functions. For the S functions, the task code generator produces in the generated code a function call of the
user-written C function. The semantics of the argument passing are identical to those of the definition in the configuration panel of the S function-builder tool.

The generated tasks code is independent of the target processor, communication protocol and abstraction level. This can be achieved by using HdS APIs
that hide many details of the underlying implementation of the architecture
and represent the abstraction of the hardware [Sangiovanni-Vincentelli et al.
2004].

5.2 Virtual Architecture Platform Generation for Diopsis

At the virtual architecture level, the tasks code uses HdS APIs, whose implement-
ation depends on the development platform. The development platform includes all the components accessed by HdS API and the resources to implement
the required communication paths (Figure 8).

The inter-subsystem communication units are partially mapped on the mem-
ory modules (DXM, REG, SRAM, DMEM), attached as slave components to the explicit interconnection component, AMBA bus. The virtual architecture software development platform is described in SystemC language and is automatically generated from the Simulink System Architecture. Figure 8 shows a conceptual view of the virtual architecture model and the generated software
development platform in SystemC.

Based on the architecture parameters of the system architecture model,
the virtual architecture platform generator automatically generates a
SC THREAD for each task and a SC MODULE for each software subsystem
that includes the tasks. It also makes explicit the interconnection component
between the different subsystems, namely, the AMBA bus. The AMBA and the
software FIFO channels used for the intrasubsystem communication are based
on SystemC channels. The platform generator connects all the hardware compo-
nents and maps the communication buffers onto the memory resources. During
the generation, it makes use of storage and interconnect hardware components
library.

5.3 Virtual Architecture Simulation Model
To produce an executable model, the task code and software development
platform are compiled together. The resulted simulation model uses the
SystemC scheduler to activate/deactivate the execution of the different tasks.
The software tasks are SystemC threads, while the processor and memories are
SystemC modules. The AMBA and the software FIFO channels are derived from

SystemC channels. The simulation at the virtual architecture level allows validation of the generated tasks codes and the application's partitioning to avoid communication deadlock.

6. TRANSACTION-ACCURATE ARCHITECTURE

At the transaction-accurate architecture level, a software stack is built for each processor subsystem. This software stack is composed of the previously generated tasks code enriched with an initialization code, an OS, and communication library. The following section will detail the software stack generation, respectively the corresponding development platform’s generation and simulation.

6.1 Automatic HdS Integration

At the transaction-accurate level, the tasks code is integrated with an OS and communication library to compose the software stack running on each processor [Guerin et al. 2007]. The HdS software refines the communication APIs to
custom hardware-specific low-level APIs (HAL APIs) and it is responsible for task and hardware-resources management. The Hs generation is based on OS and communication libraries. Thus, the proposed approach allows easy customization for specific architectures and/or applications. In addition, an initialization main code is generated to initialize the tasks for each processor subsystem. Figure 9 illustrates an example of the resulting code.

The main file includes the declaration of the tasks. It is also responsible to initialize the tasks and the communication channels among them. The task code uses Hs API. In this example, the implementation of the communication primitive `recv_data` is based on a FIFO mechanism. If the FIFO is empty, the scheduler of OS is called. If a new task is ready, the scheduler performs a context switch, by calling the HAL `cxt_switch` API.

At this step, the HAL APIs abstract the underlying hardware architecture. Their implementation is not yet defined for the target processor, allowing keeping the software code processor independent.

### 6.2 Transaction-Accurate Platform Generation for Diopsis

At the transaction-accurate level, the software stack running on each processor makes uses of HAL APIs (Figure 10).

The corresponding software development platform that allows the execution of this software task contains the components that can be accessed by the HAL APIs. This includes interrupt controller, DMA, synchronization components, bus interfaces and other peripherals accessible through the HAL APIs.

The software-development platform at the transaction-accurate architecture level is automatically generated from the system architecture model. The platform generator builds a SystemC model of the abstract architecture. At this level, the intersubsystem communication units are fully mapped on the hardware resources. The intrasubsystem communication is fully managed by the OS by using software FIFO channels.
Based on the architecture parameters annotating the system architecture model, the platform generator details the local architecture of each subsystem, including components such as peripherals, local memories, synchronization components, or bus interfaces. Then, the transaction-accurate architecture generator maps the different communication buffers on the different memory resources and creates the interrupt lines used for the synchronization between the processor for the data exchange. It also interconnects all the hardware components. During the generation, the platform builder makes use of a library composed of the memories, peripherals, interconnect, and abstract processor components. Figure 10 shows a conceptual view of the transaction-accurate architecture model and the generated software development platform in SystemC for Diopsis.

6.3 Transaction-Accurate Architecture Model Simulation

The full hardware-software executable model is based on a cosimulation between SystemC for the hardware components, including the abstract processors, and the native execution of the software stacks [Niculescu 2002]. Each
The software stack is a SystemC thread, which creates a UNIX process for the software execution. At the beginning of the simulation, the SystemC platform launches a GNU standard debugger (gdb) UNIX process for each software stack in order to start its execution. The software stack interacts with the corresponding SystemC abstract processor module through the UNIX IPC layer. The hardware–software interface uses UNIX-shared memory for the interaction between the software and hardware. The simulation at the transaction-accurate architecture level allows validating the integration of the tasks code with the OS and the communication protocol and debug of the HdS access to the hardware resources (e.g., access to the AMBA bus, and interrupt lines assignment).

7. EXPERIMENTS
In this section, we present the results of mapping an MJPEG decoder application [Wallace 1991] on the Diopsis platform starting from a Simulink model of the application. For validation of the software components through simulation at different abstraction levels, we used a ten-frame input bit stream encoded using the QVGA (320×240) YUV 444 format.

7.1 System Architecture Modeling
We developed the MJPEG decoder Simulink model using seven S-functions. During this step, the main functions of the MJPEG decoder were isolated into separate tasks (Figure 11).

The variable-length decoding (VLD) constitutes the first task. The zigzag scan, differential pulse code modulation on DC component (DPCM), run-length decoding on AC component (RLD) and inverse quantization (IQ) are grouped into a second task. We mapped these first two tasks on the ARM processor (Figure 7). The inverse discrete cosine transform (IDCT) computation takes 68% of total execution time of the decoder and it was mapped on the DSP. The resulting decoded image is displayed on a LCD panel connected through the SPI peripheral of the POT. The system architecture in Simulink is annotated with architecture information used for the further software refinement and generation of the software-development platform. To validate the MJPEG algorithm, we simulated the model using discrete-time simulation engine. The simulation time was 35 s on PC running at 1.73 GHz, with 1-GMB RAM.

7.2 Virtual-Architecture Model Generation

From the Simulink system-architecture model, we automatically generated the C code for the tasks and the virtual architecture model to execute the resulting tasks code. Table I resumes the code and data size of the generated application code. We also performed single-task code generation from Simulink by using real-time workshop (RTW) to compare our results. The code library contains the user-defined C functions commonly used by both RTW and our tool. The application code obtained by the proposed software generation flow is more efficient in terms of code size than the code generated using RTW. However, the multitasked representation requires communication buffers. Therefore, the data size is larger than in the case of RTW, which generates only single-task code.

The functionality of the generated software code was validated by execution using the generated virtual architecture development platform. In this validation, the tasks are scheduled by the SystemC simulation engine. The simulation at the virtual architecture level allowed gathering of important early performance measurements, e.g., the bus was accessed to transfer 108,000 messages during the decoding process of the ten frames. The data transfer is performed in messages of 65 words between the processors through the SRAM (64 words for the IDCT coefficients and 1 word for the coding pattern) and 16 words between the DSP and POT via DMEM. The virtual architecture simulation took 14 s for the ten frames input bitstream and assured that the communication primitives used in the tasks code do not produce communication deadlock between the different tasks.

7.3 Transaction-Accurate Architecture Model Generation

In the next step, we generated the HdS code for each processor. The tasks code remains unchanged thanks to the HdS APIs. The implementation used an in-house tiny OS and communication library to refine HdS API primitives to HAL APIs using specific addresses. The used OS supports interrupts management, FIFO communication protocol, a round-robin scheduling policy, and boot code. The communication primitives are based on blocking message-passing interface semantic. At this level, the generated tasks are dynamically scheduled by the OS scheduler according to the availability of data for read operations or the availability of space for write operations. We also automatically generated the software development platform at the transaction-accurate architecture level to execute the software stack code. The simulation at this level allowed debugging the integration of the application tasks, OS, and communication protocols. We could debug the access of the OS functions to the hardware resources through the HAL APIs, e.g., read/write in the memory, explicit synchronization using
mailboxes, and the interrupt routine services. It also provided a more precise idea on performances that allowed some architecture experimentation as detailed in the remainder of the section. The simulation took 5 m 10 s for the ten frames input image.

7.4 Results

7.4.1 Communication Mapping Exploration. In the next step, we generated the HdS code for each processor. The tasks code remains unchanged thanks to the HdS APIs. The implementation used an in house tiny OS and communication library to refine HdS API primitives to HAL API using specific addresses.

Using transaction-accurate simulation, we conducted three experiments with different communication schemes between DSP and RISC. The results are summarized in Table I. In the first scheme, the data exchange is made only via DXM. This generated 5256 k transactions to DXM. The second communication scheme makes use of DXM and REG communication units between the processors and DMEM between DSP and POT. This generated 4608 k transactions to the DXM, 72 k to the register, and 576 k to the DMEM. The third case uses the SRAM as communication unit between processors and DMEM between DSP and POT and needs 4680 k transactions to the SRAM and 576 k to the DMEM.

We used quantitative estimators provided by Atmel Inc. for the number of clock cycles needed by ARM and DSP to access data buffers of length N words located in different memories. The DMA engine of the DSP needs $14 + (N - 1)$ cycles for DXM read, $10 + (N - 1)$ for DXM write, $5 + (N - 1)$ for SRAM read, and $8 + (N - 1)$ for SRAM write. A data movement between REG and SRAM, driven by the DSP core, costs $N/4$ cycles plus a movement to/from the SRAM, driven by the DMA engine. The ARM processor is not natively equipped with a DMA engine. The cost of ARM isolated access is $11*N$ for DXM read and $8*N$ for DXM write. Forcing the compiler to use the assembler instruction, which moves blocks of eight registers, the cost of burst can be reduced to $11*(N/8) + N$ for DXM read and $2*N$ for DXM write. On the Diopsis tile, the ARM processor runs at a clock frequency, which is double the AMBA bus used as unit of measure. This factor 2 can be taken in account in the estimate of time of ARM access to SRAM. The DSP data memory can be accessed by ARM in $6*(N/8) + N$ cycles for write and $8*N$ cycles for read.

The performance estimation results are summarized in Table II. The overall number of cycles required for communication using AMBA burst mode are: approximately 9000 k when all the data transfer is made via DXM, 8316 k in the second case using REG, DXM, and DMEM, and 4392 k in the third case.
Table III. Productivity

<table>
<thead>
<tr>
<th>Software abstraction level</th>
<th>Lines of code</th>
<th>Manual est. time (hr)</th>
<th>Generation time (s)</th>
<th>Productivity gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual architecture</td>
<td>405</td>
<td>39</td>
<td>0.825</td>
<td>~170,000</td>
</tr>
<tr>
<td>Transaction-accurate architecture</td>
<td>401</td>
<td>52</td>
<td>1.007</td>
<td>~180,000</td>
</tr>
</tbody>
</table>

using SRAM and DMEM. Thus, if the software code makes use of the existing hardware resources, we obtain an improvement in communication performance of 11% in the second case, and 55% in the third case. The communication protocol is specified in the initial Simulink model by annotating the communication units. The platform generator tools then automatically refines the architecture based on the Simulink annotation. Therefore, it is quite simple to experiment with different communication schemes.

7.4.2 Productivity Gain. Table III presents the productivity gain obtained with our approach. It shows the number of generated lines of code for the software and the development platform, without the libraries, for the generation time of virtual architecture and transaction-accurate architecture levels, respectively their generation time.

The results were captured for the model with the third mixed-communication mapping scheme (DXM, SRAM and DMEM). For the manual estimation time, we considered an optimistic figure of 20 lines of code per person by hour. Based on automatic tools, the generation of the software stack and the validation platforms from the Simulink model takes only few seconds for the Motion JPEG decoder application.

7.4.3 FPGA Platform-Based Emulation. In order to validate the correctness and efficiency of the proposed software design flow, we ran the generated software stack of the MJPEG application on a Diopsis emulation platform using Xilinx Virtex-II XC2V8000 FPGA. Since the HAL library to access the resources of the emulation platform was not yet fully available, we mapped the four tasks of the application onto the ARM9 processor. We then generated and incrementally validated the software stack. For the tasks management, we used the DwarfOS which implements services, such as round-robin scheduling, specific context switch for the ARM9 processor, and software FIFO channels for the communication between the tasks. We used a Multi-Ice GDB debugger server in order to load the final software binary image on the local SDRAM memory. The FPGA platform-based emulation ensures the reliability of the generated code’s functionality. Future work will focus on running the generated application code on the FPGA-based emulation platform using multiprocessors and different OS and communication libraries.

7.5 Results Analysis

The proposed software design flow allowed to execute the generated software stack binary on a FPGA-based emulation platform. Several mappings of the communication onto the hardware resources can be experimented with using...
simple annotations of the initial Simulink model. The automatic generation of the software and platforms takes few seconds, producing a significant gain in productivity. The structuring of the software stack into layers allows a huge flexibility in terms of software components reuse (OS and communication protocol), portability to other platforms (HAL), and software code preservation during the refinement process. The different platforms allow iterative software debugging of different software stacks. The system architecture simulation allows debugging of the application's algorithm, the virtual architecture allows debugging the tasks code and communication scheduling to avoid deadlock, the transaction-accurate architecture allows debugging the low-level software code and communication protocol, and the virtual prototype, which allows final binary debugging.

8. CONCLUSION
In this article, we presented a platform-based software design flow allowing efficient software code generation and validation for architectures including heterogeneous MPSoC with specific I/O. The paper presented a high-level programming environment for a complex heterogeneous multimedia platform, namely, a reduced Diopsis tile, including a high-performance DSP and ARM9 microcontroller. The abstraction of the hardware architecture at different abstraction levels allowed an incremental validation of the different software components. Based on automatic tools, it increases the design productivity. The effectiveness of this software design flow has been illustrated on the MJPEG application. Future works consist of executing the generated software stack on both ARM9 and DSP processors using the emulation platform and apply the approach for different type of tiles with multiple DSPs.

REFERENCES


MAGICIV VLIW DSP AND DIOPISS. http://www.atmelroma.it.


Platform-Based Software Design Flow for Heterogeneous MPSoC


MODEL DRIVEN ARCHITECTURE. http://www.omg.org/mda/


Received August 2007; accepted March 2008