Chapter 4

HARDWARE ABSTRACTION LAYER

Introduction and Overview

Katalin Popovici and Ahmed Jerraya

Abstract Embedded software is playing an increasing role in heterogeneous Multi-Processor System-on-Chip (MPSoC) architectures due to its high complexity. In order to reduce the long and fastidious design process, embedded software needs to be reused over several MPSoCs. Thus, software portability becomes a key challenge.

In this chapter, we present a clear separation between the hardware independent and the hardware dependent software layers, through adopting a multi-layered organization of the software stack. We introduce a component based software design flow, which allows the gradual generation and validation of the various software layers to obtain the final software stack. Then, by changing the Hardware Abstraction Layer (HAL), the software stack can be executed on different MPSoC architectures. The HAL represents the lowest software layer, which totally depends on the target architecture. The HAL abstraction, through the use of well defined HAL APIs, makes easier the software portability and enables flexibility. The paper shows that the HAL APIs allow early software development before the hardware architecture is available, but also architecture exploration. The proposed methodology is applied to design the software stack for the Motion JPEG multimedia application and to execute it on diverse processors by changing the HAL and preserving the HAL APIs.

Keywords: MPSoC, Software Design, Software Validation, HAL, HAL Abstraction

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,
© Springer Science + Business Media B.V. 2009

4.1 Introduction

Current Multi-Processor System-on-Chip (MPSoC) architectures integrate a large number of processing subsystems on the same chip [Wol06]. The processing subsystems usually contain different types of programmable processing units or CPUs, depending on the target application domain. Thus, DSP (Digital Signal Processor) is mostly used for signal processing applications; microcontrollers are more common for control-intensive applications, while ASIP (Application Specific Instruction Set Processors) represent stored-memory CPUs whose architectures are tailored for a particular set of applications.

As more and more heterogeneous processors and hardware components are integrated together, the design and validation of the software running on these complex heterogeneous architectures become a major bottleneck, because the software is even more complex [Tur05]. The key issue of the software design for MPSoC is to produce efficient software code with strong time to market constraints. Producing efficient code requires that the software takes into account the capabilities of the target architecture. This generally requires a long and fastidious software debug cycle. The classic way to accelerate the software design process relies on automatic software code generation from high level programming models that abstract the architecture, but this approach produces a huge expense in the efficiency of the generated code.

The software is generally organized into several stacks made of two layers: application and Hardware-dependent Software (HdS). The validation and debug of the Hardware dependent Software (HdS) is the main bottleneck in MPSoC design, because each processor subsystem requires specific HdS implementation to be efficient [JW05]. Current research studies proved that the HdS debug represents 78% of the global system total debugging time of an MPSoC design cycle [YYS+04]. This may be due to incorrect configuration or access to the hardware architecture, e.g. a wrong configuration of the memory mapping for the interrupt control registers.

Besides software complexity, portability becomes a major issue to decrease the overall design and validation time, because it allows software reuse over several SoCs. Portability enables execution of the same software on different hardware architectures. In terms of design reuse, the portability enables reuse of the software designed for a particular MPSoC architecture to another. Thus, portability reduces the design efforts, otherwise necessary to adapt the software for the new hardware architecture.

In order to reduce its complexity and enable easy software portability, we propose structuring the HdS into three software components: a real time operating system (RTOS) aimed to schedule the different application tasks, a specific communication library to implement the communication protocol and

the hardware abstraction layer (HAL) to access the hardware resources. The HAL represents the thin software layer that totally depends on the underlying target architecture. Structuring the HdS in these well defined layers accessible through application programming interface (API) is essential to support software flexibility and portability on different hardware platforms.

Traditional MPSoC design flow starts with the application partitioning into hardware and software tasks that are mapped on processing elements. The definition of generic HAL APIs for the target application domain makes it possible to start designing the software before the hardware is complete, thus enabling concurrent hardware and software design. In fact, the software design is structured in two main phases. The first phase is the hardware independent software design (application tasks, OS), which may start after the definition of the HAL APIs. The second phase represents the HAL design and integration into the software stack. Figure 4.1 shows these steps. Separating the hardware dependent and hardware independent software designs also makes the architecture exploration easier, since the hardware independent software can be reused over several architectures. Only the HAL must be altered in case of different architectures.

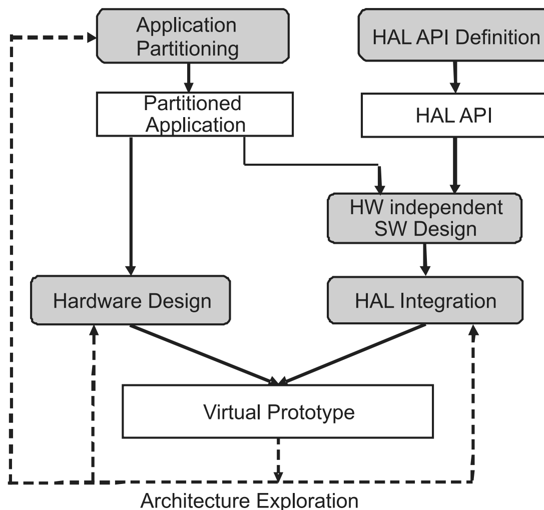

Figure 4.1. Hardware-software design flow.

In this chapter, we give the basic definitions for the different software components, and we emphasize the importance of the HAL layer in the context of MPSoC to provide software portability across different hardware platforms. We present a component based software design flow that allows generation and validation of the different software elements. In order to allow the software reuse, we define the abstraction of the HAL through APIs and the validation

of these HAL APIs. The proposed methodology is applied to design and to adapt the software stack to different processors for the Motion JPEG decoder application.

The chapter is composed of seven sections. Section 4.1 gives a short introduction to present the context of MPSoC software design. Section 4.2 defines the organization of the software stack into different components. Section 4.3 discusses the role of HAL in the software stack and explains how to achieve software portability through abstraction of the HAL. Section 4.4 enumerates several existing commercial HAL. Section 4.5 summarizes the main steps of the proposed software design and validation flow. Section 4.6 presents the HAL execution and simulation using specific software development platforms. Section 4.7 addresses the experimental results, followed by conclusion.

4.2 Software Stack

This section defines the software stack running on the different processing subsystems and presents its layered organization in several software components.

4.2.1 Software Stack Definition

The software stack represents the software running on a processing subsystem. In heterogeneous MPSoC architectures, each processing subsystem executes a software stack. The software stack is made of two layers: the application tasks code and the hardware dependent software (HdS). The HdS layer includes three software components: the Operating System (OS), specific I/O communication software and the Hardware Abstraction Layer (HAL). The HdS is responsible for providing application and architecture specific services, i.e. scheduling the application tasks, communication between the different tasks, external communication with other processing subsystems, or hardware resources management and control. The following paragraphs detail the software stack organization, including all these different components.

4.2.2 Software Components

The software stack is structured in different software layers that provide specific services. Figure 4.2 illustrates the software stack organization in two layers: application layer and HdS (Hardware-dependent Software) layer. In the first section we present the application layer. Then, the HdS layer will be defined.

Application Layer. The application layer contains the software code for applications such as multimedia (e.g. MP3, MPEG4 and JPEG 2000) or communications (e.g. protocol stack and physical layers). It may be a multi-tasking

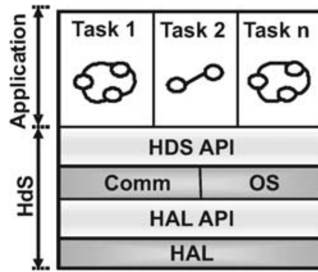

Figure 4.2. Software stack organization.

description or a single task function of the application targeted to be executed on a particular processor subsystem.

A *task* or *thread* is a lightweight process that runs sequentially and has its own program counter, register set and stack to keep track of where it is. In this chapter, the terms task and thread are used as interchangeable terms. Multiple tasks can be executed in parallel by a single CPU (single-core processor subsystem) or by multiple CPUs of the same type grouped in a single subsystem (multi-core processor subsystem). The tasks may share the same resources of the architecture, such as processors, I/O components and memories. On a single processor core node, the multithreading generally occurs by time slicing, wherein a single processor switches execution between different threads. In this case, the task processing is not literally simultaneous, as the single processor is doing only one thing at a time. On a multi-core processor subsystem, threading can be achieved via multiprocessing, wherein different threads can run literally simultaneously on different processors inside the processor node [Tan95].

The application layer consists of a set of tasks that makes use of Application Programming Interface (API) to abstract the underlying HdS software layer. These APIs correspond to the HdS APIs.

HdS Layer. The HdS layer represents the software layer which is directly in contact with, or significantly affected by, the hardware that it executes on, or can directly influence the behavior of the underlying hardware architecture [Pos03]. The HdS integrates all the software that is directly depending on the underlying hardware, such as hardware drivers or boot strategy. It also provides services for resources management and sharing, such as scheduling the application tasks on top of the available processing elements, inter-task communication, external communication, and all other kinds of resources management and control. The federative HdS term underlines 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 [JBP06].

To decrease the complexity of the HdS debug, the HdS is organized into three software components: operating system (OS), communication management (Comm) and hardware abstraction layer (HAL). Figure 4.2 illustrates these software components.

Operating System. The operating system (OS) is the software component that manages the sharing of the resources of the architecture. It is responsible for the initialization and management of the application tasks and communication between them. It provides services, such as tasks scheduling, context switch, synchronization and interrupt management. In the following, we define each of these basic OS services.

The tasks scheduling service of the OS usually follows a specific algorithm, called scheduling algorithm. Finding the optimal algorithm for the tasks scheduling represents a NP-complete problem [VBL05]. There are different categories of scheduling algorithms. The classic criteria are hard real-time versus soft real-time or non real-time; preemptive versus cooperative; dynamic versus static; centralized versus distributed.

Contrary to non real-time, the real-time scheduler must guarantee the execution of a task in a certain period of time. Hard real-time must guarantee that all the deadlines are met.

Preemptive scheduling allows a task to be suspended temporally by the OS, for example when a higher-priority task arrives, resuming later when no higher-priority tasks are available to run. This is associated with time-sharing between the tasks. Examples of preemptive scheduling algorithms are: round robin, shortest-remaining-time or rate-monotonic schedulers. The cooperative or non-preemptive scheduling algorithm runs each task to its completion. In this case, the OS waits for a task to surrender control. This is usually associated with event-driven operating systems. Examples of non-preemptive algorithm are the shortest-job-next or highest-response-ratio-next.

With static algorithms, the scheduling decisions (preemptive or non-preemptive) are made before execution. Contrary to static algorithms, the dynamic schedulers make their scheduling decisions during the execution.

The implementation of the scheduler may be centralized or distributed. In case of a centralized scheduler implementation, the scheduler controls all the task execution ordering and communication transactions. In case of a distributed scheduler implementation, the scheduler distributes the control decision to the local task schedulers corresponding to each processor [CYC+05].

When a task is ready for execution and it is selected by the scheduler of OS according to the scheduler algorithm, the OS is also responsible to perform the context switch between the currently running task and the new task. The context switch represents the process of storing and loading the state of the CPU which runs the tasks, in order to share the available hardware resources

between different tasks. The state of the current task, including registers, is saved, so that in case the scheduler gets back for execution the first task, it can restore its state and continue normally.

In order to ensure a correct runtime and communication order between the different tasks running on parallel, synchronization is required. The tasks can synchronize by using semaphores or by sending/receiving synchronization signals (events) each other. The mutex is a binary semaphore which ensures mutual exclusion on a shared resource, such as a buffer shared by two threads, by locking and unlocking it, whenever the resource is accessed by a task [TW97].

The interrupt handler is another OS service used for the interrupts management. There are two types of processor interrupts: hardware and software. A hardware interrupt causes the processor to save its state of execution via a context switch, and begins the execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set of the processor, which cause a context switch to an interrupt handler similar to a hardware interrupt. The interrupts represent a way to avoid wasting the processor's execution time in polling loops waiting for external events. Polling means when the processor waits and monitors a device until the device is ready for an I/O operation.

Examples of commercial OS are the eCos [eCos], FreeRTOS [FRTOS], LynxOS [LOS], VxWorks [VxW], WindowsCE [WCE] or μITRON [uIT].

Communication Software Component. The second software component of the HdS layer constitutes the communication component, which is responsible for managing the I/O operations and, more generally, the interaction with the hardware components and the other subsystems. The communication component implements the different communication primitives used inside a task to exchange data between the tasks running on the same processor or between the tasks running on different processors. It may include different communication protocols, such as FIFO (first-in-first-out) implemented in software, or communication using dedicated hardware components. If the communication requires access to the hardware resources, the communication component invokes primitives that implement this kind of low level access. These function calls are done in form of the HAL APIs.

The HAL APIs allow for the OS and communication components to access the third component of the software stack, that is the HAL layer.

Hardware Abstraction Layer. Low level details about how to access the resources are specified in the Hardware Abstraction Layer (HAL) [YJ03]. The HAL is a thin software layer which totally depends on the type of processor that will execute the software stack, but also depends on the hardware resources interacting with the processor. The HAL includes the device drivers to implement the interface for the communication with the device. This includes the

implementation of the drivers for the I/O operations or other peripherals. The HAL is also responsible for processor specific implementations, such as loading the main function executed by an OS, more precisely the boot code, or implementation of the *load* and *restore* CPU registers during a context switch between two tasks, but also software codes for configuration and access to the various hardware devices, e.g. MMU (Memory Management Unit), timer, interrupt enabling/disabling etc. More details about the HAL will be given in the following sections.

The structured representation and the organization of the software stack into several layers (application tasks, OS, communication and HAL), as previously described, have two main advantages: flexibility in terms of software components reuse by changing the OS or the communication software components, and portability to other processor subsystems by changing the HAL software layer.

The following paragraphs give the definition of the HAL software component and highlight its role in enabling software portability. Thereafter, the main steps required by the design and validation of these different software components are explained in detail.

4.3 Hardware Abstraction Layer

In this section, the definition of the HAL is given. This is followed by the HAL abstraction through well defined APIs to enable software portability across various hardware platforms.

4.3.1 Definition and Examples of HAL

The HAL is defined in [eCos] as all the software that is directly dependent on the underlying hardware. If the hardware architecture is changed, changes also have to be made to the HAL. The HAL can be implemented in the assembly language recognized by the processor or in specific C code. In fact, the HAL includes two types of software code:

- Processor specific software code, such as context switch, boot code or code for enabling and disabling the interrupt vectors.
- Device drivers, which represents the software code for configuration and access to hardware resources, such as MMU (Memory Management Unit), system timer, on-chip bus, bus bridge, I/O devices, resource management, such as tracking system resource usage (check battery status) or power management (set processor speed).

The HAL offers a set of services to the upper level OS and communication libraries that grant them access to the hardware platform. Generally, the HAL provides the following kinds of services:

- Integration with an ANSI C standard library to provide the familiar C standard library functions, such as *printf()*, *fopen()*, *fwrite()*, *exit()*, *abs()*, *atof()*, etc. An example of such a library is the newlib library, which represents an open-source implementation of the C standard library .newlib for the use on embedded systems [newl].
- Device drivers to provide access to each device of the hardware platform.
- The HAL API to provide a consistent interface to HAL services, such as device access, interrupts handling and debug facilities.
- System initialization to perform the initialization of the tasks for the processor before the execution of the *main()* function of the application.
- Device initialization to instantiate and initialize each device in the hardware platform before the execution of the *main()* function of the application.

The device drivers, that are part of the HAL, are the interface between a hardware resource and the application or OS. Usually, the drivers are hardware dependent and OS specific. Typical device drivers provide access to the following classes of hardware components:

- Character-mode devices, which represent hardware peripherals that send and/or receive characters serially, such as an UART (Universal Asynchronous Receiver/Transmitter) device.
- Timer devices, which are hardware peripherals that count clock ticks and generate periodic interrupt requests.
- File subsystems, which provide a mechanism for accessing files stored within physical devices. Depending on the internal implementation, the file subsystem driver may access the underlying devices either directly or by using a separate device driver. For example, a flash file subsystem driver may access a flash memory by using dedicated HAL APIs for the flash memory devices.
- Ethernet devices to provide access to an Ethernet connection for a networking stack, such as the NicheStack TCP/IP stack [NS].
- DMA devices that are peripherals that perform bulk data transactions from a data source to destination. Sources and destinations can be memory or another hardware device, such as an Ethernet connection.
- Flash memory devices, which are nonvolatile memory devices that use a special programming protocol to store data.

Besides the implementation of the device drivers, the HAL includes processor specific code as well, such as the implementation of the context switch or interrupt handling.

Figure 4.3 presents an example of processor specific HAL code, which performs a context switch between two application tasks running on an ARM7 processor. This example of HAL software code uses the assembly language specific to the ARM7 processor in order to access some particular processor registers (R0-R14, PC-Program Counter). The context switch needs two basic operations to be performed: store the status of the processor registers used by the current task and load the status of the registers of the new task.

```
__ctx_switch                  ; r0 old stack pointer, r1 new stack pointer
STMIA r0!,{r0-r14}            ; save the registers of current task
LDMIA r1!,{r0-r14}            ; restore the registers of new task
SUB pc,lr,#0                  ; return
END
```

Figure 4.3. HAL implementation for the context switch on the ARM7 processor.

Figure 4.4 illustrates another example of low level software code implementation that enables and disables the IRQ interrupts for the ARM7 processor. The interrupts are enabled and disabled by reading the CPSR (Current Program Status Registers) flags and updating bit 7 corresponding to bit I (IRQ Interrupt).

```
__inline void enable_IRQ(void)                    //HAL API
{
          int tmp;
          __asm
          {
                    MRS tmp, CPSR
                    BIC tmp, tmp,#0x80
                    MSR CPSR_c, tmp
          }
}

__inline void disable_IRQ(void)                   //HAL API
{
          int tmp;
          __asm
          {
                    MRS tmp, CPSR
                    ORR tmp, tmp, #0x80
                    MSR CPSR_c, tmp
          }
}
```

Figure 4.4. HAL implementation for enabling and disabling ARM interrupts.

4.3.2 Software Portability Based on HAL API

In the context of software design for MPSoC, software portability becomes a key issue. Portability enables execution of the same software on different hardware architectures. In terms of design reuse, the portability enables reuse of the software designed for a particular MPSoC architecture to another. Thus, portability reduces the design efforts, otherwise necessary to adapt the software for the new hardware architecture. The portability also eases the exchange of the software code and architecture exploration, e.g. trying different types of processors to find an optimal target processor.

As it was described in the previous paragraphs, the structural organization of the software stack is made of several layers separated by well defined APIs. The lowest level software component represents the HAL layer which is a totally hardware architecture dependent layer. The OS and communication software components make use of HAL APIs. Thus, without the implementation of the HAL APIs for the target processors, the software code still remains processor independent.

The HAL APIs gives to the operating system, communication and application software an abstraction of the hardware-dependent HAL, e.g. data types like the integer "int" data type in the standard C programming language, which has different bit size depending on the processor type. Furthermore, the HAL APIs ease OS porting on new hardware architecture. The HAL APIs can be classified in the following categories [eCos]:

- Kernel HAL APIs, such as task context management APIs (e.g. context creation, delete or context switch APIs, task initialization), stack pointer and program counter management APIs (e.g. *get/set_IP()*, *get/set_SP()*) or processor mode change APIs (e.g. *enable_kernel/ user_mode()*).

- Interrupt management APIs, e.g. APIs which enable/disable interrupt request from an interrupt source (e.g. *vector_enable/ disable(vector_id)*), configure interrupt vector (e.g. *vector_configure(vector_id, level, up)*), mask/unmask interrupt for a processor (e.g. *interrupt_enable/disable()*), the implementation of the interrupt routine services (e.g. *interrupt_attach/ detach(vector_id, isr)*) or HAL APIs that acknowledge to the interrupt source that the interrupt request has been processed (e.g. *clear_interrupt(vector_id)*).

- I/O HAL APIs, which configure the I/O devices and allows their access. For example, to configure a MMU device, the following I/O HAL APIs may be required: APIs for page management (e.g. *enable/disable_paging()*), address translation (e.g. *virtual_to_physical()*), TLB (Translation Lookaside Buffer) management, such as set TLB entry (e.g. *TLB_add()*) or get TLB entry virtual/physical page frame (e.g.

get_TLB_entry()). Other I/O HAL API examples can be considered the APIs for cache memory management, such as *Instruction/Data_Cache_Enable/Disable()*.

- Resource management APIs, such as APIs for power management (e.g. check battery status, set CPU clock frequency) or APIs to configure the timer (e.g. *set/reset_timer()*, *wait_cpu_cyle()*).

- Design time HAL APIs, which facilitates the software design process, or more precisely, the simulation. Example of such kind of API is the *consume_cpu_cyle()* to simulate the advance of the software execution time.

The HAL APIs are used by the upper software layers, like OS and communication components. Figure 4.5 shows an example of utilization of the HAL API in a fragment of code inside the OS scheduler. Thus, the OS scheduler searches for a new task in status ready for execution. If there is a new ready task, the scheduler performs a context switch, by calling the HAL API *__cxt_switch(. . .)*. During the context switch, the OS saves the status and registers (program counter, stack pointer, etc.) of the processor running the current task and loads those of the new task.

```
void __schedule (void){
   int old_tid = cur_tid;
   cur_tid = get_new_tid();          //get new task ready for execution

   __ctx_switch (old_tid,cur_tid);   //context switch HAL API
  …
```

Figure 4.5. Example of HAL API function call inside the OS scheduler.

4.4 Existing Commercial HAL

In the following section, we give several examples of existing commercial HAL that are used in both academic and semiconductor industry areas.

Even if the HAL represents an abstraction of the hardware architecture, since it has been mostly used by OS vendors and each OS vendor defines its own HAL, most of the existing HAL is OS dependent. In case of an OS dependent HAL, the HAL is often called board support package (BSP). In fact, the BSP implements a specific support code for a given hardware platform or board, corresponding to a given OS. The BSP also includes a boot loader, which contains a minimal device support to load the OS and device drivers for all the devices on the hardware board.

The embedded version of the Windows OS, namely Windows CE, provides BSP for many standard development platforms that support several microprocessors family (ARM, x86, MIPS) [WCE]. The BSP contains an OEM (Original Equipment Manufacturer) adaptation layer (OAL), which includes a boot loader for initializing and customizing the hardware platform, device drivers, and a corresponding set of configuration files.

The VxWorks OS offers BSP for a wide range of MPSoC architectures, which may incorporate ARM, DSP, MIPS, PowerPC, SPARC, XScale and other processors family [VxW]. In eCos, a set of well-defined HAL APIs are presented [eCos]. However, there's no clear difference between HAL and device driver. Examples of HAL APIs used by eCos are:

- Thread context initialization:
  HAL_THREAD_INIT_CONTEXT()
- Thread context switching:
  HAL_THREAD_SWITCH_CONTEXT()
- Breakpoint support:
  HAL_BREAKPOINT()
- GDB support:
  HAL_SET_GDB_REGISTERS(), HAL_GET_GDB_REGISTERS()
- Interrupt state control:
  HAL_RESTORE_INTERRUPTS(), HAL_ENABLE_INTERRUPTS(),
  HAL_DISABLE_INTERRUPTS()
- Interrupt controller management:
  HAL_INTERRUPT_MASK()
- Clock control:
  HAL_CLOCK_INITIALIZE(), HAL_CLOCK_RESET(),
  HAL_CLOCK_READ()
- Register read/write:
  HAL_READ_XXX(), HAL_READ_VECTOR_XXX(),
  HAL_WRITE_XXX(), and HAL_WRITE_VECTOR_XXX()
- Control the dimensions of the instruction and data caches:
  HAL_XCACHE_SIZE(), HAL_XCACHE_LINE_SIZE()

In the software development environment for the Nios II processor provided by Altera [HAL], the HAL serves as a device driver package, providing a consistent interface to the system peripherals, such as timers, Ethernet MAC and I/O peripherals.

In Real-Time Linux a HAL, called Real-Time HAL (RTHAL), is defined to give an abstraction of the interrupt mechanism to the Linux kernel [RTL]. It consists of three APIs for disabling and enabling interrupts and return from the interrupt.

An example of HAL that does not depend on the targeted OS is the a386 library [A386]. The a386 represents a C library which offers an abstraction of the Intel 386 processor architecture. The functions of the library correspond to privileged processor instructions and access to the hardware. The library serves as a minimal hardware abstraction layer for the OS. Later, the library is ported on ARM and SPARC processors.

4.5 Overview of the Software Design and Validation Flow

This section gives an overview of the software design and validation flow. The overall flow is illustrated in Fig. 4.6.

The software design flow has three main steps: application software generation, software stack construction, and performance and software validation through simulation on a development platform [PJ07].

The software design flow starts with a manual design step to build the high level application model that captures the grouping of the application functions

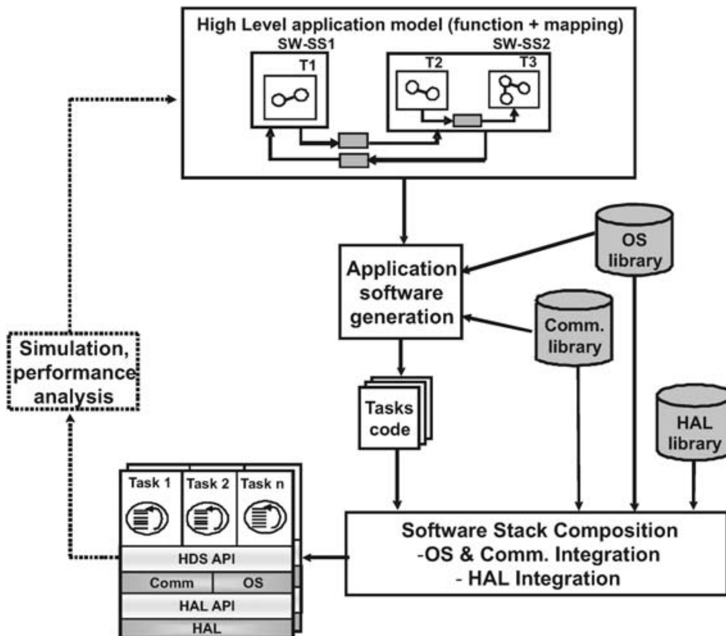

Figure 4.6. Software design and validation flow.

into tasks, and the tasks into processor subsystems. Thus, it combines the application behavior with the architecture specification, and the application mapping information onto the architecture. The result of this step represents a combined architecture/application model. This high level application model may use explicit communication units to abstract the intra-subsystem communication (communication between the different processor subsystems) and inter-subsystem communication (communication between the tasks mapped onto the same processing subsystem).

4.5.1 High Level Application Model

The high level application model represents a functional description of the application annotated with the application mapping information on the target architecture. We use Simulink environment [Math] to capture this representation. We use a specific writing style and annotation to capture the architecture details and the mapping of the communication and computation. At this level, the software is made of a set of functions grouped into tasks and the tasks grouped into software subsystems. The communication between functions, tasks and subsystems make use of abstract communication links to represent logical communication, e.g. standard Simulink links or explicit communication units that correspond to specific communication paths of the target platform. The links and units are annotated with communication mapping information. The simulation at the system architecture level allows validating the application's functionality. The hardware-software interfaces are fully abstracted. This model captures both the application and the architecture in addition to the computation and communication mapping.

4.5.2 Application Software Generation

During the application software generation, the Simulink application functions are transformed into behaviorally equivalent C code for each task. This step is similar with the code generation provided by Real Time Workshop, but the generated code uses an optimized buffer memory [Han06 +].

The generated code is made of two parts: computation and communication. The computation part represents the C behavior of the application functions, while the communication part involves high level communication primitives, such as *send(...)/recv(...)* or *channel_write(...)/ channel_read(...)*. The implementation of these APIs relies on the underlying OS and communication libraries.

4.5.3 Software Stack Composition

During the software stack composition, the previously generated application tasks code are compiled and linked together with an OS, communication

and HAL library [GPY+07]. The OS library contains the components that implement several OS services, such as scheduling, interrupt routine services, tasks management (create/kill/exit). The communication library contains the implementation of the high level communication primitives, e.g. MPI (Message Passing Interface) primitives [MPI], the TTL communication primitives [vdW04+] or YAPI communication APIs [KSW+00]. The implementation of these communication primitives can be blocking or non-blocking. The HAL library contains the implementation of the low level hardware access primitives, e.g. context switch primitives, enable/disable interrupts, boot code or specific DMA configuration primitives. The software stack composition is performed in two main steps:

- OS and communication software components integration
- HAL integration

The result of each of these steps has to be validated in order to verify the application execution on the target hardware architecture, as it will be detailed in the next section.

4.5.4 Software Validation

The software validation allows verifying the execution of the software with explicit hardware-software interaction. Traditional software development strategies make use of the concept of software development platform to debug the software before the hardware architecture is ready.

As illustrated in Fig. 4.7, the software development and validation platform is an abstract model of the architecture in form of a run-time library or simu-

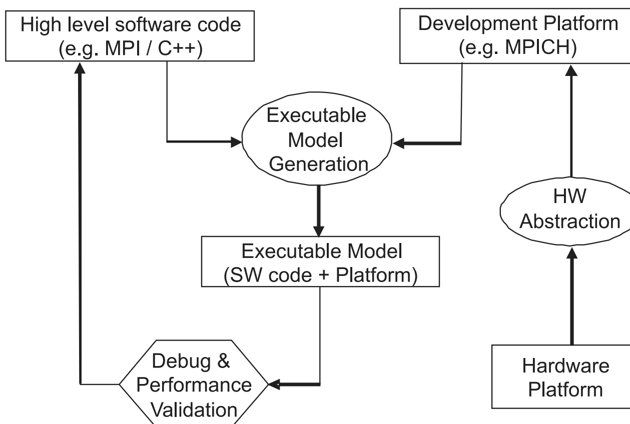

Figure 4.7. Software development and validation platform.

lator aimed to execute the software (e.g. Instruction Set Simulator). The combination of this platform with the software code produces an executable model that emulates the execution of the final system including hardware and software architecture. Generic software development platforms have been designed to fully abstract the hardware-software interfaces, i.e. MPITCH is a run-time execution environment designed to execute parallel software code written using MPI [MPI].

In this chapter, we use software development platforms implemented in SystemC TLM [GLMS02] in order to execute and debug the software code [PGR+07].

Depending on the software component to be validated (application tasks code, tasks code execution upon an OS, HAL integration in the software stack), the SystemC platform may model only a subset of hardware components, more precisely those components that are required for the software validation. The rest of the hardware components, which are not relevant for the software validation, are abstracted. For example, the debug of the application tasks code does not need explicit implementation of the synchronization protocol between the processors, such as mailboxes, semaphores or mutexes, while the debug of the integration of the tasks code with the OS requires such kind of detail in the SystemC platform.

The debug is performed using standard debuggers such as GNU debuggers or tracing SystemC waveforms during the simulation. The software validation is an iterative process because the different software components need different detail levels.

4.6 HAL Execution and Simulation Using Software Development Platforms

In order to explore and reuse the validated software components for better performance achievement, by executing them on various hardware architectures, the HAL layer plays a key role to guarantee software portability. Thus, the following sections will focus on the HAL execution on a virtual prototype using Instruction Set Simulators (ISS), and HAL APIs simulation using a transaction accurate SystemC development platform.

4.6.1 HAL Execution on Virtual Prototype

The integration of the HAL layer into the software stack needs to be validated for functional verification purpose. In order to validate such kind of HAL code, there are two possible execution techniques of HAL:

1. direct loading the software code onto the processor's program memory and execute it on a real chip or an equivalent FPGA-based emulation board;

2. using a software development platform that models the target architecture and incorporates Instruction Set Simulators (ISS) for the processors.

In this chapter, we detail the HAL execution using a SystemC development platform that combines ISS for the software execution and SystemC for the hardware simulation. This platform is also known as virtual prototype [HYL+06] and the execution model corresponds to classical hardware-software cosimulation models with ISS [Row94] [SG00].

The integration of instruction set simulators for the software execution on specific processors with hardware simulators of the architecture behavior is largely used in MPSoC domain. By using ISS, this approach allows simulating a detailed hardware-software interaction, including the HAL of the software stack. For performance verification, the timing information can be measured instead of estimated.

The execution model of the virtual prototype resides on a cosimulation between the software stack simulator and the hardware simulator [NYBJ02]. Two types of simulators are combined: ISS for simulating the programmable components running the software and SystemC for the dedicated hardware part [EPTP07].

The hardware-software simulation is driven by SystemC. The SystemC initializes the processor SystemC modules that encapsulate the ISS. During the simulation, the ISS features a simulation loop which fetches, decodes and executes instructions one after another. The ISS is developed as sequential software running on a single processor. The simulation performed at this level is cycle accurate. The simulation of the virtual prototype allows validating the HAL integration into the final software stack.

Figure 4.8 shows the execution model of an architecture made of two processors, ARM7 and XTENSA. The model contains two ISS to execute the binary

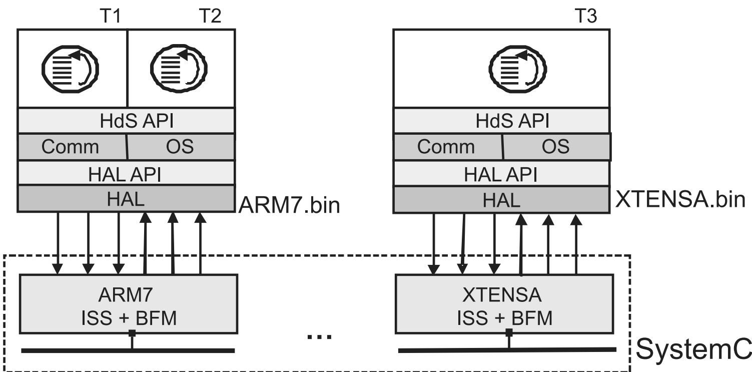

Figure 4.8. Virtual prototype execution model.

codes, corresponding to the ARM7, respectively XTENSA processors. The rest of the architecture components are cycle accurate SystemC components modeled at TLM with execution timing information. The two software stacks that are executed by the two processors include the application tasks code, communication and operating system layer and the processor specific HAL.

4.6.2 HAL Simulation on Transaction Accurate SystemC Platforms

Instead of executing the HAL on the virtual prototype, as it was previously described, the HAL can be simulated using the APIs provided by the OS running on the host machine. In this manner, the HAL APIs are executed natively on the host machine, thus providing a simulation model of the OS and the inter-processor communication scheme [BYJ04].

For example, the implementation of the *ctx_switch (old_tid, cur_tid)* HAL API, which performs a context switch between two tasks, relies on the APIs provided by the OS running on the host machine (Windows, Linux, UNIX, etc.). Figure 4.9 exemplifies the implementation of the context switch on a host machine running Linux OS, which makes use of *sigsetjmp* and *siglongjmp* APIs to save and switch the context of a task.

```
void __ctx_switch(int old_tid, int new_tid)
{
          sigjmp_buf old_buf, new_buf;

          old_buf = task[old_tid].buf;
          new_buf = task[new_tid].buf;

          if(!sigsetjmp(old_buf, 1))                              //LINUX APIs
                              siglongjmp(new_buf, 1);
}
```

Figure 4.9. Simulation of the __ctx_switch() HAL API.

Using this kind of HAL simulation model, the software stack still remains processor independent. Therefore, by abstracting the HAL through the use of HAL APIs, the application tasks code, OS and communication software components can be migrated between various processors. In this case, the only requirement is that those processors need to support the implementation of the HAL APIs, thus allowing software portability.

In order to verify the hardware-software interface, the HAL APIs are required to be executed upon a development platform with detailed hardware-software interaction. In the following, we present the execution model that allows the HAL native simulation and makes use of a transaction accurate hardware platform implemented in SystemC. The hardware platform contains

all the hardware resources that are required for the HAL APIs native execution and validation.

The combination of the transaction accurate platform with the software stack based on HAL APIs results in an executable model. The full hardware-software executable model is based on a co-simulation between SystemC for the hardware components including the abstract execution models of the processors, and the native execution of the software stacks [NYBJ02].

Each software stack is a SystemC thread which creates a Linux process for the software execution. At the beginning of the simulation, the SystemC platform launches a GNU standard debugger (gdb) Linux process for each software stack in order to start its execution. The software stack interacts with the corresponding SystemC abstract processor module through the Linux IPC layer. The hardware-software interface uses Linux shared memory (IPC Linux *shm*) for the interaction, data and synchronization exchange between the software and the hardware.

Figure 4.10 shows the execution model of two software stacks running on two processors, ARM7 and XTENSA. This represents a co-simulation between the gdb Linux processes of each software stack *gdb1* and *gdb2* (one gdb per each software stack) and one SystemC Linux process for the whole SystemC simulation of the hardware platform. The interface between the three Linux processes is performed using the Linux IPC shared memory.

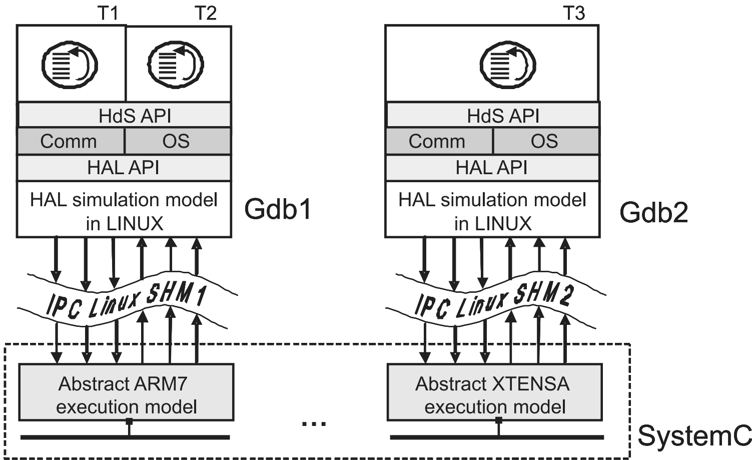

Figure 4.10. Transaction accurate SystemC execution model.

The simulation of the transaction accurate architecture allows validation of the integration of the tasks code with the OS and the communication protocol, providing a simulation model for the HAL APIs. The simulation allows debugging the software access to the hardware resources (e.g. access to the AMBA

bus, interrupt lines assignment, OS scheduling, etc.). It makes possible the debug of the access of the OS functions to the hardware resources through the HAL APIs, e.g. *read(...)*/ *write(...)* from/to the memory, explicit synchronization using mailboxes or the interrupt routine services. The simulation also gives more precise statistics on the communication and computation performances, such as number of exchanged bytes during the application execution or estimation of the processors cycles spent on communication.

4.7 Experiments

In this chapter, we present the HAL integration for the Motion JPEG decoder application. This application targets various hardware architectures, involving Xtensa processor [Xte], ARM processor [ARM] or Atmel DSP [mVD].

The Motion JPEG Decoder application represents an image processing multimedia application. In this chapter, the baseline Motion-JPEG decoder is used as target application example, which represents the basic JPEG decoding process supported by all the JPEG decoders [Wal91]. The JPEG decoder performs the decompression of an encoded JPEG bitstream (01011...) and renders the decoded bitmap images on a screen. The JPEG compression algorithm operates on blocks of 8×8 pixels of the image. The main functions of the Motion JPEG application, as illustrated in Fig. 4.11 are:

- Variable Length Decoding (VLD), which transforms the input binary sequence into a symbol sequence using the Huffman tables
- Differential Pulse Code Demodulation (DPCD) applied upon the DC coefficient
- Run Length Decoding (RLD) applied upon the 63 AC coefficients
- Zigzag Scan, which reconstructs the matrix of the DCT coefficients from the DC and 63 AC elements
- Inverse Quantization (IQ), which uses the quantification tables

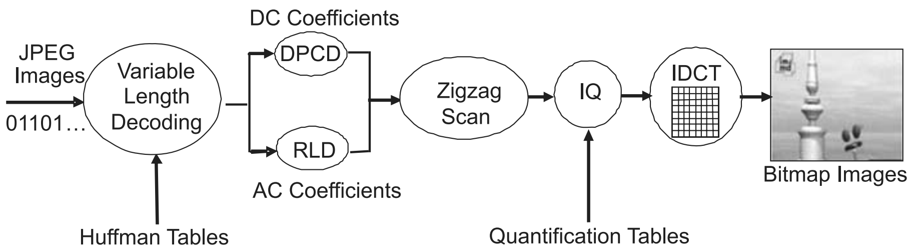

Figure 4.11. Motion JPEG decoder.

- Inverse Discrete Cosine Transformation (IDCT), which transforms the DCT coefficients from frequency domain to spatial domain.

The experimentation is carried out by using three types of processor cores. The first processor core represents the Xtensa processor [Xte]. This processor works at 350 MHz frequency and has 8 Kbytes data cache and 8 Kbytes instruction cache memories. The second core belongs to ARM9 processors family and represents the ARM926EJ-S type of core [mVD]. This runs at 200 MHz frequency and is equipped with 16 KBytes data cache and 16 KBytes instruction cache memories. The third processor represents the magicV VLIW DSP Atmel processor, running at 100 MHz [Wal91].

The Motion JPEG application aims to be executed on these different types of processors. A small OS is used to start the execution of the application and to initialize diverse hardware devices, i.e. I/O devices. The execution and portability of the application software is performed by changing the HAL component of the software stack. The processor specific application code optimization techniques are not considered during the experimentation, in order to preserve the application code hardware independent. Due to the use of the HAL APIs, the application code and OS remains unmodified, thus enabling software portability. The OS makes use of the same HAL APIs for all the hardware architectures. We illustrate three examples of HAL APIs that are identical for the different processors. These HAL APIs are the following:

- *_set_context()* HAL API, which initializes the task that will be executed by the processor, more precisely the stack

- *vector_enable()* HAL API, which enables the interrupts

- *vector_disable()* HAL API, which disables the interrupts

Figures 4.12, 4.13 and 4.14 illustrate the diverse implementations of the same HAL APIs targeting the Xtensa, ARM9, respectively DSP processor. The implementation for the ARM9 processor mainly uses assembly language. The implementations of the HAL APIs for the Xtensa processor and the DSP are based on other APIs, provided by the processors vendors.

After the compilation of the software stack, composed of the Motion JPEG decoder application, a tiny OS and the HAL specific to each CPU, the resulted memory requirements are as follows: 3072 bytes data memory and 4802 bytes of code size for the program memory in case of the Xtensa processor, 3056 bytes data memory and 5092 bytes program memory for the ARM9 processor, respectively 739 bytes data memory and 2806 bytes program memory for the DSP. Table 4.1 summarizes these values and also the code and data size of the HAL for the three different types of processors.

Xtensa

```
void _set_context(thread_context_t buf, fcall_t function, void *stack)
{
          _setjmp(buf);
          buf[JB_PC] = (int)(function);
          buf[JB_SP] = (int)(stack + STACK_SIZE);
}
```

ARM9

```
_set_context:
          stmia     r0!, {r0-r10, fp, ip}         @ we save r0-r12
          stmia     r0!,  {r2}                    @ sp
          stmia     r0!,  {r1}                    @ lr
          stmdb     sp!, {r4-r5}
          mrs       r5, cpsr                      @ we get the cpsr
          mrs       r4, spsr                      @ and the spsr
          stmia     r0!, {r4-r5}                  @ we can save them
          ldmdb     sp!, {r4-r5}
          mov       pc, lr                        @ and we branch
```

DSP

```
void _set_context(thread_context_t buf, fcall_t function, void *stack)
{
           _DBIOS_Init(function, buf, stack);
}
```

Figure 4.12. Implementation of _set_context() HAL API for different processors.

Xtensa

```
void vector_enable(int irq_type)
{
           if(intr_init==0)
                   {
                    _xtos_set_interrupt_handler(0, _irq_dispatch);
                    intr_init =1;
                   }
           _xtos_ints_on ( 1L << (0));
}
```

ARM9

```
void vector_enable(int irq_type)
{
           asm("mrs r1, cpsr");
           asm("bic r1, r1, r0");
           asm("msr cpsr_c, r1");
}
```

DSP

```
void vector_enable(int irq_type)
{
           AT91F_DSP_INTERRUPT_Enable (irq_type);
}
```

Figure 4.13. Implementation of the vector_enable() HAL API for different processors.

Xtensa

```
void vector_disable(int irq_type)
{
            _xtos_ints_off ( 1L << (0));

}
```

ARM9

```
void vector_disable(int irq_type)
{
            asm("mrs r1, cpsr");
            asm("orr r1, r1, r0");
            asm("msr cpsr_c, r1");
}
```

DSP

```
void vector_disable(int irq_type)
{
            AT91F_DSP_INTERRUPT_Disable (irq_type);
}
```

Figure 4.14. Implementation of the vector_disable() HAL API for different processors.

Processor	Application		HAL	
	Data [Bytes]	Code [Bytes]	Data [Bytes]	Code [Bytes]
Xtensa	3072	4802	112	1185
ARM9	3056	5092	14	1248
DSP	739	2806	52	296

Table 4.1. Code and data size.

Figure 4.15 illustrates the total execution cycles measured when executing the whole Motion-JPEG application on the different processors using ISS. In all the cases, the input bitstream represents a 10 frames image encoded using QVGA format, and stored in the local memory of the processor. As shown in Fig. 4.15, the number of execution cycles required to decode the 10 frames image is approximately 137 Mega cycles on the Xtensa processor, 71 Mega cycles on the ARM9 processor and 164 Mega cycles on the DSP. Table 4.2 indicates the characteristics of each of these processors, as specified by the IP vendors, in terms of speed (clock frequency), surface and corresponding power consumption. The processors are configured as shown in Table 4.2.

The performance difference between the processors is explained by the availability of the additional cache memories and improvement in number of cycles required for the load/store operations. The real time requirement of 25 frames decoded per second implies an execution per frame within 8 Mega cycles on a CPU running at 200 MHz, 4 Mega cycles on a CPU running at 100 MHz

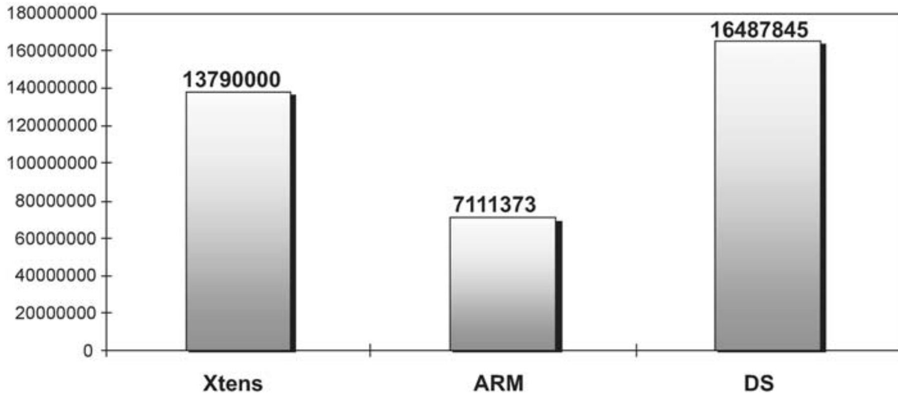

Figure 4.15. Execution clock cycles of Motion JPEG decoder QVGA.

Processor	Frequency	Surface	Power Consumption
Xtensa (core)	350 MHz	0.26 mm^2	26.25 mW
ARM9	200 MHz	2.78 mm^2	96 mW
DSP	100 MHz	13.2 mm^2	229.2 mW

Table 4.2. Frequency, surface and power consumption.

and 14 Mega cycles on a CPU running at 350 MHz. Thus, the Motion JPEG decoder can be executed in real-time by using the ARM9 processor and the Xtensa configurable processor. The surface of the hardware in case of the ARM9 processor is 2.78 mm^2 with a power consumption of 96 mW. The execution on the DSP can be improved by using DSP specific optimization features in order to speed up the critical computing part of the application. But the processor specific application optimization reduces software portability. The DSP is the biggest power consumer among the three targeted processors, and it implies a surface of 13.2 mm^2. The Xtensa core is the optimal processor in terms of surface and consumption, but it is not equipped with any extra hardware accelerators in the configuration used during the experimentation.

4.8 Conclusions

In this chapter, we presented a layered organization of the software stack into application tasks code, operating system and communication libraries, and HAL. The structured representation of the software stack separates the hardware independent and hardware dependent software layers, thus allowing easy software portability. The different software components are generated and validated gradually by using specific software development platforms. Abstracting and simulating the HAL through HAL APIs allows software reuse and flexibil-

ity. To illustrate the effectiveness of the proposed methodology, we generated the software stack for the Motion JPEG application targeting different hardware architectures. The execution of the Motion JPEG on multiple processors (Xtensa, ARM9, DSP) was possible due to the clear separation between the hardware independent software code (application tasks code, OS and communication) and the hardware dependent HAL.

References

[A386] A386. a386.nocrew.org

[ARM] ARM. www.arm.com

[BYJ04] A. Bouchima, S. Yoo, and 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*, January 2004, Yokohama, Japan, 2004.

[CYC+05] Y. Cho, S. Yoo, K. Choi, N.E. Zergainoh, and A.A. Jerraya. Scheduler implementation in MPSoC design. In *Proceedings of ASP–DAC 2005*, 18–21 January 2005, Shanghai, China, pages 151–156, 2005.

[eCos] eCos. www.ecos.sourceware.org/docs-1.3.1/ref/ecos-ref.b.html

[EPTP07] C. Erbes, A.D. Pimentel, M. Thompson, and S. Polstra. A framework for system-level modeling and simulation of embedded systems architecture. *EURASIP Journal on Embedded Systems*, Volume 2007, Article ID 82123, June 2007.

[FRTOS] FreeRTOS. www.freertos.org

[GLMS02] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic, Dordrecht, 2002.

[GPY+07] X. Guerin, K. Popovici, W. Youssef, F. Rousseau, and A. Jerraya. Flexible application software generation for heterogeneous multi-processor system-on-chip. In *Proceedings of COMPSAC 2007*, 23–27 July 2007, Beijing, China, 2007.

[HAL] HAL. www.altera.com/literature/hb/nios2/n2sw_nii5v2_02.pdf

[Han06 +] S.I. Han et al. Buffer memory optimization for video codec application modeled in simulink. In *Proceedings of DAC 2006*, San Francisco, USA, pages 689–694. IEEE Press, New York, 2006.

[HYL+06] S. Hong, S. Yoo, S. Lee, S. Lee, H.J. Nam, B.S. Yoo, J. Hwang, D. Song, J. Kim, J. Kim, H. Jin, K. Choi, J.T. Kong, and S. Eo.

Creation and utilization of a virtual platform for embedded software optimization: An industrial case study. In *Proceedings of CODES+ISSS 2006*, Seoul, Korea, 2006.

[JBP06] A. Jerraya, A. Bouchhima, and F. Petrot. Programming models and HW–SW interfaces abstraction for multi-processor SoC. In *Proceedings of DAC 2006*, San Francisco, USA, pages 280–285. IEEE Press, New York, 2006.

[JW05] A. Jerraya and W. Wolf. Hardware–software interface codesign for embedded systems. *Computer*, 38(2):63–69, 2005.

[KSW+00] E.A. de Kock, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, and G. Essink. YAPI: application modeling for signal processing systems. In *Proceedings of DAC 2000*. IEEE Press, New York, 2000.

[LOS] LynxOS. www.lynuxworks.com/rtos

[Math] The MathWorks. www.mathworks.com

[MPI] MPI. www-unix.mcs.anl.gov/mpi

[mVD] magicV VLIW DSP. www.atmel.com

[newl] newlib. sourceware.org/newlib

[NS] NicheStack. www.iniche.com/nichestack.php

[NYBJ02] G. Nicolescu, S. Yoo, A. Bouchhima, and A.A. Jerraya. Validation in a component-based design flow for multicore SoCs. In *Proceedings of ISSS'02*, 2–4 October 2002, Kyoto, Japan, 2002.

[PGR+07] K. Popovici, X. Guerin, F. Rousseau, P.S. Paolucci, and A. Jerraya. Efficient software development platforms for multimedia applications at different abstraction levels. In *Proceedings of IEEE RSP 2007*, May 2007, Porto Alegre, Brazil, pages 113–122, 2007.

[PJ07] K. Popovici and A. Jerraya. Simulink based hardware-software codesign flow for heterogeneous MPSoC. In *Proceedings of SCSC 2007*, 15–18 July 2007, San Diego, USA, pages 497–504, 2007.

[Pos03] F. Pospiech. Hardware dependent software (HdS). Multiprocessor SoC aspects—An introduction. In *Proceedings of MPSoC 2003*, 7–11 July 2003, Chamonix, France, 2003.

[Row94] J.A. Rowson. Hardware/software cosimulation. In *Proceedings of DAC 1994*, San Diego, USA, pages 439–440. IEEE Press, New York, 1994.

[RTL] RTLinux. www.fsmlabs.com

[SG00] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings of ASPDAC 2000*, Yokohama, Japan, pages 405–408, 2000.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, 1995.

[Tur05] J. Turley. Survey says: Software tools more important than chips. *Embedded Systems Design Journal*, 2005.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, Englewood Cliffs, 1997.

[uIT] uITRON4.0. www.sakamura-lab.org/tron/itron

[VBL05] N. Ventroux, F. Blanc, and D. Lavenier. A low complex scheduling algorithm for multi-processor system-on-chip. In *Proceedings of Parallel and Distributed Computing and Networks*, 15–17 February 2005, Innsbruck, Austria, 2005.

[vdW04+] P. van der Wolf et al. Design and programming of embedded multiprocessors: an interface-centric approach. In *Proceedings of CODES+ISSS 2004*, Stockholm, Sweden, pages 206–217, 2004.

[VxW] VxWorks. windriver.com/vxworks

[Wal91] G.K. Wallace. The JPEG still picture compression standard. *Communications of the ACM, Special Issue on Digital Multimedia Systems*, 34(4):30–44, 1991.

[WCE] Windows CE. www.microsoft.com/windows/embedded

[Wol06] W. Wolf. *High Performance Embedded Computing*. Morgan Kaufmann, San Mateo, 2006.

[Xte] Xtensa. www.tensilica.com

[YJ03] S. Yoo and A. Jerraya. Introduction to hardware abstraction layers for SoC. In *Proceedings of DATE 2003*, 3–7 March 2003, Munich, Germany, pages 336–337, 2003.

[YYS+04] M.W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, and A. Jerraya. Debugging HW/SW interface for MPSoC: Video encoder system design case study. In *Proceedings of DAC 2004*, 7–11 June 2004, San Diego, USA, pages 908–913. IEEE Press, New York, 2004.