

# Embedded Linux Co-simulation

Julien TAILLARD      Philippe MARQUET      Jean-Luc DEKEYSER

<http://www.lifl.fr/west>

Laboratoire d'informatique fondamentale de Lille

Université des sciences et technologies de Lille

France

## Abstract

The augmentation of number of gates on chip makes SOC design more difficult. So we have to work on SOC design tools to make designer work easier and manage all the available gates.

We propose an embedded Linux co-simulation with hardware simulation at a high level of abstraction (TLM) to verify the system very early in the design flow. This will allow to avoid return behind in the design flow.

## 1 Introduction

Despite System on Chip usage is the trend in many fields, the complexity of their design may slow down their dissemination. As predicted by the Moore's law, the number of transistors which can be put on a chip doubles every 18 months. Nevertheless, designers are not able to benefit from this progression. The chip surface is not used entirely due to the complexity of the design process. To face this problem and reduce this so-called design gap, between what can be integrated on a chip and what designers are able to produce, new design and simulation tools must be considered. A co-simulation tool that is able to mix a high level description of the hardware with the basic functions of an operating system and the applications may alleviate the designer's task.

Our goal is to provide a simulation of the execution of application tasks on an embedded Linux operating system running on a hardware which is described at a high level of abstraction. This hardware/software co-simulation can be made very early in the design flow. In fact, the co-simulation

is separated into two simulation modules: a software simulation and a hardware simulation. They communicate via a socket protocol.

This paper is organized as follows. Section 2 presents related works on Transaction Level Modeling and software simulation. Section 3 explains how we have made our co-simulation. Section 4 concludes the paper and extracts futures works.

## 2 Related Work

A co-simulation has two parts, the hardware part and the software part. Hardware simulation are generally written with a Hardware Description Language (HDL), like SystemC or VHDL, while software are written in classical language (C/C++). First, we will see the model used for hardware description and then how software simulations are made.

Nowadays, hardware models are generally designed in RTL (Register Transfer Level), because a lot of tools take it in entry. However this level cannot scale enough anymore, the increase in productivity is limited and incidentally the number of gates used is restrained by the complexity of managing such a great number of gate. A solution to improve the productivity is to use a higher abstraction level, this was the reason behind the introduction of the Transaction Level Modeling (TLM).

### 2.1 Transaction Level Modeling

The Transaction Level Modeling (TLM) has been introduced to bridge the productivity gap. TLM is not a single level, it's a group of level. Galski and Cai have defined several levels of TLM [1, 2].

Those levels (figure 1) are:

- assembly model,
- bus-arbitration model,
- time-accurate bus model,
- time-accurate computation model.

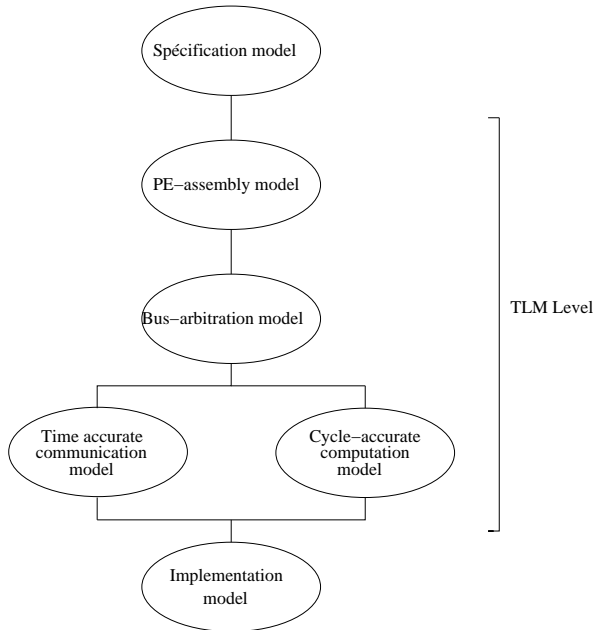


Figure 1: TLM levels

The first model is *assembly model*. The entities at the top level of the model represent concurrently executing processing elements (PEs), which communicate through channels. The channels allow to communicate with complex data type. It is an abstraction of the bus protocol. The communication part of the model is un-timed while computation part is timed through estimation. The estimated time is annotated into the code by inserting wait statement.

The second model is *bus-arbitration model*. At this level, the channels are blocking and non-blocking I/O. No cycle-accurate and pin-accurate protocol details are specified. The communication are timed approximately with a wait statement by transaction.

The third level is *time-accurate bus model*. This model contains time/cycle accurate communication and approximated time computation.

The channels are replaced by protocol channels which are time/cycle-accurate and pin-accurate.

The fourth level is *time-accurate computation model* where the computation part is cycle-accurate timed and the communication part is estimated. Computation components are pin-accurate and execute cycle-accurately, but the communication is made by estimated-time channels.

TLM gathers some levels of abstraction from high level to the limit of RTL. Those levels can be used to simulate hardware before having a RTL model.

Three types of embedded software simulation can be distinguished [3]: functional simulation, usage of an Operating System (OS) simulation models and usage of instruction set simulators.

## 2.2 Functional Simulation

Functional simulation allows to verify the functionality of applications. This kind of simulation may be timed with delay annotations and may use the simulation environment (SystemC) for scheduling of software tasks.

## 2.3 OS Simulation Models

There are three types of OS simulation models: native OS, virtual OS and aggregate timing models of OS.

### 2.3.1 Native OS

A native OS runs the OS on a host workstation. The OS is compiled for the workstation. For instance, WindRiver Systems Inc. provides VxSim [4] as a native simulation model of its OS, VxWorks. It allows to easily validate the applications written for the OS. Timed co-simulation are not always supported.

### 2.3.2 Virtual OS

A virtual OS simulates the functionality of a real OS. The purpose of such kind of simulation is to validate the functionality and the timings of applications without to choose the final OS. Virtual OS can implement a dedicated API (Application Programming Interface) or use a standard API. Applications have to use the API of the virtual OS, so with a special API, applications have to be changed between the simulation and the final use. That is

why using a virtual OS which implements a standard must be a solution. It is made in [5] which uses the  $\mu$ ITRON 4.0 standard [6].  $\mu$ ITRON 4.0 is a standard of OS for little and middle system in Japan; 40% of embedded OS are based on this standard. So when a simulation is made with a standard, it is not necessary to adapt the application for use on the final OS. Virtual OSs suffer from the code equivalence problem, because the code of a virtual OS is not the same than the final OS. So a call to a virtual OS function cannot be guaranteed to take the same amount of time than the equivalent call to the final OS function.

### 2.3.3 Aggregate Timing Models

The aggregate timing model of OS is to simulate the timing delay of OS. A context switch delay can be calculated as a function of the number of ready tasks or the size of task context. Real-time system area used it to study the effect of OS in task schedulability analysis.

## 2.4 Instruction Set Simulator

This is a simulation at a very low abstraction level. An Instruction Set Simulator takes instructions as entry. So the software has to be compiled for the processor which is simulated. The time is managed by the Instruction Set Simulator. Each instruction takes the simulated time that it will take on the target processor.

## 3 Embedded Linux Co-simulation

Our goal is to provide an embedded Linux co-simulation with hardware simulation at a high level of abstraction. Therefore we use a hardware simulation at a TLM level (bus arbitration). In a goal of generality and standardization, our virtual OS has to implement a POSIX interface [7].

The co-simulation is divided in two parts: the hardware and the software (figure 2). Hardware simulation allows software simulation to call hardware functions like read/write. Software simulation simulates operating system and applications. Our co-simulation is timed, but annotated source code are not used. Annotated source code represents too

much constraint for us. So a multiplicative factor between simulation workstation and simulated processor has been introduced to estimate application execution time.

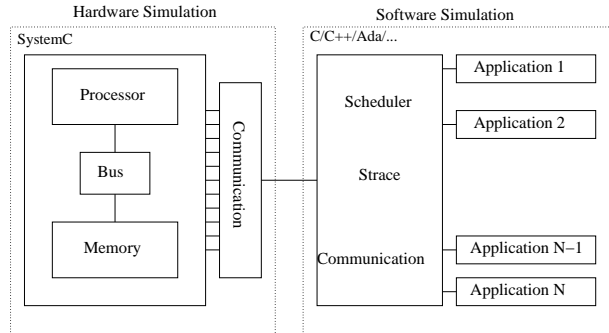


Figure 2: General view

## 3.1 Entity of Simulation

### 3.1.1 Hardware Simulation

Making a simulation early in the design flow requires to have a simulation at a high abstraction level, that is why TLM model simulation has to be supported. *Bus-arbitration model* is the base model, because all is estimated-time and it allows to have an idea of the execution time of system. Like TLM is not well widespread, RTL model can be supported too but it will slow down the simulation. The time unit will depend on hardware description. In TLM, seconds or milliseconds can be used but at RTL cycle unit has to be used. A communication interface will be used to make communication between hardware and software. This interface will have a generic part which can be reused with every hardware, and a hardware specific part which will be adapted to the simulated hardware.

### 3.1.2 Software Simulation

The software part is divided in two parts:

- applications,
- virtual embedded Linux.

**Applications** The applications are written for a classical Linux and compiled for the host workstation.

**Virtual embedded Linux** The virtual embedded Linux has to implement the POSIX standard. Instead of implement all POSIX functions, we have decided to trace application system calls. So when an application makes a `read()`, our virtual OS intercepts the signal and executes this call on hardware simulation if necessary.

## 3.2 Implementation

Some details of our implementation will be described. An overview of the usage of strace and a method to trace RAM accesses will be presented.

### 3.2.1 Strace

Applications are traced with the `ptrace` system call which allows a process to trace all system call of his children. The `strace` [8] program, which is a classical Linux utility, has been used. Strace was extended to send hardware calls to the hardware simulation and a dynamic scheduler was added in order to test different kinds of policy.

### 3.2.2 RAM Access

The `ptrace` system call does not allow to trace RAM accesses because accesses do not use any system call. Therefore a modification was required to add to strace this feature. The principle is as follows:

- forbid the application to access the RAM memory with a `mprotect(..., ..., PROT_NONE)`,
- when the application tries to access the memory, the segmentation fault (`SIGSEGV`) that occurs is caught and the execution is carry on in single step mode (`ptrace(PTRACE_SINGLESTEP, ...)`),
- the RAM access is allowed,
- the application executes one step and blocks again,
- RAM access are forbidden again,
- execution mode follows in normal mode (`ptrace(PTRACE_SYSCALL, ...)`).

RAM access tracing will slow down simulation, but it possible do deactivate it.

## 3.3 Execution Time of Software

Introducing annotation delay in software is very restricting, because source code has to be modified, so this technique was not employed. The execution time on host processor is considered as linear with the time on simulated processor. So a multiplicative factor will be determined to know the time on simulated processor. For example, if the application takes 5 time units on the host workstation and the multiplicative factor is 4, we will consider that it will take 20 units of time on the simulated processor.

The estimation of this factor is something which is not simple. Two possibilities have been imagined:

- use of benchmark,
- Comparison between technical specifications.

If the target processor is available, benchmarks can be written to test every part of the processor and estimate the factor. Benchmarks can be based on applications which represent real applications in term of memory access, CPU usage, etc. Benchmarks might also be the time measurement of “real” applications. If the processor is not available or does not exist yet, a comparison of the technical specification of both processors can give an broad estimate of the factor.

## 3.4 Communication Protocol

To allow communication between the two sides of the co-simulation, a protocol communication based on socket has been defined. This protocol has to provide:

- synchronization between the two parts of the simulation,
- triggering of the hardware functions.

Two types of synchronization have been identified:

- mandatory synchronization when a hardware call is made,
- optional regular synchronization.

A synchronization has to be made when a hardware function is called because hardware simulation has to know when the hardware call has to begin. The optional regular synchronization is useful to

have a better scheduling, because the virtual OS will know earlier the end of a hardware call than without.

### 3.4.1 Protocol Description

Software simulation controls hardware simulation, because hardware simulation does not have to precede software simulation. If the hardware simulation precedes software simulation, the software simulation could ask a hardware function at a time where the hardware is already passed then, the hardware cannot execute the function because going back into time is difficult. Consequently the hardware simulation waits for software synchronization to execute itself. When the hardware simulation receives a synchronization signal, it advances until the synchronization time. When it advances, it makes the hardware call if necessary.

### 3.4.2 Commands Description

Before each hardware call, a synchronization is needed. This synchronization will be triggered by the “timer” command.

**Synchronization** Commands where  $t$  represents time.

- timer  $t$ : synchronization before hardware call,
- synchro  $t$ : regular synchronization,
- end: to tell the end of a synchronization.

**Hardware calls** Until now, two hardware calls have been defined, but this will be extended to all simulable system calls.

- read: to execute a read,
- write: to execute a write.

### Special command

- execute: ask the hardware simulation to run until the next hardware call finishes

### 3.4.3 Example

Here is the description of an example of communication, which is schematized on figure 3.

In this example, there are two applications, the software simulation and the hardware simulation.

The software simulation schedules application 1. At time T1, application 1 does a `write()` which is detected by virtual OS simulation. So, the virtual OS pauses application 1. Then, it asks an execution to the hardware simulation with the two commands “timer T1” and “write 1”. Then the hardware simulation runs until T1, and sends “end”. The virtual OS can now schedule application 2 which is executed until T2 where the OS detects a `read()`. Like before, the OS asks a hardware execution with “timer T2” and “read 2”, and the hardware answers “end”. At this moment, all the tasks (application 1 and application 2) are blocked, waiting for a hardware simulation. Therefore the software simulation has to ask hardware simulation to run until a call finishes. This is requested via the “execute” command. The answer of the hardware is “timer T3” and “end 1”, which means that the call 1 has finished at time T3. Consequently, at time T3, application 1 can restart until T4 where his quantum of time expires. In the example, regular synchronisation is activated that is why at time T4, synchronisation is made with “synchro T4”. When this synchronisation is made, the end of the call number 2 is detected. Without this synchronisation, this end will be detected later. So the scheduling will not be identical but always valid.

## 4 Conclusion and Future Work

A new method of co-simulation has been presented that includes a virtual embedded Linux and a highly abstracted hardware model. This method allows to compare many parameterization of the operating system and/or the applications and hardware at an early stage in the design flow. This simulation platform is now implemented and we are currently validating its usage on representative designs. This first implementation will be extended to support real-time Linux and multi-processor hardware simulation.

## References

- [1] Lukai Cai and Daniel Gajski. Transaction level modeling in system level design. Technical report, Center for Embedded Computer Systems,

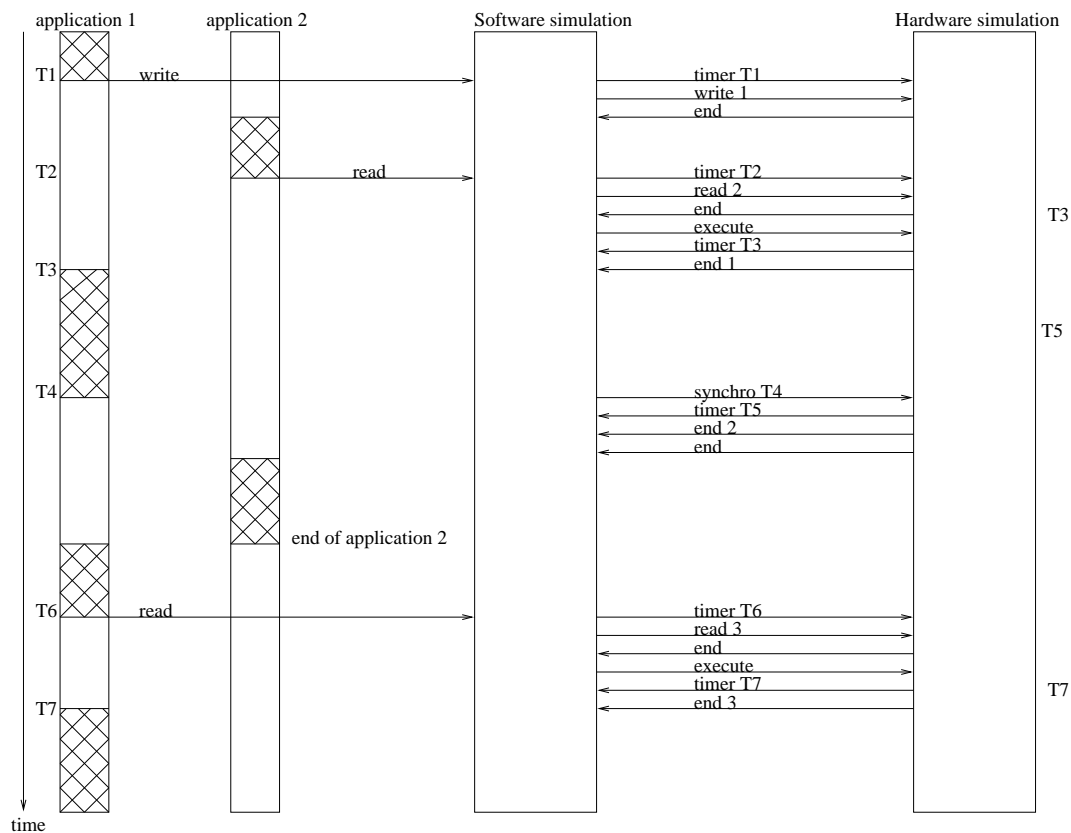


Figure 3: Communication protocol example

- Information and Computer Science, University of California, Irvine, CA, 2003.
- [2] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. In *Hardware/Software Codesign and System Synthesis*, pages 19–24, October, 2003.
- [3] Sungjoo Yoo, Gabriela Nicolescu, Lovic Gauthier, and Ahmed A.Jerraya. Automatic generation of fast timed simulation models for operating systems in soc design. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pages 620–627, Paris, France, March 2002.
- [4] Vxworks. World Wide Web document, URL: [http://www.windriver.com/products/development\\_tools/ide/wind\\_river\\_vxworks\\_simulator/vxsim.pdf](http://www.windriver.com/products/development_tools/ide/wind_river_vxworks_simulator/vxsim.pdf).
- [5] Shinya Honda, Takayuki Wakabayashi, Hiroyuki Tomiyama, and Hiroaki Takada. Rtos-centric hardware/software cosimulator for embedded system design. In *Hardware/Software Codesign and System Synthesis*, pages 158–163, September 2004.
- [6]  $\mu$ itron. World Wide Web document, URL: <http://www.asso.tron.org>.
- [7] Posix. World Wide Web document, URL: <http://www.opengroup.org/certification/idx/posix.html>.
- [8] strace. World Wide Web document, URL: <http://strace.sourceforge.net>.