TECHNIQUES FOR COMPILED HW/SW COSIMULATION

Andreas Ropers, Stefan Pees and Thomas Brüggen
Institute for Integrated Signal Processing Systems
Aachen University of Technology
Templergraben 55, 52056 Aachen, Germany
e-mail: {ropers,pees,brueggen}@ert.rwth-aachen.de

Abstract

This paper presents a new approach for the coupling of compiled hardware and software simulators. Unlike existing methods this technique offers a very flexible way for integrating a software simulator into the hardware environment. In contrast to existing in-house or off-the-shelf hardware software cosimulators the user has the ability to the select the appropriate cosimulation method. In addition the compiled hardware/software cosimulation approach offers a very high simulation speed. From a very tight coupling of the simulators which improves the simulation speed, to a loose interaction of the simulators which improves flexibility, various kinds of cosimulation mechanisms can be applied. In this paper the different methods for cosimulation and the limitations are analyzed and the realization of a HW / SW cosimulation environment is presented

1 Introduction

With the increasing complexity of DSP's and ASIC's the design verification has become the most critical aspect of assuring the overall product quality and performance. Typically, the complexity ot these designs actually disqualifies the use of a unique level of abstraction (a unified model) to verify all aspects of a design. As the hardware-part of the design matures, many different problems have to be faced using different models. In order to achieve acceptable performance, each model is used in conjunction with one specific verification tool. During the design phase different verification tools have to be used to ensure a complete design verification.

The interface between the hardware and the software can be modeled on different abstraction layers. On the *application layer* only signals like *send, receive* and *wait* are modeled. The next lower abstraction layer is based on the operating system or device drivers (software part) and the bus interface (hardware part). Signals like register reads and writes are modeled on this level of accuracy. The lowest level of abstraction is presented by the *bus functional model* where pin-accuracy is required. In general asynchronous communication schemes between hardware and software allows for an abstract communication method. This could be for example, a process or device communication mechanism. This speeds up simulation but make performance evaluation more inaccurate.

Debugging and verification can be done using hardware or software based models. The main advantage of hardware models (*emulators*) is their speed. The main disadvantage of emulators is their low flexibility, high costs and an the reduced visibility of internal states. In addition the boundary between the hardware and the software is fixed, thus contradicting the main principle of HW/SW co-design.

This is the reason, why most design-engineers prefer software-based models. The main drawback of software models is the reduced simulation speed, which is up to four times slower than the real hardware. Whereas most of the cosimulation approaches rely on the interpretive simulation technique, compiled approaches proved to speed up the simulation up to 2 orders of magnitude[1]. The principle of compiled simulation is a well known approach for hardware simulation. For software simulation this technique has been developed in the last four years [2, 3]. The speedup achieved with this new technique is up to three orders of magnitude. Of course in a hardware / software cosimulation environment the overall speedup can be reduced (Amdahl's Law). Let us suppose that for example a hardware- and a software simulator are scheduled the same times. If the hardware simulator is 10 times slower, the software simulator can only speedup the whole simulation about 10%. On the other hand that means that a system containing only a little amount of hardware but a complex algorithm on a DSP, can gain a lot of a fast software simulator. Due to that it makes a lot of sense to use the compiled simulation principle, in the hardware- and the software-simulator.

This paper is organized as follows. After the introduction in Section 2 the motivation guiding this work is presented. Section 3 discusses previous work which is related to those presented in the paper. The main possible connections of hardware and software simulators are presented in Section 4. The implementation and performance measurements are given in Section 5. Finally, in Section 6 the conclusions are given.

2 Motivation

The main motivation for the work presented in this paper was to analyze the effects of different hardware/software cosimulation techniques. In the past the connection of the different simulators was often thought to be a serious bottleneck in the simulation performance. With the development of fast simulators and the technique of the *compiled simulation* a very high simulation speed can be reached. So the amount of time wasted in the communication between the simulators becomes even higher. In order to provide a fast HW/SW cosimulation environment, we have to identify the bottlenecks and provide a way to connect the software simulator with the hardware environment. This was the motivation for our work to implement a cosimulation between a hardware and a software simulator, both using the principle of compiled simulation. The software simulator we used is the SUPERSIM described in [4]. For the hardware simulator we used CYCLONE, a RTL-Level, cycle based VHDL-simulator provided from Synopsys.

3 Previous Work

Till today almost all DSP-simulators use the interpretive simulation technique. They are shipped with off-the-shelf or in-house DSP processors and offer a comfortable

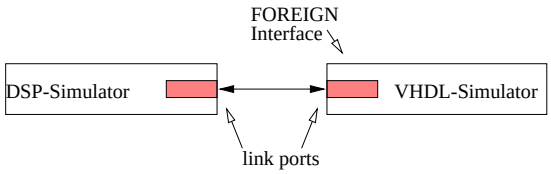

Figure 1: Direct communication scheme

debugging frontend. In addition the user has full visibility of internal states and breakpoints for example do not interfere with the processor state in contrast to emulator boards. The main disadvantage of these DSP-simulators is their low simulation speed[5] and the inability to be adopted for a special environment. All existing hardware/software cosimulation environments rely on the interpretive simulation technique at least for the software part [6, 7].

The connection of hardware and software simulators can be done in different ways. In all cases API's[1] have to be provided. In [8] a systematic overview of different API's is given. An automatic generation of such interfaces is given in [9]. On the one hand two simulators can be coupled in a direct manner. This has the advantage that the interface can be optimized for this special purpose. On the other hand it is expendable to implement these link modules for more than two simulators. For example for three simulators 6 link modules are needed to connect each simulator to the other in an optimized way. The amount of link modules needed can be reduced with a *simulation backplane*. The simulation backplane[10] provides an API and a simulation layer for the connection of various simulators, thus suffering from the requirement to connect various kinds of simulators. So a tradeoff has to be done between a very sophisticated solution for the cosimulation with one hardware and one software simulator, and a more general cosimulation environment if more than two simulators are supported.

4 Connection Schemes

The connection between a hardware and a software simulator is essential for a powerful and flexible HW/SW cosimulation environment. Even the fastest simulator is useless if an inappropriate communication scheme is used. This chapter gives an overview on possible connection methods and analyses pros and cons. Various possibilities are provided by the software vendors to connect the different debuggers and hardware simulators. In general communication schemes for simulators can be separated in:

- direct connections and
- indirect connections

Common to both approaches is the need to have a link in the simulator. The only difference is that the indirect connection makes use of Inter-Process-Communication (IPC). Figure 1 depicts the direct communication scheme. The simulators are coupled with two appropriate link ports, one of them the *FOREIGN*-interface of the VHDL-language.

[1] Application Programmers Interface

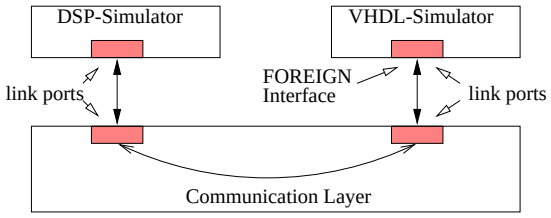

Figure 2: Indirect communication scheme

This offers the possibility to implement a certain behaviour not in VHDL but in a different module (C or other languages).

So every time the entity is activated by the hardware simulator the DSP-simulator is invoked once and can perform the appropriate behaviour. So the scheduling of the simulator is determined by the activation of the entity. As the two simulators are not able to run in parallel, a conservative scheduling mechanism results which means that each simulator is invoked once for each cycle step.

Even if the hardware (VHDL-code) does not need data from the DSP-simulator it has to wait until the DSP has finished his operation. The advantage is that there is no IPC overhead.

Direct connections require a linkable object of the simulator. The advantage is the lack of communication overhead, thus enabling a powerful cosimulation. The disadvantage is, that this technique is based on the existence of an object code of the simulator. In addition the software vendor has to provide external entry points for the main simulation functions. For the following examples we assume that the hardware simulator acts as a master and the software simulator is the slave. This implies that the DSP-simulator is embedded in the hardware environment.

This holds true for the *indirect communication* schemes (Fig. 2).

The communication layer can be implemented in different ways. The most popular IPC mechanisms are *file-IO*, *Pipes* or *Shared Memory/Semaphores*. All these can be implemented in different ways like *POSIX* or *Solaris* IPC. For our experiments we used the shared memory in conjunction with semaphores for a proper synchronisation. As the Solaris shared memory and semaphores are supposed to be much heavy weighted, we also implemented the POSIX version in order to measure the overhead. The advantage of an indirect communication scheme is, that both simulators can run in parallel on different processors or machines. The synchronization between the simulators is done for example by means of IPC-semaphores. The disadvantage is the additional overhead introduced by the IPC-mechanisms.

5 Implementation

The implementation of both, the direct and the indirect connection is illustrated on a small application example (Fig. 3). Because of the complexity of *real-life* examples we restrict ourselves to a simple example to show the principle of the cosimulation mechanisms. Here the Texas Instruments TMS320C54x-DSP does a multiplication and the hardware-accelerator adds a constant. The DSP has a multistage pipeline which is simulated phase-accurate. That means that each pipeline stage is simulated separate and

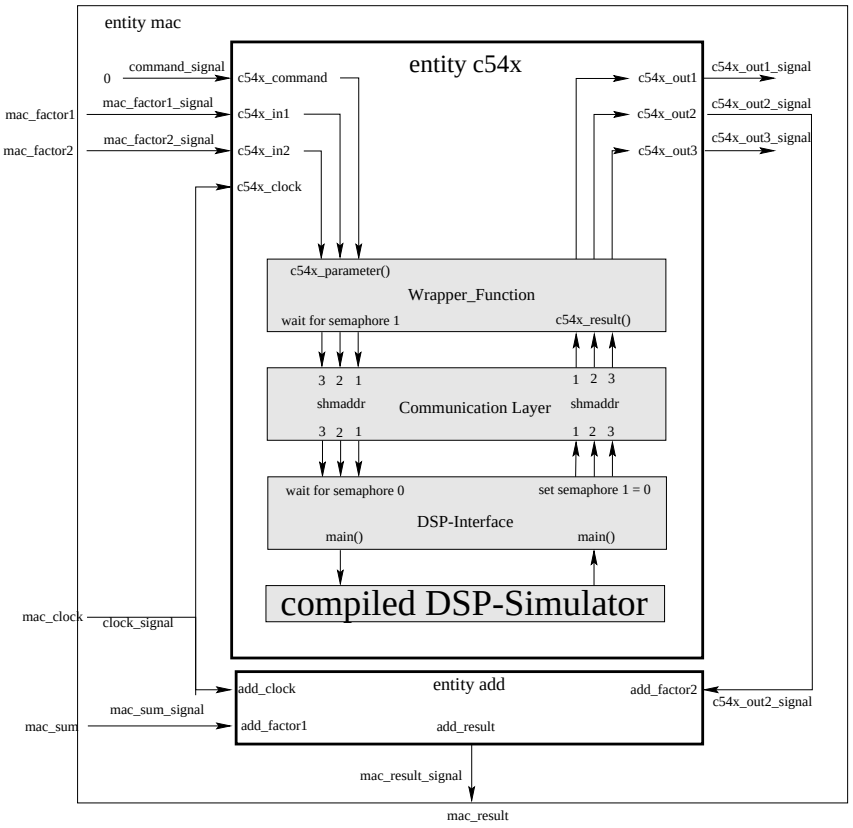

Figure 3: Application example: DSP interacting with MAC-Unit

the user has full visibility of all internal states. Of course this level of detail slows down the simulation speed dramatically compared to *normal* cycle-accurate DSP-simulators. First we describe the implementation for the indirect communication method. The direct method works the same way, but uses a direct link through the communication layer, so that this one is not needed.

VHDL-Interface

The VHDL-interface is implemented by using the *FOREIGN*-interface of the VHDL-language. The behaviour of the hardware (here the DSP) is not written in VHDL code, but in an external function. This allows us to include the whole DSP within a few lines of VHDL-code.

```
1    ARCHITECTURE c54x_behaviour OF c54x IS
2
3      -- entry point
4      -- init shared memory and semaphore
5      PROCEDURE init_all_c IS
6      BEGIN
7      END;
8      ATTRIBUTE foreign OF init_all_c: PROCEDURE IS "init_all";
9
10     --entry point
11     PROCEDURE c54x_param_c (PARAMETER_LIST: OUT INTEGER) IS
12     BEGIN
13     END;
14     ATTRIBUTE foreign OF c54x_param_c: PROCEDURE IS "c54x_param";
15
16     --entry point
17     PROCEDURE c54x_result_c (RESULT_LIST: IN INTEGER) IS
18     BEGIN
19     END;
20     ATTRIBUTE foreign OF c54x_result_c: PROCEDURE IS "c54x_result";
21
```

Line 1 defines the behaviour of the DSP (C54x) architecture. In the body of the architecture declaration three external functions are declared. The first one in line 5 is the *init*-routine which initializes the DSP. The other two declarations in line 11 and 17 are the entry points for the output resp. input of the parameters. This is a very flexible way to embed software simulators into a hardware environment. Multiple instances of a DSP-simulator can be used, only limited by the amount of memory. So complex multiprocessor systems can be simulated which offers a powerful tool for the debugging and verification of such a system.

Communication Layer

The communication layer offers various commands which are used by the link ports to establish a communication. These are provided as *C++-class* in order to hide the implementation details and to offer a clean API. The following commands are implemented for POSIX and Solaris:

- *establish communication*: this includes the reservation of shared memory for data transfer, semaphores for the synchronization and the instantiation of a DSP-simulator (which is automatically launched by invoking this initialisation routine).
- *send data*: allows the VHDL-simulator to pass data to the DSP-simulator.
- *receive data*: allows to receive data from DSP-simulator.
- *send command*: allows to send commands like *simulate instruction* to the DSP-simulator.
- *end communication*: cleans all reserved memory and semaphores and closes the DSP-simulator.

The commands *establish communication* and *end communication* are normally invoked during the initialization (resp. cleanup)-phase of the VHDL-simulator. So the

user only has to start the VHDL-simulator and all other things are done automatically within the startup-phase.

DSP-Simulator

The link port of the DSP-simulator is implemented as a wrapper function that encapsulates the internal functions of the simulator in a *C++*-class. The commands are the same as for the communication layer.

Results

The following table shows the results for the different cosimulation methods. The DSP-simulator offers different stages of compiled simulation[4]. For the measurements we used the *dynamic scheduling*. In contrast to the *static scheduling* this is offered without an extra compilation step before simulation. The disadvantage is the lower simulation speed which is about two to three times compared to the static scheduling.

All results are performed on an Ultra-Sparc 1 (network-mode, all daemons on) with 170 MHz. The application is the MAC-application with the multiplication done by the DSP and the add-operation in hardware. This represents a very small application, but in order to analyze the IPC overhead we have to eliminate most of the simulation task. For each method two results are given. One with GUI-Update, the other one without. GUI update means, that after each step of the DSP-simulator the debugging frontend is updated. We see, that the GUI-update worsens the measurements by nearly two

	Simulation Speed [kCycles./sec.]	
Method	GUI-update	no GUI-update
POSIX	0.362	7.8
Solaris	0.370	8.1
Direct cosim.	0.454	34
Only DSP-sim.	0.5	47

Table 1: Measurements for cosimulation-methods

dimensions. Thus the differences in the methods can be measured, but are not clear. Obviously the original SuperSim is the fastest with about 500 cycles./sec followed by the direct cosimulation with 454 cycles./sec. The Solaris and POSIX implementations follow with 370 resp. 362 cycles./sec.
The differences in the simulation speed become more clear if the GUI update is omitted. The IPC cosimulation (POSIX and Solaris) starts with about 8 kCycles./sec. followed by the direct cosimulation with almost 34 kCycles./sec. Here we see that the Solaris IPC-mechanisms (especially shared memory and semaphores) are much heavy weighted. The POSIX implementation promises a much more efficient implementation of these communication methods, but the measurements show that they provide almost the same performance. If only the DSP-simulator is running, the simulation speed is about 47 kCycles./sec. This speed is only achieved because of the compiled simulation technique.

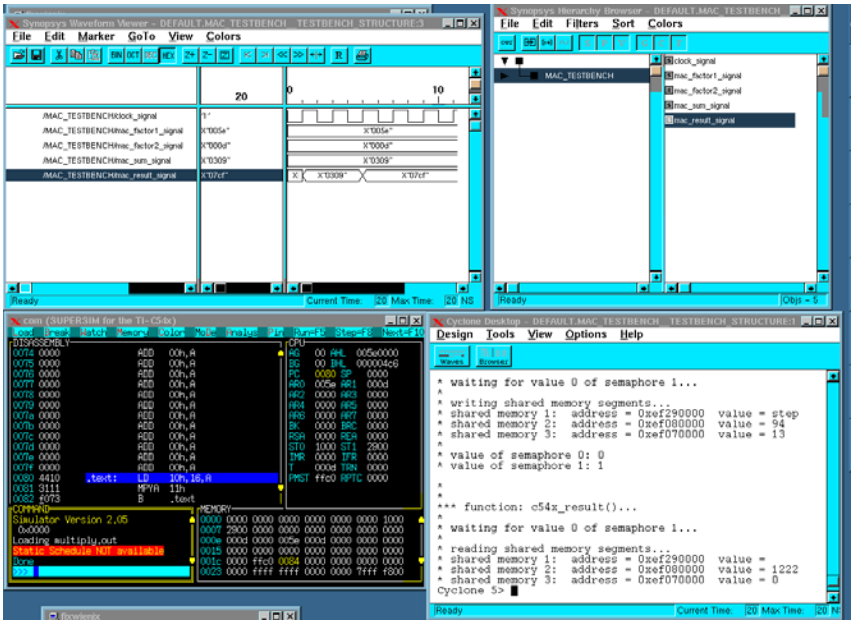

Figure 4: Debugging environment with DSP- and VHDL-simulator

By using the indirect cosimulation method with IPC the simulation speed is slowed down by a factor of three to four compared to the direct cosimulation. This is only valid if small applications are simulated. As soon as more complex examples are simulated, the overhead of the IPC is no longer significant. In this case, it might be advantageous to be able to use two or more machines for the simulation. So a tradeoff has to be done between the flexibility of the simulation and the simulation speed. Small examples will profit from the direct cosimulation, while bigger ones will be faster in terms of simulation on different machines, using the indirect cosimulation The simulation of a big *real-life* example will be one of or tasks in the future.

Adaption to other DSP- and VHDL Simulators

Our technique for the cosimulation of compiled simulators with VHDL simulators is tailored for the Cyclone simulator. Nevertheless other simulators can easily use this technique if some requirements are met:

- The VHDL-simulator supports the *foreign interface* and so allows the use of external c-functions.
- The DSP-simulator must provide entry-points for several functions like *simulate instruction.* All SuperSim simulators provide this functionality

So a cosimulation between VSS (Synopsys) and SuperSim can be done in the same way. Only the definition of the ports and the interfacing parts is different from the way it is done in Cyclone.

6 Conclusions and Further Research

We have shown that compiled simulators can easily be interfaced with VHDL-simulators by using the C++-classes developed in this project. The indirect cosimulation method is significant slower with small examples than the direct cosimulation. When the design matures this disadvantage is no longer significant and it is better to use the indirect cosimulation, thus allowing the processes to run on separate processors. Some aspects of cosimulation have not been covered. *Named Pipes* or *multi-threaded programming*. Concerning the simulation performance we expect the latter one to be between the direct cosimulation and the IPC based one. A common executable, where the DSP-simulator and the VHDL simulator are one binary executable, would be the fastest solution, but also the most inflexible one. Multi-threaded programming and direct cosimulation offers good simulation performance without neglecting the simulation performance. The most flexible solution is the cosimulation with means of IPC, but will only be superior if complex examples are addressed.

In the future the generation of DSP-simulators will more and more done automatically. Coming from a machine description[11] DSP-simulators will be generated including an interface for the cosimulation. With the easy way of integrating the simulator in the hardware environment presented in this paper, this will offer a very powerful environment for hardware/software Co-design. The turnaround times will slow down dramatically, thus enabling more powerful solutions per time-unit.

References

[1] V. Živojnović and H. Meyr, "Compiled HW/SW co-simulation," in *Proceedings of the Design Automation Conference (DAC)*, (Las Vegas), pp. 690–695, June 1996.

[2] V. Živojnović, S. Tjiang, and H. Meyr, "Compiled simulation of programmable DSP architectures," in *Proc. of 1995 IEEE Workshop on VLSI Signal Processing – Sakai, Osaka*, pp. 187–196, Oct. 1995.

[3] V. Živojnović, S. Pees, C. Schläger, R. Weber, and H. Meyr, "SuperSim - A new technique for simulation of programmable DSP architectures," in *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, (Boston), pp. 1748–1763, Oct. 1995.

[4] S. Pees, V. Živojnović, A. Ropers, and H. Meyr, "Schnelle Simulation des TI TMS 320C54x DSP," in *Proc. of DSP Deutschland 97*, (München), Oct. 1997.

[5] J. Rowson, "Hardware/software cosimulation," in *Design Automation Conference*, (San Jose, CA), pp. 439–440, Jun. 1994.

[6] A. Kalavade and E. A. Lee, "Manifestations of heterogeneity in hardware/software codesign," Design Automation Conference, 1994.

[7] S. Sutarwala, P.Paulin, and Y.Kummar, "Insulin: An instruction set simulation environment," (Ottawa - Canada), pp. pp. 355–362, CHDL, 1993.

[8] B. Schnaider and E. Yogev, "Software development in a hardware simulation environment," in *Design Automation Conference*, (Las Vegas), 6 1996.

[9] C. A. Valderrama, F. Nacubal, P. Paulin, and A. A. Jerraya, "Automatic generation of interfaces for distributed c-vhdl cosimulation of embedded systems: an industrial ex perience," in *7th International Workshop on Rapid Systems Prototyping*, (Greece), June 1996.

[10] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamoto, “A hardware-software co-simulator for embedded system design and debugging,” in *Asia South Pacific Design Automation Conference (ASP-DAC)*, 1995.

[11] S. Pees, A. Hoffmann, and H. Meyr, “Retargetable Timed Instruction Set Simulation of Pipelined DSP Architectures,” in *Proc. of DSP Deutschland 98*, (München), Oct. 1998.