A Fully Static Scheduling Approach for Fast Cycle Accurate SystemC Simulation of MPSoCs

Richard Buchmann
Université Pierre et Marie Curie, LIP6/UPMC
4 place Jussieu, 75005 Paris
France
Email: richard.buchmann@lip6.fr

Alain Greiner
Université Pierre et Marie Curie, LIP6/UPMC
4 place Jussieu, 75005 Paris
France
Email: alain.greiner@lip6.fr

***Abstract*—This paper presents principles and tools to facilitate *Multi-Processor System on Chip*s (MPSoCs) design and modeling, and to speed up cycle accurate SystemC simulation. We describe an effective way to build an hardware architecture virtual prototype, using a library of SystemC simulation models based on communicating synchronous finite state machines. This modeling approach supports a fully static scheduling strategy, based on the analysis of the combinational dependency graph. Our static scheduling algorithm has been implemented in the SystemCASS simulator, and provides speed-up of one order of magnitude versus the standard event-driven SystemC simulation engine. The modeling approach proposed in this paper has been adopted by the SoCLIB french national project, that is an open modeling and simulation platform for multi-processors system on chips.**

I. Introduction

Virtual prototyping of MPSoCs relies on efficient simulation tools, and requires more and more performance from the simulator, as system designers have to simulate embedded softwares execution onto the virtual hardware platform. The main goals of virtual prototyping are :

- Accurate performances evaluation, for both the timing and the power consumption;
- Embedded software debug, especially for efficient analysis of synchronization errors.

Therefore, most hardware/software co-design methodologies suppose to simulate complex software applications representing millions of cycles on complex multi-processors hardware platforms containing several tens of processors.

Important classes of bugs are impossible to detect without cycle accuracy. To get a precise performance analysis, we focus on Cycle Accurate and Bit Accurate (CABA) simulation models. So, the simulation speed is a major issue.

SystemC provides a modeling environment which enables the development and exchange of system-level C++ simulation models for re-usable hardware components. As most hardware description languages (including VHDL and Verilog), the SystemC core language relies on the event driven model of computation. This model is very general and supports a large class of simulation models, but the dynamic scheduling approach is known to slow down the simulation[1] speed.

This work has been supported in part by the french competitiveness cluster System@tic, in the framework of the Modrival sub-project.

In this paper, we propose a SystemC modeling approach supporting a fully static scheduling strategy that brings a simulation speed-up of one order of magnitude versus the standard SystemC simulator. This modeling approach is now used in the SoCLIB french national project : Eleven academic laboratories, and six industrial companies (including ST Micro-electronics, Thales, and Thomson joined to develop an open virtual prototyping environment for MPSoCs. The core of this platform is a library of fast SystemC simulation models for IP cores, using the modeling principles described in this paper.

In section II, we review some previous works. Section III describes the proposed modeling approach based on *Communicating Synchronous Finite State Machine* (CSFSM) model. Section IV presents the algorithm to build a fully static scheduling. Section V describes the experimental results.

II. Related Work

An MPSoC architecture is built by instantiation of re-usable, existing hardware components, such as processor cores, embedded memory banks, dedicated co-processors or peripheral controllers, and a communication micro network.

Each hardware component behavior is described as a set of parallel processes, communicating by signals. Those processes define how to compute output signals and the new internal component state, given input signals and the current internal component state. SystemC is the language commonly adopted to write those simulation models.

The standard SystemC simulator[2], distributed by the OSCI consortium, uses an event-driven simulation kernel, which schedules events at run time. The SystemC scheduler builds dynamically the processes evaluation order : Whenever a process writes into a signal, the scheduler produces an event and propagates it. Processes are triggered based on their sensitivity list. But it is well known that dynamic scheduling causes too many process wake-ups, hindering the simulation performance.

FastSysC[3] proposes a method to reduce the number of useless process wake-ups. At elaboration time, FastSysC builds an incomplete static scheduling, but without giving any guaranties to reach a stable state. The scheduler works in two steps :

1) Performs the incomplete static scheduling, at the first delta-cycles of each cycle;
2) Performs a dynamic scheduling, until the system is stable.

For each process, the designer can add the dependency informations between input ports and output ports. FastSysC exploits this information to build a combinational dependency graph, where the nodes are the signals, and the directed edges are the dependencies specified by the designer. If this graph contains cycles, FastSysC arbitrarily breaks them, and derives an initial ordering for process evaluation. No matter how wrong is this static scheduling, because FastSysC relies on the dynamic scheduling final step. With this method, the number of process wake-ups decreases significantly and the simulation speedup versus the standard SystemC simulator is about a factor 3.5.

CASS[4] is a Cycle Accurate System Simulator using a quasi-static scheduler. The hardware component are described in C language as a set of CSFSM. The behavior of each component is split into two separates functions :

- The sequential function computes the next internal state and the outputs signals that depends only on the current internal state.
- The combinational function computes the outputs signals that depends on inputs signals and the current internal state.

To define a quasi-static scheduling, CASS builds a graph, where the nodes are the processes (combinational functions), and the directed edges are the combinational dependencies between processes. As this process dependency graph can contain cycles, CASS introduces a relaxation mechanism, executing repeatedly all processes in a cycle until the system stabilizes. We implemented this strategy in our SystemC simulation environment. In many cases, CASS outperforms the standard SystemC simulator by a factor 7, but the simulation speedup drops dramatically when the number of cycles in the process dependency graph increases, because of the relaxation loops.

Some research works[5] focus on optimizing signal writings and register updating but the performance loss due to relaxation loops still remains.

In this paper, we tried to merge the best ideas of both the FastSysC approach (namely the combinational dependency graph analysis), and the CASS approach (namely the Communicating Synchronous Finite State Machines model) to propose a fully static scheduling approach for cycle accurate SystemC simulation.

III. Hardware Modeling with Communicating Synchronous Finite State Machines

All hardware modules in the system (or sub-system) are supposed to be clocked by the same system clock. Each hardware module is modeled by one or several CSFSMs. Each *Finite State Machine* (FSM) is defined by a set of states, a set of input ports, and a set of output ports. A state is defined by the values stored in the internal registers. The transition function defines the next state, and depends on both the inputs and the current state. The generation functions define the output values. An output port depending on at least one input is called a Mealy output. An output port that doesn't depend on any input is called a Moore output. As shown figure 1, the behavior is described by three types of functions, that are implemented as SystemC methods :

- Transition function : 1 per module;
- Moore generation : 1 per module;
- Mealy generation : 1 per Mealy output.

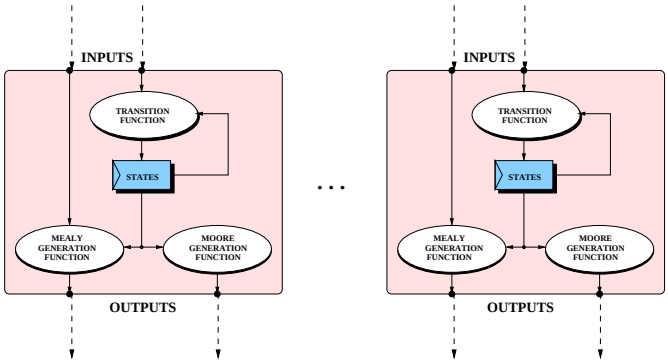

Fig. 1. Communicating Finite State Machine Modeling

Therefore, the complete hardware architecture is described as a set of CSFSM connected by signals. A Mealy signal is a signal connected to a Mealy output port. To simulate this architecture, it is necessary to compute at each cycle the new values of all signals, and the new state of all . The - cycle based - simulation loop can be split in three steps :

- Compute the new state of all CSFSMs, by evaluating all Transition functions. The signal values being stable, the evaluation order is not significant.
- Compute the new value of all Moore signals, by evaluating all Moore functions. The Moore signals depending only on the current FSM state, the evaluation order is not significant.
- Compute the new value of all Mealy signals, by evaluating all Mealy functions. For this step, the scheduling must handle the combinational dependencies between the Mealy signals.

We can force the event-driven SystemC simulation engine to respect this scheduling. The sensitivity lists for the three types of methods must be defined as follows :

Function kind	Sensitivity list
Transition Function	Positive clock edge only
Moore Generation	Negative clock edge only
Mealy Generation	Negative clock edge and all input ports involved in the combinational dependencies

TABLE I
Sensitivity lists for each type of method

For each hardware module, the designer must declare explicitly the combinational dependencies between input ports

and output ports. The easiest way is to define one Mealy function for each output port, and to define the combinational dependencies in the associated sensitivity lists.

Such simulation models can be efficiently simulated by the standard OSCI simulation engine, as most methods (all Transition functions, and all Moore generation functions) will be evaluated only once per cycle.

IV. Computing a Static Scheduling for Fast Simulation

As explained in the previous section, the only step requiring dynamic scheduling in the simulation loop is the computation of the Mealy signals. To obtain a fully static scheduling, we exploit the specific characteristics of the targeted MPSoCs architectures : Such architectures are built by direct instantiation of existing IP cores, and such hardware components have very few Mealy signals. For example, most available IP cores available in the SoCLIB library have zero Mealy outputs, to avoid long combinational paths and give the system designer some guaranties on the clock frequency. Therefore, the Mealy signals are only a small percentage of all the system signals.

We define the *Combinational Dependency Graph* (CDG), where the nodes are the signals, and the directed edges are the combinational dependencies between two signals. Each edge leading from a signal X to a signal Y is marked by a label identifying the Mealy function that drives Y.

Figure 2 shows a combinational dependency graph. A, B, C, and D are Mealy functions. $S1$ to $S7$ are signals. All signals that have no input edges are called sources : In this example, 1 and 4 are source signals.

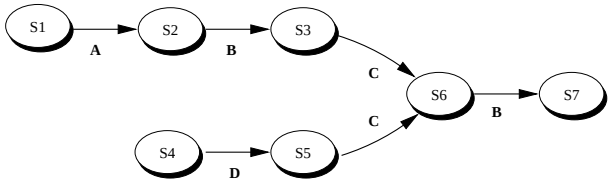

Fig. 2. Example : combinational dependency graph

As the hardware architecture should not contain combinational loops, the CDG must be acyclic. If it is not the case, the simulator stops and prints out an error message. The static scheduling algorithm analyzes the CDG, and builds a fixed evaluation order for the Mealy functions. This is not trivial, because, depending on the sensitivity lists, a given Mealy function can be attached to several edges. The algorithm proceeds as follow :

1) If there is a Mealy function F that only depends on source signals, we select it. Else, we select any Mealy function F that drives at least one signal depending exclusively on source signals;
2) We add the selected Mealy function F into the ordered list of functions to execute;
3) Delete all the edges, labeled F and exiting from a source node. Then delete all the nodes that have a degree of 0;
4) Repeat step 1 to 3 until the combinational dependency graph is empty.

This static scheduling has been implemented in the SystemCASS simulation engine.

V. Experimental Results

As the hardware components modeled with the CSFSM approach can be simulated by both the standard OSCI SystemC and the SystemCASS simulators, we want to compare the simulation speeds obtained with the two simulators. We used the standard SystemC 2.1.v1 simulation engine.

In this section, we present simulation results for two different hardware architectures :

- A clustered MPSoC architecture :
  A multi-threaded software application performing automotive obstacle detection by stereo-vision techniques is mapped on a clustered multiprocessor architecture;
- A Java processor micro-architecture :
  The detailed micro-architecture of a Java processor is modeled with the CSFSM approach, and executes a sequential ray tracer software application.

The workstation used for our simulations is a Pentium 4 at 2.80GHz.

A. Clustered multiprocessor architecture

The application is a pre-crash obstacles detection using stereo-vision[6] for automotive area, that requires intensive computation. This C software application is multi-threaded and runs on the massively parallel architecture. modeled with the SoCLIB IP cores library. It contains 8 clusters, as shown in figure 3, 30 general-purpose 32-bit processors and 750 K-bytes embedded memory.

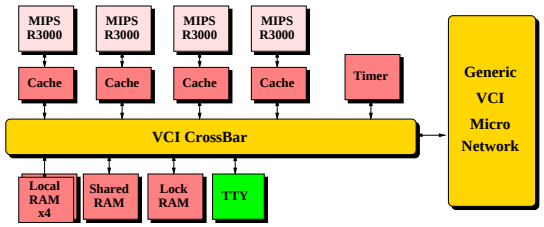

Fig. 3. Stereovision : cluster architecture

There are few combinational dependencies between signals in this architecture, and the longest path in the CDG has only 1 edge. Figure 4 shows the combinational dependency subgraph for one of the eight clusters. Therefore, the scheduling is very simple, as the Mealy functions can be executed in any order.

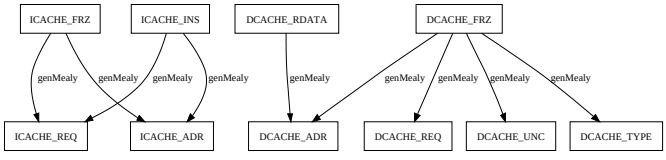

Fig. 4. Stereovision : combinational dependency subgraph

The table II shows the simulation speed for SystemCASS and SystemC simulators. SystemCASS is more than 10 times faster, and this speed-up is entirely related to the static scheduling approach.

SystemC 2.1.v1 OSCI *Dynamic scheduler*	**SystemCASS** *Static scheduler*
1 594 cycles/second	17 094 cycles/second

TABLE II
STEREO VISION : SIMULATION SPEEDS

B. Java Processor micro architecture

A ray tracer application implements a rendering algorithm in 3D computer graphics. The simulated architecture contains a Java processor[7], an instruction cache, a data cache, and an embedded RAM.

The JAVA processor micro architecture modeled as a set of interconnected hardware components, such as instruction decoder, execution unit, instruction folding unit, branch predictor and so on. The processor internal micro-architecture being precisely described, most of the instantiated components have a combinational behavior. The CDG contains about 100 signals (nodes), and 400 combinational dependencies (edges). The simulation engine has to schedule 33 processes, and the simulation speed strongly depends on the scheduler efficiency.

The table III shows the simulation speeds for SystemCASS and SystemC simulators. SystemCASS is 12 times faster than SystemC.

The processes dependency graph, shown in figure 5, contains several cycles. One cycle involves almost all the processes. Any dynamic scheduler will execute repeatedly those processes until the system stabilizes. Unlike those simulators, SystemCASS relies on the CDG, to compute an optimal static scheduling.

SystemC 2.1.v1 OSCI *Dynamic scheduler*	**SystemCASS** *Entirely static scheduler*
3 289.49 c/s	39 682 c/s

TABLE III
JAVA MICRO ARCHITECTURE : SIMULATION SPEEDS

VI. CONCLUSIONS

Our work focus on fast simulation of SystemC simulation models. It is well known that simulation speed drops dramatically when the number of combinational dependencies between signals increases. This paper presents both a new modeling approach, based on the CSFSM, and a new static scheduling algorithm, based on the *Combinational Dependency Graph* (CDG) analysis.

In the CSFSM modeling approach, the system designer describes each hardware component as a set of SystemC methods (Transition, Moore Generation, Mealy Generation), with the guaranty that most of the Transition and Moore Generation functions will be evaluated only once per cycle. For this reason, the CSFSM modeling is efficient with classical, event driven simulators, as it helps to reduce the number of processes wake-ups.

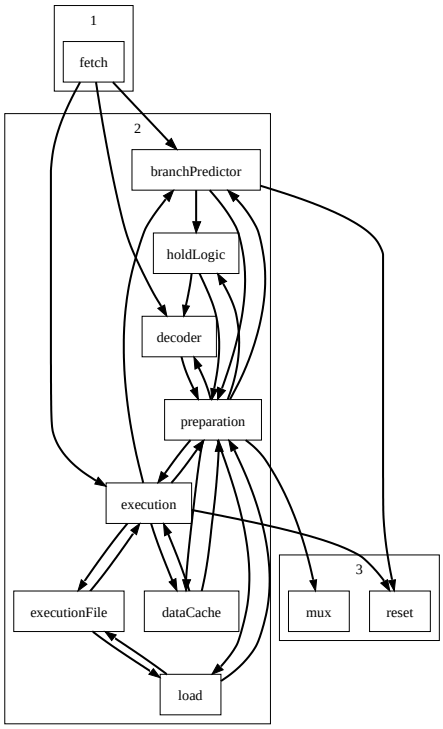

Fig. 5. Java processor : Processes dependency graph

The CSFSM modeling helps to identify the combinational dependencies between signals, which makes possible to define an optimal static scheduling by analyzing the CDG. The corresponding algorithm has been implemented in the SystemCASS simulator that outperforms the standard SystemC simulator (which uses a dynamic scheduling) by one order of magnitude.

REFERENCES

[1] A. Ki, "Internal report," in *Empirical Study of SystemC*. R&D Center,Dynalith Systems, Korea, 2003. [Online]. Available: http://www.dynalith.com

[2] *OSCI*, http://www.systemc.org.

[3] D. G. Perez, G. Mouchard, and O. Temam, "A new optimized implemention of the systemc engine using acyclic scheduling," in *Design, Automation and Test in Europe Conference and Exhibition Volume I (DATE'04)*, ALCHEMY INRIA Futurs & LRI, Paris South University. Paris, France: IEEE, 2004, p. 10552.

[4] F. Petrot, D. Hommais, and A. Greiner, "Cycle precise core based hardware/software system simulation with predictable event propagation," in *Proceeding of the* 23rd *Euromicro Conference*, ASIM/LIP6/UPMC. Budapest, Hungary: IEEE, Sept. 1997, pp. 182–187.

[5] R. Buchmann, F. Petrot, and A. Greiner, "Fast cycle accurate simulator to simulate event-driven behavior," in *Proceeding of The 2004 International Conference on Electrical, Electronic and Computer Engineering (ICEEC'04)*, ASIM/LIP6/UPMC. Cairo, Egypt: IEEE, 2004, pp. 35–39.

[6] A. Greiner, F. Petrot, M. Carrier, M. Benabdenbi, R. Chotin-avot, and R. Labayrade, "Mapping an obstacles detection, stereo vision-based, software application on a multi-processor system-on-chip," in *Proceedings of Intelligent Vehicles Symposium 2006*. IEEE, 2006, pp. 370–376.

[7] J. M. O'Connor and M. Tremblay, "picojava-i: The java virtual machine in hardware," *IEEE Micro*, vol. 17, no. 2, pp. 45–53, March 1997.