FAST CYCLE ACCURATE SIMULATOR TO SIMULATE EVENT-DRIVEN BEHAVIOR

Buchmann R., Pétrot F., Greiner A.
ASIM/LIP6, 4 Place Jussieu, 75252 Paris cedex 05, FRANCE
{buchmann;fred;alain}@asim.lip6.fr

Abstract - *Architectural exploration and application development for digital System On Chip need more and more performance from the simulator. Today, the standard design flow use a unified modeling language and only one simulator for every development step. SystemC based simulators are efficient to validate hardware specifications but its performances are not good enough to write and debug embedded softwares. Using some specialized simulators increase the effectiveness of a particular step. So, we specialize the simulator to become more suitable for targeted steps. In this paper, we describe an approach to accelerate simulation performances by focusing on the cycle accurate simulation level.*

I. INTRODUCTION

SystemC is a modeling language. Its key advantage is to allow the user to use various levels of abstractions under the same unified language. The SystemC simulator provided by OSCI is able to simulate mixed abstraction models. The cost is the use of a general simulation engine. The actual implementation is an event-driven simulator. Many researchers look through performance issues[1] to find the fastest subset of process/data types.

In the synthesis/implementation flow[2] shown figure 1, the embedded software developer and the RTL designer works separately early after the system design is completed.

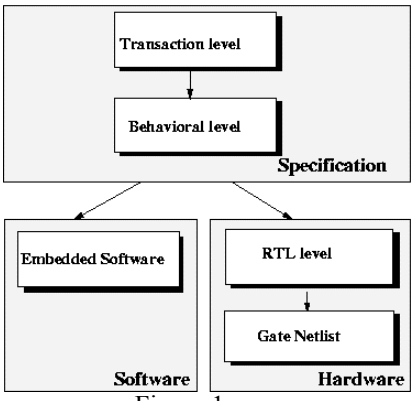

Figure 1.
SYNTHESIS/IMPLEMENTATION FLOW

As the complexity of application grows, the needs of on -chip multiprocessor, DSP, and a lightweight operating system increase. The software development becomes a huge part of the system-on-chip process time. Advanced programming tools lacks and debug time is incredibly high in many cases.

Systems we would like to model contain a small set of components (under 100). They are synchronous and potentially multi clocked.

We have done some profiles about the SystemC simulations to evaluate the kernel weight. The simplification of the SystemC engine allows us to increase the speed rate significantly if we define constraints about the way to write models. So, we would like to get the best simulation time using a particular abstraction level modeling.

The goals are to minimize the engine overhead, and to speed-up models execution, while keeping a simple way to design.

We choose the finite state machine with data path abstraction level (FSMD), cycle and bit accurate.

Firstly, we give a brief overview of the FSMD modeling to introduce cycle true simulation; we present a formalism to identify mandatory model writing constraints and describe a basic approach to check model descriptions; we suggest some advanced implementations to take advantages from the FSMD modeling. Secondly, we discuss some experimental results, and finally conclusions are presented.

II. SIMULATION ALGORITHMS

I. Event-driven simulation

Each component is described by one or several process bounded to an input port list. The list name is sensibility list. Any input port assignment generates an event call the process.

A simplified algorithm[3] is as follows:

- **Initialization**: Execute all process to initialize the system.
- **Execute**: Execute a set of process that are ready to run. Each assignment to a register or a signal sends an event that will be handled in the next step.
- **Update**: For each posted event e:
  - Update the corresponding register or signal.
  - Resume all the bounded process according the sensitivity list.
- Go to the **Execute** step while any event appears. The simulation time is increased by a delta.
- **Increase simulation time**: The system is stable. The simulator determines the next simulation time; jump to the Execute step.

This algorithm is very common and widely used in the hardware simulation (VHDL, Verilog, etc.).

The formal module model is a set of functions:

$\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \ldots \cup \mathcal{F}_n$,

Where:

- Each function $\mathcal{F}_i$ has the static sensitivity list *ei*.
- *I*, *O* and *S* are respectively inputs, outputs and states.

Let $tc \le t < t+d \le tc+1$ in,

$$S_{t+d} \Leftarrow_{ei} \mathcal{F}_i^s(S_t, I_t) \quad (1)$$

$$O_{t+d} \Leftarrow_{ei} \mathcal{F}_i^o(S_t, I_t) \quad (2)$$

II. FSMD modeling and Cycle Accurate engines

A system can be described as a set of synchronous FSMD connected using signals. For each component, we write one or several FSM(s).

Definition: A FSM is defined using a quintuple $\{I, O, S, \Theta, \Gamma\}$, where I is the input set, O the output set, S the state set, Θ the set of transition functions such that $\Theta : I \times S \to S$, and the set of generation functions such that either $\Gamma : S \to O$, for Moore FSMs, or $\Gamma : I \times S \to O$, for Mealy FSMs.

Three kinds of function describe an entire FSM: transition, Moore generation, and Mealy generation. Transition and Moore generation function are synchronous, called once at every cycle. Mealy generations are asynchronous, called until its inputs are stables. Several calls may occur at each cycle.

Functions above need to match all the following rules.

The FSMD model is a set of function:

$$\mathcal{F} = \Theta \cup MO \cup CF$$

Where Θ is transition functions; *MO* Moore generation functions; *CF* combinational functions.

Let $tc \le t < t+d \le tc+1$; d is the elapsed time after a delta cycle, tc is in number of cycles, *CK* is an arbitrary clock edge in,

$$S_{tc+1} \Leftarrow_{CK} \theta(S_{tc}, I_{tc}) \quad (3)$$

$$SO_t = SE(S_t) \quad (4)$$

$$CO_t = CB(S_t, I_t) \quad (5)$$

Where *SO* is the set of Moore outputs; and *CO* Mealy-only outputs: $O = SO \cup CO$.

III. Formal Model

The formal model of event-driven and FSMD simulation are slightly differents. Our objective is to define a set of additional constraints to be able to use both the event driven and FSMD approaches on the same model description.

We consider two main cases:

Clocked functions

If the static sensitivity list of $\mathcal{F}_i$ is ei = CK , this implies d = 1 in,

$$S_{t+1} \Leftarrow_{CK} \mathcal{F}_i^s(S_t, I_t) \quad \text{equivalent to (3)}$$

$$O_{t+1} \Leftarrow_{CK} \mathcal{F}_i^o(S_t, I_t)$$

Since the function $\mathcal{F}_i$ is called only one time at the beginning of the cycle, the input It acts like a register:

$$S'_{t+1} \Leftarrow_{CK} I_t \Rightarrow O_t = \mathcal{F}_i(S_t, S'_t) \text{ equivalent to (4)}$$

The function $\mathcal{F}_i$ describes either transitions or Moore generation functions.

Other functions

If the static sensitivity list o $\mathcal{F}_i$ is $ei = CK \cup I$, then $S_{t+d} \Leftarrow_{ei} \mathcal{F}_i(S_t, I_t)$

$$O_{t+d} \Leftarrow_{ei} \mathcal{F}_i^o(S_t, I_t) \quad \text{equivalent to (5)}$$

If the writing condition is CK on registers, we have:

$$S_{tc+1} \Leftarrow_{ei} \mathcal{F}_i^s(S_{tc}, I_{tc}) \quad \text{equivalent to (3)}$$

The function $\mathcal{F}_i$ describes either transitions or Mealy generations.

Writing Constraints

Finally, we need to put some strong constraints to stick to those two previous cases:

- Clocks are fully identified.
- All the used input ports appear in the static sensitivity list.
- The writing conditions on registers include clocks.

Using these constraints, we prove that we can translate from FSMD modeling to event-driven modeling and vice-versa.

IV. Validation of the Description

Before executing any system simulation, it is interesting to know if all the models follow the previous rules. Note that some languages provide a built-in validation efficient enough to validate the model description. In our SystemC case study context, a third party tools is needed to do this optional step.

Our approach is to use a simple syntax checker to analyze and validate the relationship of operands source/destination. The denied operations depend on the considered function.

Firstly, we enumerate all the functions and determine its kind: constructor, transition, Moore generation, and Mealy generation. Secondly, we build the variables dependency graph and check expression types.

We distinguish:

- Ports (sc_in, sc_out, sc_inout for examples);
- Registers (sc_signal);
- Architectural constants (data members, template parameters);
- Local variables;

We define a table I that describes four function types where some assignations are allowed and others not. Any local variable uses are not restricted.

Syntax checking is useful to validate the FSMD model writing. The basic approach is good enough to avoid mistakes and compatibility problems.

Table I
Allowed Operations

Function	Result type	Operand Types
Constructor	Constants	Constants
Transition	Registers, Constants	Inputs, Registers, Constants
Moore	Outputs	Registers, Constants
Mealy	Outputs	Inputs, Registers, Constants

V. Advanced Optimizations

The use of the FSMD models on an event-driven simulator significantly reduces the number of scheduled events at least cost and provides greater performances. While a basic FSMD simulation engine gives some good results, we have found a number of optimizations that are useful to produce a more efficient simulation. The FSMD modeling has some constraints allow us to simplify the simulator:

- Lightweight read/write primitives
- Static scheduling [4] of asynchronous functions
- Fast relaxation algorithm because Mealy function granularity is very small and occurs seldom at system wise level

Lightweight read/write primitives

Each read/write operations target two signal tables. The first one holds the current values; the second one holds the new values. At the end of synchronous calls, the engine copies the second one into the first one. During the Mealy computations, read/write operations target the same table.

Partial Static scheduling

We build a directed graph (figure 4). Each node is a signal. Each arrow A to B tells about a data dependency: B depends on A. We perform a topological sort and extract the strong component to produce an ordered static scheduling.

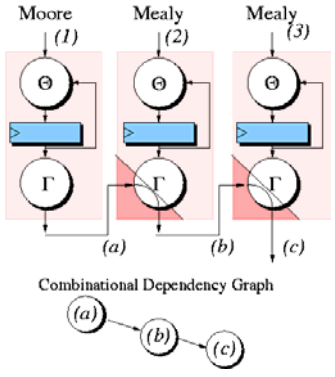

Figure 4.
FSMs without combinational cycle

The simulation kernel calls each combinational function only once time without using any scheduling overhead.

Dynamic scheduling

When the graph is cyclic (figure 5), we need to iterate simulation until all signals are stable. Complex dynamic schedulers cost is too high because combinational functions are very small. So, we use a simple loop to minimize overheads. At the loop beginning, the flag unstable holds false. Each Mealy generation function is called. When a new value is written to any output port, the flag switches to true. The loop ends when the flag unstable stays false after the iteration.

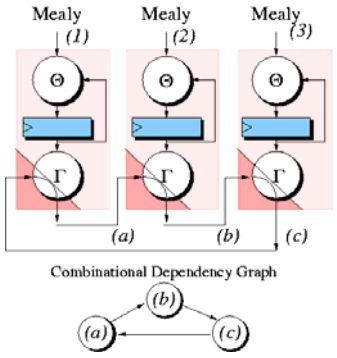

Figure 5.
FSMs with a combinational cycle

Two main semantic differences exist between the event-driven and the FSMD approach.

The first one is about clocks. Since the new proposed engine is Cycle Accurate, clocks meanings are slightly differents. Clock signals/ports haven't any value and the system/model designer has to identify them explicitly using a dedicated syntax. Some hand-written simulation loops are incompatibles: clocks should be used only to drive data register inputs.

The second one is about sensitivity lists. In the event-driven context, some variables or output ports may have a register behavior. FSMDs modeling principles presented above put some constraints on the register accesses. The consequences are that the static scheduler gives no guaranties for users to get the same execution order than the event-driven scheduler. System behaviors could be differents if the sensitivity list is not complete.

The event-driven modeling gives a syntax allowing a lot of mistakes. Commercial tools use only a small subset from the initial language reference. By the same way, the FSMDs modeling avoid some mistakes by using a well-known formal model that allows accurate hardware modeling.

VI. Results

We use the SystemC language to develop four VCI[5] models: a MIPS R3000, a data cache, an instruction cache and a simple RAM. These models belong to SOCLIB[6] library.

We use only a small part of the language reference [7]. SC_THREAD, SC_CTHREAD[1], and hardware data types aren't implemented.

The syntax checker, built upon g++ front-end, validates all the models. Note that the analysis time is very short and optional.

To select the simulator engine, either SystemC 2.0 or our implementation called SystemCASS, we change only the include/library directories. Otherwise, all the models are strictly identical. The software application running on our system is a set of simple algorithms: Fibonacci, factorial, memory copy, ... The table II indicates the simulation performances using the both engines. The compiler is g++ 3.0.4. The system ends after 12 million cycles. The elaboration phase is excluded from analyzed time.

Table II
Allowed Operations

SystemC Engine	SystemC 2.0	SystemCASS
Simulation time (in s)	163	26
Performance (in cycles/s)	76k	474k

The following tables III and IV give a summary of the simulation profiles. The first table shows that the SystemC 2.0 kernel cost is very high: half of execution time is spent into the kernel. Update is one of the most costly steps.

SystemCASS saves a lot amount of time by using a lightweight kernel.

Table III
Kernel Weight

SystemC Engine	SystemC 2.0	SystemCASS
Simulation time (in s)	70.1	35.7
Cumulative time in methods (in cycles/s)	34.3	27.2
Cumulative time in kernel (in cycles/s)	35.8	8.5

Note that this example doesn't underline performance lost by combinational loop. We can observe only the advantage of the lightweight primitives and the static scheduling. Many designs have no combinational loop anyway.

Table IV
Time spent in Models Evaluation

SystemC Engine	SystemC 2.0	SystemCASS
Instruction Cache (in s)	55	26
Data Cache (in s)	66	42
MIPS R3000 (in s)	40	27
RAM (in s)	61	32

We wrote a second example to profile the combinational cost: the system got nine modules including two separate combinational loops. Each method is reduced to one or two simple assignments.

SystemC evaluates up to 3 times some combinational methods and may not evaluate at all if the input ports are stable. SystemCASS always call each method at least one time but the total number of iterations is lesser.

The SystemCASS initialization needs to elaborate (at execution time) the net list description first, compute the static scheduling, write the C++ code, compile and link dynamically. This initialization is more costly than the original SystemC simulator but it tends to be very quick even for short simulation duration.

VII. Conclusions

SystemC goals are to unify the different modeling levels under the same language but this forces the simulation kernel to be very general [2]. This choice doesn't allow some optimizations to increase performances in specific contexts. When the abstraction level is higher, the models are more complex, faster but the kernel weight increases.

Our approach is to define an abstraction level appropriated to the embedded software development. The cycle-based SystemC simulator we implemented is more than four times faster than the official one; and more suited to embedded software design and architectural exploration. In practice, SystemCASS is integrated easily in a design flow and provides a great utility value in some early stage. A hardware synthesis tool, like UGH designed by LIP6[8], is able to translate a C functional model into a cycle accurate model. SystemCASS may find a large use in SoCLIB[6] for University purpose.

VIII. Further work

As scheduling cost is low and the engine is very lightweight, we don't expect to increase again the overall performance. So, the next work is about enhancing the modeling domain to get the best of FSMD simulators using others various level of abstractions and/or basically others syntax. Some third party tools could be useful to debug models, system architectures, and softwares; to validate FSMD modeling; to trace result values; or to check format compatibility.

IX. References

[1] Ando Ki, “Empirical study of systemc”, *R&D Center,Dynalith Systems*, Korea, 2003.

[2] Preeti Ranjan Panda. “Systemc”. *In ISSS*, pages 7580, 2001

[3] Robert S. French, Monica S. Lam, Jeremy R. Levitt, and Kunle Olukotun. “A general method for compiling event-driven simulations”. *In Design Automation Conference*, pages 151156, 1995.

[4] Frédéric Pétrot, Denis Hommais, and Alain Greiner. “Cycle precise core based hardware/software system simulation with predictable event propagation”. *In Proceeding of the 23rd Euromicro Conference*, pages 182 187, Budapest, Hungary, September 1997. ASIM/LIP6, IEEE.

[5] “Virtual Component” , *http://www.vsi.org*. Interface Standard.

[6] SoCLIB, “A modelisation & simulation platform for system on chip”, http://soclib.lip6.fr , 2003.

[7] OSCI, “SystemC 2.0.1 Language Reference Manuel”, http://www.systemc.org, 2003.

[8] LIP6, “Digital system design environment”, http://www-asim.lip6.fr/recherche/disydent.

[1] deprecated since SystemC 2.0.1