Parallel Simulation of Systems-on-Chip Models at the Transaction Level

JURY members:

- Christian Boitet, President (Clips-UJF)
- Jean-Luc Dekeyser, Reviewer (Lifl-LILLE)
- Frédéric Pétrot, Reviewer (Tima-INPG)
- Laurent Maillet-Contoz, Examiner (STMicroelectronics)
- Florence Maraninchi, Advisor (Verimag-INPG)
- Pascal Raymond, Co-advisor (Verimag-CNRS)
- Emmanuel Dufour, Advisor (Silicomp-Orange)
Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

Parallel execution preserving the semantics

Algorithm and implementation

Experiments and results

Conclusion and further work
Systems on Chip (SoC)

Hardware components (IP) : processor(s), memories, DMA (Direct Memory Access), bus ...
+ embedded software (OS + Applications)
Testing the embedded software on the physical chip

- Fast
- Accurate
- "Black box"
- Bug => high cost
Simulating the embedded software on an accurate hardware description is too slow and the time to target is too long. However, it is possible to synthesize an accurate model of the hardware description.
Simulating the embedded software on an abstract hardware model

- Embedded software

+ Simulating the embedded software on an abstract hardware model

+ Abstract model of the hardware

+ D = read(A);
  transform(D);
  write(D, A);

- Hardware-software co-design
  - No automatic synthesis

- MEMORY
  - UART
  - DMA
  - CPU

- BUS

DRT - Yussef Bouzouzou
**Transactional Model (TLM): principle**

**transaction** = atomic exchange of data between modules

A concept of **port** (component interface)

a **bus** (transport addresses and data)

a **communication protocol** (e.g., read(), write())

```
D = read(A);
transform(D);
write(D, A);
```
SystemC : C++ library for the modeling of embedded systems

\begin{verbatim}
SC_MODULE(top){
    unsigned x;
    sc_event e;
    SC_HAS_PROCESS(top);
    SC_CTOR(top){
        bus.port(M.port);
        SC_THREAD(P);
        SC_THREAD(Q);
    }
    void top::P() {
        ... wait(e); ... }
}
\end{verbatim}

The whole C++ language, plus:
- modules and ports classes describing the architecture,
- Processes and events describing the behavior,
- A concept of simulation time,
- A simulation engine: non-preemptive (collaborative) scheduler
Official execution semantics

SystemC model + SystemC Simulator kernel = Sequential program

Non-preemptive scheduler

non-preemptive: SystemC processes decide when to yield back to the scheduler

non-determinist norm: many possible and valid schedules

void top::P()
{
    while(true)
    {
        cout << "p" << endl;
        wait(2, SC_SEC);
    }
}

void top::Q()
{
    while(true)
    {
        cout << "q" << endl;
        wait(2, SC_SEC);
    }
}

Set of valid behaviors (wrt the IEEE standard)

DRT - Yussef Bouzouzou
Objective

Speed up SystemC/TLM simulations on multiprocessors computers (SMP)
Do not change the semantics (IEEE-1666 standard)
Without manual patches nor annotations of the model

≡? : we need to decide
What we want to compare
When we compare
Semantics-preserving parallel execution

Comparison criteria $\equiv$:
The same outputs on the screen,
The memory state (= variable values)

When?:
always (asm instruction): no way to parallelize
At delta-cycle borders:

\[
\begin{array}{c}
\equiv \\
\equiv \\
\equiv \\
\equiv \\
\end{array}
\]
Using multiprocessor machines
Using multiprocessor machines

SystemC processes | OS processes | Processors

SystemC kernel | OS scheduler
Using multiprocessor machines

SystemC processes

OS processes

Processors

SystemC scheduler

OS scheduler
Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

Parallel execution preserving the semantics

Algorithm and implementation

Experiments and results

Conclusion and further work
Transition, action, and interleaving

**Transition**: smallest atomic section w.r.t. SystemC

**Action**: smallestset atomic section w.r.t. the OS

**a||b**: the interleavings of the actions a and b

```cpp
void top::P(){
g = 0;
wait(2, SC_SEC); // S1
g++; // a1
g++; // a2
}

void top::Q(){
valeur = 0;
wait(2, SC_SEC); // S1
valeur = g ;  // b
}
```

- **S1**: value = 0, g = 0
- **S2**: value = 2, g = 2
- **S3**: value = 0, g = 2
- **S4**: value = 1, g = 2

Invalid global state

DRT - Yussef Bouzouzou
Independent tasks: consequences for parallel executions

S is a state. A and B are two tasks to be executed from S. A and B are **independent** if they do not share **resources**. 

Consequence: all interleavings lead to a same state S'.

A and B are independent $\Rightarrow$ A $\parallel$ B is valid, w.r.t. the IEEE standard.
Independent tasks: consequences for parallel executions

S is a state. A and B are two tasks to be executed from S. A and B are independent if they do not share resources. consequence: all interleavings lead to a same state S'.

A and B are independent => A || B is valid, w.r.t. the IEEE standard.
Independent tasks: consequences for parallel executions

S is a state. A and B are two tasks to be executed from S. A and B are **independent** if they do not share **resources**

consequence: all interleavings lead to a same state $S'$

**A and B are independent** $\Rightarrow$ A $\parallel$ B is valid, w.r.t. the IEEE standard
Task structural independance

Based on the architecture of the model (i.e., the SystemC modules)
Two tasks of distinct modules are independent

1) Philippes Combes (ST Genève PhD):
    - **clusters**
    - modify the language (new `sc_node_module` concept)
    - **Static partition** of the independent task done by the developer

2) Eric Paire (ST Grenoble):
    - **Multicore machines (SMP)**
    - SystemC models with **synchronous channels**
    - A prototype working on TLM models, but introduce **new preemption points**
    - Do not preserve the **semantics**
Structural independence and transactions

The transactions allow the processes to access data hosted by other modules. Thus, transactions break the structural independency rule.
Many possible level for dependency analysis

If we look closer, we can parallelize more...

...but the analysis is harder
Our solution

Semantics-preserving parallelization requires dependency analysis
The best observation level for dependency analysis is the transition

Runtime algorithm
New scheduler for parallel simulation
Use both the Pthread (POSIX) library and the QuickThread library
Protect shared memory internal to SystemC with mutexes and condition variables
Use a function \( \text{indep}(t_1, t_2) \):
true iff \( t_1 \) and \( t_2 \) do not share any resource

Combined with a static algorithm
Static dependency analysis to provide the \( \text{indep}(t_1, t_2) \) function
Identify the places where processes may yield back to the scheduler
Map each yield place to a list of maybe-accessed resources
Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

Parallel execution preserving the semantics

Algorithm and implementation

Experiments and results

Conclusion and further work
Sequential scheduler

**Eligible** : set of all eligible processes

```plaintext
while Eligible ≠ Ø
    choose e ∈ Eligible
    execute(e);
```

DRT - Yussef Bouzouzou
Parallel scheduler

One « scheduler » per PThread
Each « scheduler » executes the same algorithm

- **Eligible**: set of all the eligible processes
- **Running**: set of all the processes being executed

while Eligible ≠ Ø
choose e ∈ Eligible
If for all r ∈ Running
Indep(e, r)
then
execute(e)

**Eligible** and **Running** protected by a mutex
Mutex released during `execute(e)`
Another tool must provide the function `Indep(e, r)`
# Static analysis

```c
sc_event e;
int x, y, z, t, u;
int * T;
void run()
{
    do {
        wait(e);
        x++;
        y = y-T[x];
        f();
        y = T[0];
    } while (x<100);
}

void f()
{
    if(z >= 2) {
        x = 0;
        g();
        wait();
    } else {
        wait();
        y = 0;
        g();
    }
}

void g()
{
    t = 0;
    wait();
    g();
}
```

DRT - Yussef Bouzouzou
Static analysis

```c
sc_event e;
int x, y, z, t, u;
int * T;

void run()
{
    do {
        wait(e);
        x++;
        y = y - T[x];
        f();
        y = T[0];
    } while (x < 100);
}

void f()
{
    if (z >= 2) {
        x = 0;
        g();
        wait();
    }
    else {
        wait();
        y = 0;
        g();
    }
}

void g()
{
    t = 0;
    wait();
    u++;
    return?
}
```

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x, y, T, z, t</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>???</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>y, t</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>u + ???</td>
<td>yes</td>
</tr>
</tbody>
</table>
Using the static analysis data during runtime

At runtime:

Run: 5 +

\[ \{ u \} \cup \emptyset \cup \{ x, y, T, e \} \cup \emptyset \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>x, y, T, z, t</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Ø</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>y, t</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>u</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>x, y, T, e</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>Ø</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>Ø</td>
<td>yes</td>
</tr>
</tbody>
</table>
Architecture of the parallel kernel

- SystemC model
- C++ files
- Parallel simulation

Dependency tables

C++ files

numbered transitions

Pinapa Front-end SystemC

Dependency analysis

AST

Back-end Static analysis

Parallel scheduler

Parallel simulation

C++ files

Dynamic analysis

Parallel simulation

C++ files

Hypergraph

Dependency tables
Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

Parallel execution preserving the semantics

Algorithm and implementation

Experiments and results

Conclusion and further work
Experiments

Examples provided with the OSCI SystemC library
Not realistic enough, no at the TLM level
Allow to validate the parallel simulator

Examples provided by STMicroelectronics
Intrinsically sequential...
...or too much complicated to be analyzed by hand

Our case study: « Mandelbrot fractal »

Mathematic function
Result can be displayed as an image
Each pixel can be computed independently
Parallel computation of distinct parts
Possible to parametrize the number of parts
Case study: «Mandelbrot fractal»
Results

Addresses of the transaction: `port.read(address);`
It is impossible to compute statically a transaction address in the general case
A naive over-approximation is too strong:
Any target is possibly accessed (⇒ no parallelization)

The « RAM memory » (Tac_memory) :
All memory accesses are considered as dependent, because the RAM memory is considered as a whole
But in general each initiator accesses a different part of the memory

Loose timing annotations with randomization (pv_wait(50,SC_SEC)) .

Results : 4 processors, 5 Pthreads
4, 8, 16, 160 instances of the « Fractal » component
The parallel simulator is about 3 times as fast as the sequential one.
conclusion and further work

We proposed an algorithm for a **semantics-preserving SystemC parallel** implementation of a **parallel SystemC simulator**

Fully working on one class on SystemC models

(synchronous channels, no transactions)

**Static analysis algorithms** and **architecture** of a parallel SystemC framework

Identification of the efficiency issues
conclusion and further work

Decoding transactions addresses
Numeric abstract interpretation
Additional data provided by the user

Partition of the RAM memory: Idem

pv_wait (delay, unit) :
Select a « favorable » value

Modify the TLM library:
Non-atomic transaction (mp_read(), mp_write()...)
Only the bus need to be modified