Transaction Generator 2 Tutorial

Updated: September 15, 2010

Esko Pekkarinen
Department of Computer Systems
Tampere University of Technology

Contents

1 GETTING STARTED

1.1 Introduction

Transaction Generator 2 is a SystemC based benchmarking tool for network-on-chips [1]. It can be used to simulate abstract models and platforms described in RTL or higher levels of abstraction and collect various statistics about system performance. The system model is divided into three separate segments as shown in figure 1. Refer to [2] for more information on application modelling and network-on-chip benchmarking.

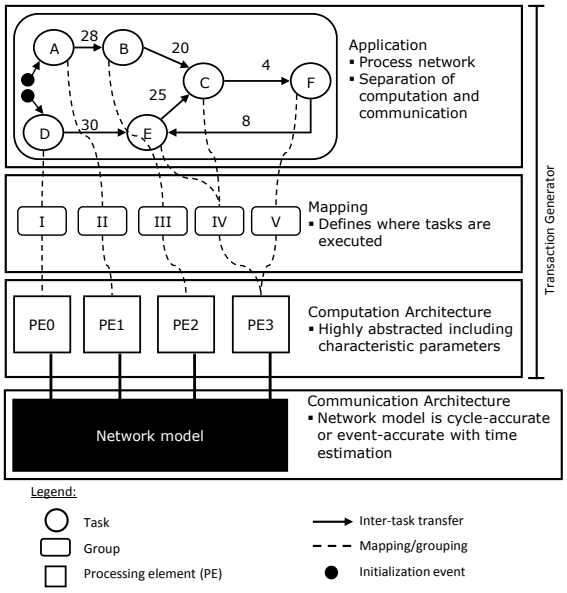

Figure 1: *Conceptual view of Transaction Generator*

This tutorial introduces the usage of Transaction Generator 2 and Execution Monitor [3] by demonstrating some of their properties. It should take less than two hours to complete. The example system is a basic 2x2 mesh with four identical processing elements (PEs) executing a simple application.

1.2 TG package and compilation

If Transaction Generator 2 (TG) is already installed on your computer, proceed to section 1.3.

The Transaction Generator 2 package includes the source codes for the simulator program and scripts for compilation. Before proceeding, make sure your computer has the following tools and libraries.

Tool	Purpose	Notes
Make	Compilation and simulation automation	Tested with version 3.80
gcc	Compilation	Tested with version 3.4.3
Boost c++ libraries	TG implementation	At least version 1.42.0 is needed.
SystemC	HW description in C++	Tested with version 2.2.0
OSCI TLM 2.0	TLM extension for SystemC	Tested with version 2.0.1

Start by entering the directory TG is extracted to.

```
$ cd /path-to-tg/
```

The directory contains the following files and directories.

bin/	Location for executables
COPYING	GNU General Public License
COPYING.LESSER	GNU Lesser General Public License
examples/	Example XML models
execution_monitor/	Execution Monitor source files
hw_lib/	Network-on-Chip hardware models
adapters/	Example OSCI TLM adapter
fifo/	Fifos used by Mesh_2D
mesh_2d/	Mesh_2D NoC models
noc_factory/	Source code to modify for adding NoC models
packet_codec/	Packet codec used by Mesh_2D
simple_bus/	Simplified OSCI TLM shared bus model
java_tool_installer/	Java libraries for Execution Monitor
lib/	Support libraries for executables
Makefile	Main makefile
Makefile.env.tmp	Template for environment depended variables for Makefile
README	Project info and known error messages
scripts/	Auxiliary scripts
transaction_generator_2/	Main simulator source files
work_libs/	Working directory for Modelsim

Before the compilation certain enviroment specific changes must be applied for the scripts to function properly. Rename or copy Makefile.env.tmp as Makefile.env.

```
$ cp Makefile.env.tmp Makefile.env
```

Open Makefile.env using a text editor, for example Emacs.

```
$ emacs Makefile.env
```

Edit the paths on the following lines to match your directory structure. Note that the paths should be relative to the location of Makefile.env.

```
BOOST_INC     = ???/boost_1_42_0/include
BOOST_LIBS    = ???/boost_1_42_0/lib
TLM_INC       = ???/tlm-2009-07-15/include/tlm
SC_INC        = ???/systemC/include
```

Now compile the simulator program using `make`. This might take a few minutes.

```
$ make
```

`sctg` (or `sctg.exe`) simulator executable is now ready in the working directory. It is needed later for the simulation.

1.3 Application XML source file

The application and the platform network-on-chip (NoC) are described in an XML file which is automatically parsed by the TG before running. Another XML file defines the properties of the processing elements. The application XML consists of the following sections: application, mapping, platform and constraints.

Application describes the load of computational tasks and communications which mapping assigns to hardware for execution. Platform describes the network along with its hardware resources. Finally, constraints specify the simulation and statistics parameters for TG. An example source file is explained in sections 2, 3 and 4. Sections 5 and 6 describe how to run the simulation and view the statistics.

The complete XML documentation [4] is available at NocBench project's web site:
`http://www.tkt.cs.tut.fi/research/nocbench/`

2 MODELING THE APPLICATION

2.1 Application

The application section defines the computational and communicational behavior of the system which is generated using tasks and events. Kahn process networks can be used to visualize the functionality, although TG enables also more versatile and detailed modeling.

The example application of this tutorial is presented in figure 2. It has 4 tasks that model the computation performed by the PEs. Arrows denote the communication between them. When a task receives a data token, it starts "computation" and after that it emits tokens to others. The start and end points of the arrows are here referred to as ports.

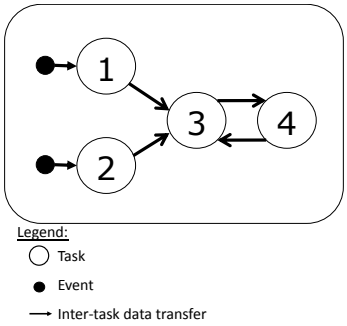

Figure 2: *The process network of the example application*

2.1.1 Events

Events are used to model the simulus from the environment. They generate and send data tokens to tasks via unidirectional ports. The tokens received by tasks activate the functionality in the recipient task whereas events are time based. An activated event sends a fixed number of bytes to the given output port. The starting time of the first transmission (offset) is by default zero, but can be easily adjusted. At least one event is required to start the application. The list of events in our example application is given below.

Listing 1: *Events in the example*

```
<event_list>
  <!-- 1.event  -->
  <event name="ActivateSendOnce"
         id="1"
         amount="8"
         count="1"
         offset="0.050"
         prob="1.0"
         out_port_id="70" />

  <!-- 2.event  -->
  <event name="ActivateSendPeriodically"
         id="2"
         amount="1"
         count="25"
         period="0.004"
         prob="1.0"
         out_port_id="80" />
</event_list>
```

The first event is used to activate a single task once. 50 ms (=0.050 s) after the start of the simulation it sends 8 bytes to port 70 triggering task "SendOnce". Since the amount of transmissions is only one, it's unnecessary (yet, possible) to define the period between transmissions.

The second event activates another task every 4 ms by sending 1 byte to port 80. After 25 transmissions the event stays idle for the rest of the simulation. If the transmission count isn't defined, the event will send unlimited times by default.

2.1.2 Tasks

Tasks are the primary method for modeling the computational load and the amount of communication. They activate on the incoming data token sent by events or other tasks according to their trigger condition. Every task has at least one trigger which reacts to data in certain port(s) and then executes computational operations and/or sends data to another task(s). "Computation" is implemented simply by waiting for a certain period of time.

The tasks in the example application are described next. The tasks are created so that they demonstrate the most common usage possibilites.

Listing 2: *Task 1: very simple task*

```
<!-- 1.task -->
<task  name="SendOnce" id="1" class="general">
  <in_port id="11"/>
  <out_port id="10"/>
  <trigger>
    <in_port id="11"/>
    <exec_count>
      <op_count>
        <int_ops>
          <polynomial>
            <param value="1500000" exp="0"/>
          </polynomial>
        </int_ops>
      </op_count>
      <send out_id="10" prob="1">
        <byte_amount>
          <polynomial>
            <param value="1024" exp="0"/>
          </polynomial>
        </byte_amount>
      </send>
      <next_state value="READY"/>
    </exec_count>
  </trigger>
</task>
```

The first task is activated when a data token arrives in port 11. After executing 1500000 integer operations it send 1024 bytes to port 10. The ports will be connected to other tasks later.

Listing 3: *Task 2 introduces memory operations*

```
<!-- 2.task -->
<task  name="SendPeriodically" id="2" class="general">
  <in_port id="21"/>
  <out_port id="20"/>
  <trigger>
    <in_port id="21"/>
    <exec_count>
      <op_count>
        <mem_ops>
          <polynomial>
            <param value="3000" exp="0"/>
          </polynomial>
        </mem_ops>
      </op_count>
      <send out_id="20" prob="1">
        <byte_amount>
          <polynomial>
            <param value="16" exp="0"/>
          </polynomial>
        </byte_amount>
      </send>
      <next_state value="READY"/>
    </exec_count>
  </trigger>
</task>
```

Tasks can execute three different types of operations: integer, floating point and memory operations. The second tasks executes 3000 memory operations every time it receives a data token in port 21. After the execution, it sends 16 bytes is to port 20.

Listing 4: *Task 3 with polynomial data amount and multiple triggers*

```
<!-- 3.task -->
<task  name="ForwardData" id="3" class="general">
  <in_port id="31"/>
  <in_port id="32"/>
  <in_port id="33"/>
  <out_port id="30"/>
  <trigger>
    <in_port id="32"/>
    <exec_count>
      <op_count>
        <int_ops>
          <polynomial>
            <param value="16284" exp="0"/>
          </polynomial>
        </int_ops>
      </op_count>
      <send out_id="30" prob="1">
        <byte_amount>
          <polynomial>
            <param value="4" exp="1"/>
          </polynomial>
        </byte_amount>
      </send>
      <next_state value="READY"/>
    </exec_count>
  </trigger>
  <trigger dependence_type="and">
    <in_port id="31"/>
    <in_port id="32"/>
    <exec_count>
      <op_count>
        <int_ops>
          <polynomial>
            <param value="16384" exp="0"/>
          </polynomial>
        </int_ops>
      </op_count>
      <send out_id="30" prob="1">
        <byte_amount>
          <polynomial>
            <param value="512" exp="0"/>
          </polynomial>
        </byte_amount>
      </send>
      <next_state value="READY"/>
    </exec_count>
  </trigger>
</task>
```

Tasks can contain more than one input or output ports and multiple triggers can be defined for a single task. Task 3 has three input ports, one output port and two separate triggers. The first trigger is similar to the previous except that it sends out a number of bytes relative to the received bytes. In this case data arrives from the second task. Hence, 16 bytes arrive every 4 ms and 4*16=64 bytes is sent after 16384 integer operations have been completed.

The second trigger is activated when data has been received in both ports 31 and 32. Since port 31 receives data from the first task, this trigger is activated only once during the simulation. Data arriving in port 32 naturally activates the upper trigger as well.

Listing 5: *Task 4 with differing functionality and distributional data amounts*

```
<!-- 4.task -->
<task  name="DataProcessing" id="4" class="general">
  <in_port id="41"/>
  <out_port id="40"/>
  <trigger>
    <in_port id="41"/>
    <exec_count mod_phase="0" mod_period="2">
      <op_count>
        <float_ops>
          <distribution>
            <uniform min="16384" max="32769"/>
          </distribution>
        </float_ops>
        <mem_ops>
          <polynomial>
            <param value="1500" exp="0"/>
          </polynomial>
        </mem_ops>
      </op_count>
      <next_state value="READY"/>
      <send out_id="40">
        <byte_amount>
          <polynomial>
            <param value="16" exp="0"/>
          </polynomial>
        </byte_amount>
      </send>
    </exec_count>
    <exec_count mod_phase="1" mod_period="2">
      <op_count>
        <float_ops>
          <distribution>
            <uniform min="1200" max="2600"/>
          </distribution>
        </float_ops>
      </op_count>
      <next_state value="READY"/>
      <send out_id="40">
        <byte_amount>
          <polynomial>
            <param value="8" exp="0"/>
          </polynomial>
        </byte_amount>
      </send>
    </exec_count>
  </trigger>
</task>
```

The functionality of a trigger can also vary according to how many times it has already been executed. This is set by the <exec_count> tag. In previous tasks no attributes are given and hence every time similar execution begins. In task 4 there are two variations to the execution. A modulo operation selects which one is executed. In addition, the number of operations is partially randomized.

When the number of trigger activations is even, a random number of floating point operations followed by 1500 memory operations are executed before sending 16 bytes to port 40. The number of floating point operations is a random number between 16384 and 32769 with uniform distribution. When the number of trigger activations is odd, a smaller amount of floating point operations are executed and only 8 bytes are sent.

2.1.3 Task connections

Finally, the tasks and events are connected by their individual ports. Port id numbers can be chosen freely, but each number must be unique. In this example every task has exactly one output port. Task 3 has three input ports, whereas the others have only one.

Listing 6: *Task conncections*

```
<!-- Connect the task and event ports  -->
<!-- 1.event to 1.task -->
<task_connection src="70" dst="11"/>

<!-- 2.event to 2.task -->
<task_connection src="80" dst="21"/>

<!-- 1.task to 3.task -->
<task_connection src="10" dst="31"/>

<!-- 2.task to 3.task -->
<task_connection src="20" dst="32"/>

<!-- 3.task to 4.task -->
<task_connection src="30" dst="41"/>0

<!-- 4.task back to 3.task -->
<task_connection src="40" dst="33"/>
```

3 MODELING THE PLATFORM

3.1 Platform

This section defines the PEs, the NoC and assigns the tasks of the application to separate PEs for execution.

3.1.1 Resources

This segment specifies the used PEs and some of their properties. Most of the PE's attributes are defined in the processing element library file which is selected in the constraints section. However, in order to enable unambiguous mapping, a type, name and an id must be given to each PE. The type has to match a defined type in the PE library.

Listing 7: *Resources*

```
<resource_list>
  <resource id="0" name="cpu1" frequency="20"
            type="CPU_TYPE_2" packet_size="16">
    <port terminal="0"/>
  </resource>

  <resource id="1" name="cpu2" frequency="20"
            type="CPU_TYPE_2" packet_size="16">
    <port terminal="1"/>
  </resource>

  <resource id="2" name="cpu3" frequency="20"
            type="CPU_TYPE_2" packet_size="16">
    <port terminal="2"/>
  </resource>

  <resource id="3" name="cpu4" frequency="20"
            type="CPU_TYPE_2" packet_size="16">
    <port terminal="3"/>
  </resource>

</resource_list>
```

As you can see, our example system has four identical CPUs running at 20 MHz frequency. According to the PE library file, they are capable of executing 3 integer operations, 1 floating point operation, or 1 memory operation per clock cycle. Also an optional parameter for packet size has been defined. Therefore, any data transfer longer than 16 bytes is divided into smaller packets.

3.1.2 NoC

The platform network is a 2D-mesh which requires one router for each PE. The routers are defined in this section and connected to the CPUs specified in the previous section.

Listing 8: *Noc*

```
<noc class="mesh_2d" type="sc_rtl_1" subtype="2x2">

  <router_list>
    <router width="32" id="0" name="mesh_r1" frequency="50" type="mesh_router">
      <port name="mesh_p1" id="0" type="ptk_if" address="0x00000000"/>
    </router>
    <router width="32" id="1" name="mesh_r2" frequency="50" type="mesh_router">
      <port name="mesh_p2" id="1" type="ptk_if" address="0x00000001"/>
    </router>
    <router width="32" id="2" name="mesh_r3" frequency="50" type="mesh_router">
      <port name="mesh_p3" id="2" type="ptk_if" address="0x00010000"/>
    </router>
    <router width="32" id="3" name="mesh_r4" frequency="50" type="mesh_router">
      <port name="mesh_p4" id="3" type="ptk_if" address="0x00010001"/>
    </router>
  </router_list>

  <terminal_list>
    <connection port="0" router="0" name="mesh_p" id="0"/>
    <connection port="1" router="1" name="mesh_p" id="1"/>
    <connection port="2" router="2" name="mesh_p" id="2"/>
    <connection port="3" router="3" name="mesh_p" id="3"/>
    <network_interface type="mesh_if"/>
  </terminal_list>

</noc>
```

The type of the network determines which network model TG will use for the simulation. Router data widths and addresses must be defined for the simulation to run properly. For our example mesh the first four hexes of the address define the row number and the last four hexes the column number. Terminal list connects the router ports by id to the PE ports identified by the terminal number. The frequency of the network is set to 50 MHz.

3.2 Mapping

Finally, the individual tasks can be assigned for the CPUs.

Listing 9: *Mapping*

```
<mapping>

  <resource name="cpu1" id="0">
    <group name="g1" id="0">

      <!--  Assing task 1 to cpu1  -->
      <task name="SendOnce" id="1"/>

    </group>
  </resource>

  <resource name="cpu2" id="1">
    <group name="g2" id="1">

      <!--  Assing task 2 to cpu2  -->
      <task name="SendPeriodically" id="2"/>

    </group>
  </resource>

  <resource name="cpu3" id="2">
    <group name="g3" id="2">

      <!--  Assing task 3 to cpu3  -->
      <task name="ForwardData" id="3"/>

    </group>
  </resource>

  <resource name="cpu4" id="3">
    <group name="g4" id="3">

      <!--  Assing task 4 to cpu4  -->
      <task name="DataProcessing" id="4"/>

    </group>
  </resource>
</mapping>
```

The example maps one task for each CPU based on their id number.

4 CONSTRAINTS

All the settings for the simulation are specified in the constraints section.

Listing 10: *Constraints*

```
<constraints>

  <rng_seed        value="42"/>

  <sim_resolution time="1.0"    unit="fs"/>
  <sim_length     time="100.0"    unit="ms"/>

  <measurements   time="2.0"    unit="ms"/>

  <pe_lib          file="examples/pe_lib.xml"/>

  <log_exec_mon     file="log_execmon.txt"/>
  <log_token        file="log_token.txt"/>
  <log_summary      file="log_summary.txt"/>
  <log_pe           file="log_pe.txt"/>
  <log_app          file="log_app.txt"/>

  <cost_function    func="ec_1"/>
  <cost_function    func="ec_2"/>
  <cost_function    func="tc_1"/>
  <cost_function    func="tc_2"/>
  <cost_function    func="tc_3"/>
  <cost_function    func="tc_4"/>
  <cost_function    func="tc_tot"/>
  <cost_function    func="pf_0"/>
  <cost_function    func="pu_0*100"/>
  <cost_function    func="pf_1"/>
  <cost_function    func="pu_1*100"/>
  <cost_function    func="pf_2"/>
  <cost_function    func="pu_2*100"/>
  <cost_function    func="pf_3"/>
  <cost_function    func="pu_3*100"/>
  <cost_function    func="pu_avg*100"/>
  <cost_function    func="lat_20_32_avg"/>

</constraints>
```

The seed for random number generators, the log files as well as the simulation length and the interval between snapshots can be adjusted. If the log files aren't defined, TG will skip the log writing process. The file containing the PE attributes is also defined here, although the PEs are already defined in the platform section.

During the simulation performance statistics is gathered by TG. Cost functions are used to view the desired values after the simulation has been completed. The functions of the example along with their explanations are listed on the following page.

Cost function	Description
ec_1	Total times event 1, ActivateSendOnce, has happened
ec_2	Total times event 2, ActivateSendPeriodically, has happened
tc_1	Total times task 1, SendOnce, has been triggered
tc_2	Total times task 2, SendPeriodically, has been triggered
tc_3	Total times task 3, ForwardData, has been triggered
tc_4	Total times task 4, DataProcessing, has been triggered
tc_tot	Total times all tasks have been triggered
pf_0	Frequency of PE with id 0, cpu1, MHz
pu_0*100	Average utilization percentage of PE with id 0, cpu1
pf_1	Frequency of PE with id 1, cpu2, MHz
pu_1*100	Average utilization percentage of PE with id 1, cpu2
pf_2	Frequency of PE with id 2, cpu3, MHz
pu_2*100	Average utilization percentage of PE with id 2, cpu3
pf_3	Frequency of PE with id 3, cpu4, MHz
pu_3*100	Average utilization pertcentage of PE with id 3, cpu4
pu_avg*100	Average utilization percentage of all PEs
lat_20_32_avg	Average latency for a token transfer from task 2 to task 3

5 SIMULATION

Now we're ready for the actual simulation. Start the simulation simply by invoking the `sctg` with the example XML file as a parameter.

```
$ ./sctg -i examples/tutorial.xml
```

The resulting print should match the following.

```
Transaction Generator 2
input-file: examples/tutorial.xml
Delay for playback: 1ms
Directory to save logs: .
cpu1: int (3) float (1) mem (1)
cpu2: int (3) float (1) mem (1)
cpu3: int (3) float (1) mem (1)
cpu4: int (3) float (1) mem (1)
Task SendOnce mapped to PE cpu1
Task SendPeriodically mapped to PE cpu2
Task ForwardData mapped to PE cpu3
Task DataProcessing mapped to PE cpu4
PE cpu1 mapped to event 1
PE cpu2 mapped to event 2
Found address 0x0 for PE cpu1 (terminal:0 router:0 port:0)
Found address 0x1 for PE cpu2 (terminal:1 router:1 port:1)
Found address 0x10000 for PE cpu3 (terminal:2 router:2 port:2)
Found address 0x10001 for PE cpu4 (terminal:3 router:3 port:3)
Starting simulation with seed 42
Measuring statistics with 2 ms interval
Measuring. Current simulation time is 2 ms
Measuring. Current simulation time is 4 ms
Measuring. Current simulation time is 6 ms
.
.
.
Measuring. Current simulation time is 96 ms
Measuring. Current simulation time is 98 ms
Measuring. Current simulation time is 100 ms
Simulation ends at 100 ms
Cost 1   ; 1                        ; "ec_1"
Cost 2   ; 25                       ; "ec_2"
Cost 3   ; 1                        ; "tc_1"
Cost 4   ; 25                       ; "tc_2"
Cost 5   ; 26                       ; "tc_3"

Cost 6   ; 26                       ; "tc_4"
Cost 7   ; 78                       ; "tc_tot"
Cost 8   ; 20                       ; "pf_0"
Cost 9   ; 25                       ; "pu_0*100"
Cost 10  ; 20                       ; "pf_1"

Cost 11  ; 3.75                     ; "pu_1*100"
Cost 12  ; 20                       ; "pf_2"
Cost 13  ; 7.058050000000001        ; "pu_2*100"
Cost 14  ; 20                       ; "pf_3"
Cost 15  ; 18.5937                  ; "pu_3*100"

Cost 16  ; 13.6004375               ; "pu_avg*100"
Cost 17  ; 4.6e-06                  ; "lat_20_32_avg"
```

At the start of the simulation TG prints information regarding the simulation, PEs, tasks, events and routing. After that, the measurements begin. Our greatest interest is, naturally, the cost function results.

One of the assets of TG is fast and easy modification. Next, we'll map three of the tasks to a single CPU and see how it affects the results. Make a copy of tutorial.xml in examples directory, rename it as tutorial_1.xml and open it using a text editor. Locate the mapping section and cut-paste the lines containing the task assignments for CPUs 2 and 3 below the task assignment of cpu1. Save the file.

Listing 11: *Modified task assignments*

```
<resource name="cpu1" id="0">
    <group name="g1" id="0">

      <!--  Assing task 1 to cpu1  -->
      <task name="SendOnce" id="1"/>
      <task name="SendPeriodically" id="2"/>
      <task name="ForwardData" id="3"/>

    </group>
</resource>
```

Now three of the tasks are executed by cpu1. Run the simulation again using tutorial_1.xml as an input file. The number of triggered events and tasks hasn't changed, but the effect can be seen in the utilization function results.

```
Cost 9   ; 25                      ; "pu_0*100"
Cost 11  ; 3.75                    ; "pu_1*100"
Cost 13  ; 7.058050000000001       ; "pu_2*100"
Cost 15  ; 18.5937                 ; "pu_3*100"
Cost 16  ; 13.6004375              ; "pu_avg*100"
```

Make a copy of tutorial_1.xml for another modification and rename it as tutorial_2.xml. This time, we'll adjust the hardware by increasing the frequency of one of the CPUs. Locate the platform section and set the frequency of cpu1 (id 0) to 45 MHz. Save the file and run the simulation. See, how the change affects the results.

Now, cost functions aren't the only source of TG's information. If enabled in the XML application file, TG will also write comprehensive log files. The most important logs are listed below.

File	Information
log_app.txt	Task states and data buffers
log_pe.txt	PE utilization and HW data buffers
log_summary.txt	Cost function results, PE, task and event statistics

If you open log_summary.txt, you'll notice that most of our cost function results could have also been found in this file without the use of cost functions. Therefore, we could have used cost functions to gather more complex information and read other results, for example PE utilizations, from the log file. An excerpt of the log is given below.

```
- PE cpu3 average utilization during simulation was 0.0705805
  idle:    1858839  exec:     141161  total:    2000000
  sent           2112 bytes
  received       1736 bytes
  intra tx          0 bytes (traffic between tasks)
```

Another easy tool for processing TG's information is graphical Execution Monitor which is introduced in the following section.

6 SIMULATION WITH EXECUTION MONITOR

Execution Monitor is a Java based program for visualization of processing element utilization and simulation. Although Transaction Generator 2 can be used as a stand-alone simulator, Execution Monitor is meant to complement the information provided by TG.

6.1 Setup

Simulation with Execution Monitor requires a compiled installation of Boost's libraries as well as the following tools.

Java Development Kit (JDK)	Tested with version 1.6.0_16
Apache Ant	Tested with version 1.7.1

Modify Makefile.env again and uncomment the following lines. Values of `ASIO_FLAGS` and `ASIO_LINK` are given to the compiler and depend on the used system. Refer to Boost.Asio documentation for help. The following example values are for Windows XP with Cygwin.

```
USE_EXECMON    = -DSCTG_USE_EXECMON
ASIO_FLAGS     = -D_WIN32_WINNT=0x0501 -D__USE_W32_SOCKETS
ASIO_LINK      = -lboost_system -lws2_32
```

Set environment variable `JAVA_HOME` to point JDK and install Execution Monitor by invoking `make`. This will also recompile Transaction Generator 2 automatically.

6.2 Simulation

Before simulation, launch Execution Monitor.

```
$ bin/execution_monitor &
```

Load the example user interface (*File->Open*) using file `tutorial_conf.xml` in the examples directory. The window now matches figure 3.

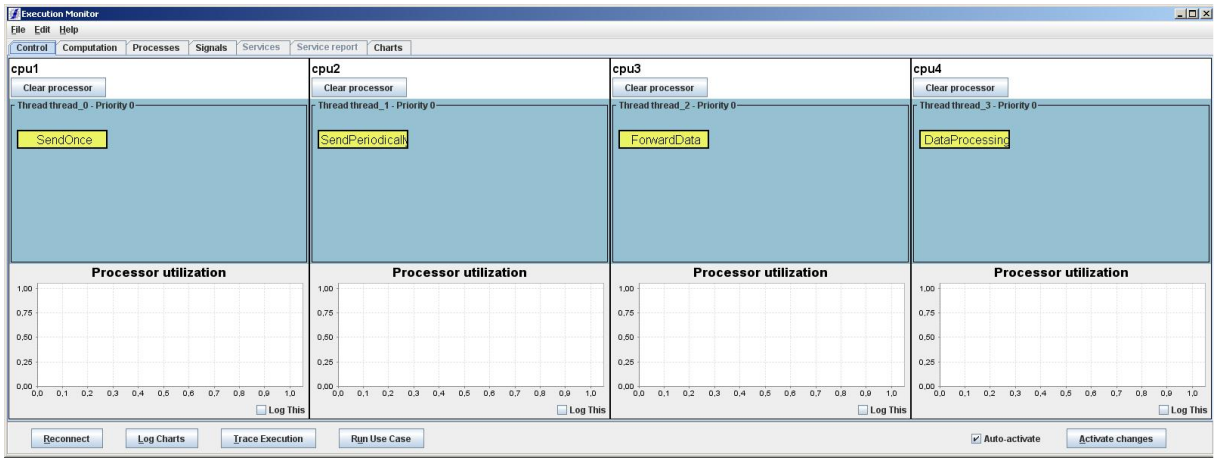

Figure 3: *Execution Monitor Control tab*

Now, run the simulation again. To enable the use of Execution Monitor, simply add parameter -e.

```
$ ./sctg -e -i examples/tutorial.xml &
```

TG will wait for connection from Execution Monitor. Press the reconnect button in the lower left corner of Execution Monitor window to start the simulation. You can now observe the utilization of the CPUs in the Control tab.

At 50 ms task 1 is activated and cpu1 begins the computation of 1500000 integer operations which takes 25 ms to complete. The CPU then stays idle for the rest of the simulation.

Data arriving from cpu1 causes a minor deviation in cpu3 at 75 ms, but otherwise cpu2 and cpu3 operate at regular intervals according to event 2. The utilization of cpu4 differs due to the two execution variations and the randomized number of operations in DataProcessing.

Next, open the Computation tab (figure 4). It shows the execution statistics. If all the CPUs run at the same frequency as the do in the example, it's quite easy to balance the computational load by assigning equal amount of execution cycles to the CPUs. Of course, the dependencies between the tasks must be considered as well. However, system balance is peripheral in this example.

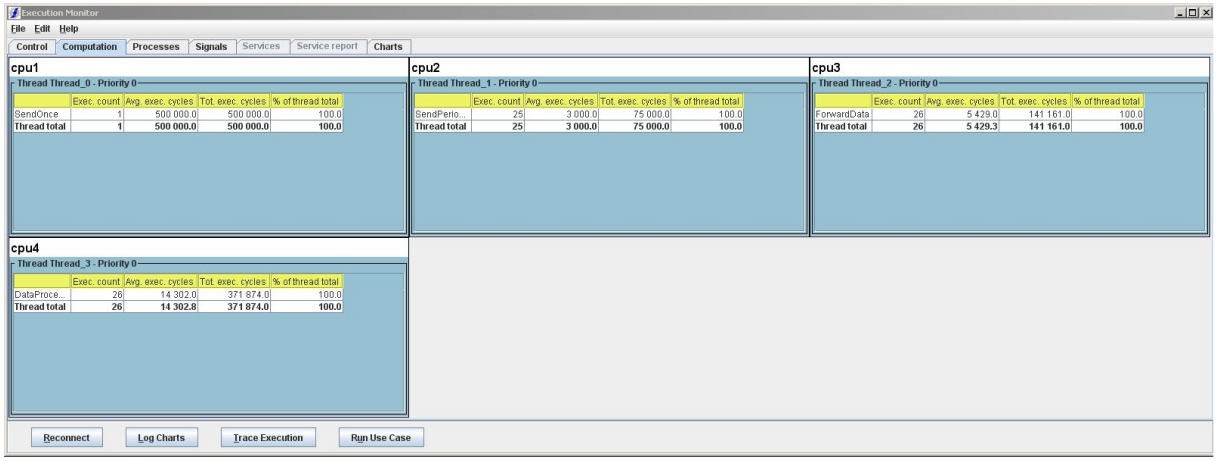

Figure 4: *Computation tab*

Have a look at the other tabs. Processes shows simulation statistics by task and Signals the amount of data transferred between the tasks. The Charts tab isn't used in the example.

Next, simulate tutorial_1.xml and tutorial_2.xml with Execution Monitor and observe the different tabs during the run. An example of Control tab is given in figure 5.

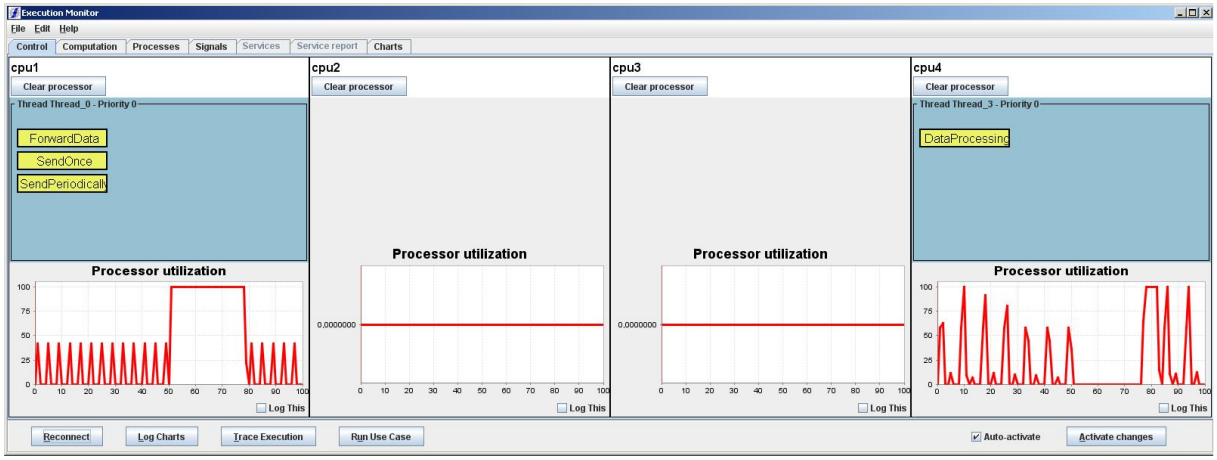

Figure 5: *Example CPU utilizations*

You'll notice, how SendOnce uses all the computation capacity of cpu1 after 50 ms, blocking the other two tasks on the CPU. This also forces cpu4 to stay idle for a long period since no token arrives from ForwardData. Once the computation is finished the other tasks are executed for several times and data tokens are sent to cpu4 repeatedly until all the blocked executions are processed. Figure 6 depicts the scenario.

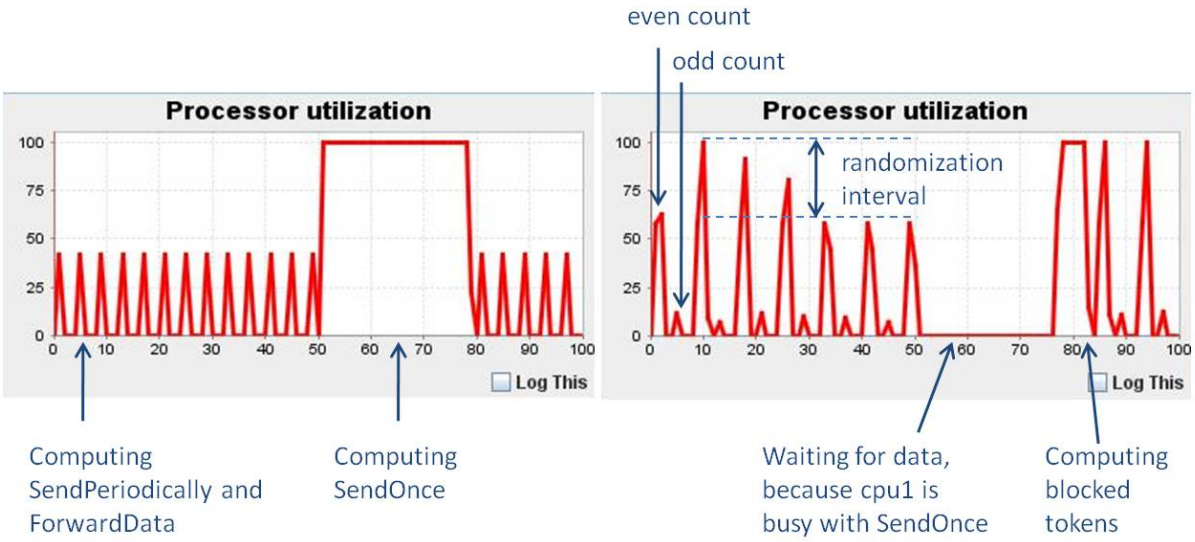

Figure 6: *CPU1 and CPU4 utilization highlights*

Althought simulation with TG reveals that all the necessary tasks are executed during the run, it cannot detect bad mapping of the tasks. With Execution Monitor we have found a bottleneck in the example: Assigning three tasks to cpu1 throttles the performance of the whole system.

7 CONCLUSIONS

The tutorial is now finished. It introduced the XML system modeling, running simulations and the use of Execution Monitor with Transaction Generator 2. You should now know how the application and the platform NoC is described in the XML file, how to reassign the tasks among the PEs, how to modify the hardware of existing systems and how to simulate high abstraction level models using Transaction Generator 2.

To learn more, see also the example real-life application provided with the TG package. It features a video coded that encodes and decodes a QCIF sized video stream using a 2x2 mesh NoC. The application XML file (test_mesh.xml) is located in the examples directory. You can also keep experimenting with the example application given in this tutorial.

References

[1] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of Network-on-chip", ACM Computing Surveys, Volume 38 , Issue 1, 2006, Article No. 1.

[2] Erno Salminen, Cristian Grecu, Timo D. Hämäläinen, Andrè Ivanov, "Application modeling and hardware description for Network-on-chip benchmarking", IET Computers & Digital Techniques, September 1, 2009, Vol.3, Issue 5, Special issue on Network-on-chip, pp. 539-550.

[3] Kalle Holma, Tero Arpinen, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, "Real-Time Execution Monitoring on Multi-Processor System-on-Chip", International Symposium on System-on-Chip, Tampere, Finland, November 5-6, 2008, pp. 23-28.

[4] Lasse Lehtonen, Esko Pekkarinen, "Transaction Generator 2 Technical", spesification document, July 15, 2010, 70 pages