SycView: Visualize and Profile SystemC Simulations

Denis Becker, Matthieu Moy, Jérôme Cornet

To cite this version:

Denis Becker, Matthieu Moy, Jérôme Cornet. SycView: Visualize and Profile SystemC Simulations. 3rd Workshop on Design Automation for Understanding Hardware Designs, DUHDe 2016, Mar 2016, Dresden, Germany. hal-01295282

HAL Id: hal-01295282

https://hal.science/hal-01295282

Submitted on 30 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SycView: Visualize and Profile SystemC Simulations

Denis Becker*†
Denis.Becker@st.com

Matthieu Moy†
Matthieu.Moy@imag.fr

Jérôme Cornet*
Jerome.Cornet@st.com

*STMicroelectronics, F-38019 Grenoble, France
†Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France

***Abstract*—The design of systems-on-chip requires simulation of highly abstracted models, such as SystemC Transaction Level Models (TLM), in addition to traditional register transfer level models. Due to the growing complexity of products, analyzing and understanding the behavior of the corresponding SystemC platforms becomes itself a challenge. Huge code bases are generally written by multiple authors and it is rare that a single person has a comprehensive detailed knowledge of the model. However, models have to be developed, validated and used, so understanding them is important. Moreover, the increasing complexity leads to slower simulations, so there is a need to speed them up.**

We propose a tool, SycView, which provides a view of the profile of a SystemC simulation. This tool helps answering these two major needs: understanding complex hardware simulations, and highlighting bottlenecks of SystemC simulations.

I. Introduction

SystemC is a hardware modeling C++ library standardized by IEEE. It enables a designer to simulate concurrent processes, and thanks to the use of C++, provides an easy way to simulate mixed software/hardware systems. For this reason, it is mostly used for systems-on-chip (SoC) modeling. It has been designed so that it can be used at different levels of abstraction, thus at different steps of the design workflow.

The increasing complexity of models and components has led to the need to speed up such simulations. The natural method to optimize a program, SystemC or not, is first to analyze it in order to find the bottleneck and then to apply an optimization on the corresponding part of the program. Since SystemC is a C++ library, usual profilers for C++ like `gprof` or `valgrind+kcachegrind` can be used. They will however miss important aspects of the execution of a SystemC program, like: how much time is spent in the SystemC kernel as opposed to the user-written parts, per-process statistics, simulated time based visualization. To the best of our knowledge, there is no turnkey application available to get these information from a SystemC simulation. Also, as the complexity of models grows, it becomes necessary to visualize executions at a high abstraction level to get a better understanding of the system synchronization.

In this paper, we present SycView, a tool for visualization and profiling of SystemC simulations. Our purpose is to provide SystemC users with a tool to get *SystemC-aware* information about the execution of their simulations. The underlying motivation of the tool is to increase the understandability of complex SystemC models and to help the set up of optimizations and/or parallelization.

II. Background

In this paper, the term *wall-clock time* refers to the time spent by the execution of the simulation on the host machine, and the term *simulated time* to the virtual time spent by the model during the execution (i.e. what the real hardware will be supposed to spend). We call a *transition* any portion of code in `SC_METHOD`s or `SC_THREAD`s that is executing atomically from the point of view of SystemC. As an example, the following code will produce 3 transitions:

```
void compute() { // declared as SC_THREAD
    do_stuff();             // transition #1
    wait(15, SC_NS);
    do_more_stuff();        // transition #2
    wait(event);
    do_even_more_stuff();   // transition #3
}
```

III. Proposed tool

The principle of SycView is simple: it consists in trace recording during simulation (III-A) and then trace visualization (III-B). In order to print traces, we instrumented the SystemC kernel, based on the ASI (formerly OSCI) implementation. For the visualization we developped a graphical user interface (GUI) written in Java.

A. Trace Recording

The trace recording consists in printing data at interesting points. For instance, we print a trace each time a transition yields to the kernel, containing:

- The SystemC process which triggered this transition.
- The type and arguments of the `wait` performed.
- The wall-clock time duration of the transition.

When the simulation ends, we store those information on text files. Then for example, we are able to find which SystemC process (or group of processes) globally consumes most of the time. We can also use statistical metrics to get the average execution time of a process and compare it with the number of executions to highlight how each process spends its time.

We also keep track of the number of runnable processes at the beginning of each delta cycle. More precisely, what we get is the maximal number of runnable processes at the beginning of each immediate notification cycle within each delta cycle.

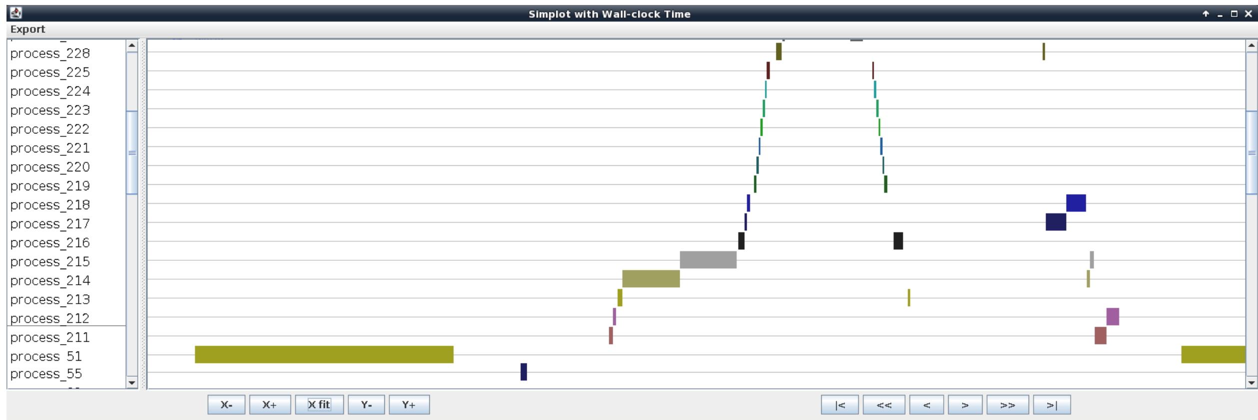

Figure 1: Wall-clock time axis

B. Visualization

We have implemented a GUI providing six different views, which can be split in two main categories.

The first category is plotting the scenario of an execution (order and duration of transitions) as a function of either wall-clock time or simulated time. For example, Figure 1 shows the screenshot of a wall-clock time plot. On this plot we can see that there seems to be a chain triggering of the processes executing in one order on the left part, and in the reverse order on the right part. The width of a rectangle is proportional to the wall-clock time duration of the corresponding SystemC transition. This view, as illustrated by this little example, can be used to identify patterns in the executions of transitions, and can highlight which transitions are particularly time-consuming. Such information could hardly be identified by a static code analysis, because of its complexity. On the other hand on Figure 2 we can see an example of a simulated time plot. The processes are also represented on the left part, and on the right part, a vertical stroke is placed for each process execution, in the simulated time it occured. Note that what appears to be black rectangles are strokes close together. The blue rectangles represents time ranges in which the transition may have occured, based on loosely-timed information on the model. Note that time ranges are not part of the SystemC kernel so we added that when we instrumented the kernel because it was part of our industrial context.

The second category consists in giving statistical metrics about the platform execution. We display the wall-clock time consumption per process (with statistical metrics), as on Figure 4. We can see that the first process uses almost 13 % of the time in 10111 transitions, while the third process uses almost 8 % of the time in only 14 transitions. That makes a huge difference in the mean execution time, and we can deduce that optimizing the third process may be more efficient than the first one. We also show the number of runnable SystemC processes at each delta cycle, shown in Figure 3. This table can also be displayed for `SC_THREAD`s or `SC_METHOD`s only.

IV. Related Work

We have found that most of the litterature about SystemC visualization focus on the model description. For example, the

Runnable Processes by Cycle

Export

# runnable	Number of cycles	Part
0	1	0 %
1	34872	16.168 %
2	5473	2.537 %
3	165472	76.719 %
4	841	0.39 %
5	42	0.019 %
6	10	0.005 %
7	2	0.001 %
17	5991	2.778 %
21	26	0.012 %
25	2957	1.371 %

Figure 3: Number of runnable processes

tools presented in [1], [2], [3], [4], [5] and more recently [6] gives access to the visualization of the hierarchical architecture of a platform, as well as the exploration of the components behaviour. Such information are precious and these tools are complementary with our tool especially when it comes to understanding a design. But what we found missing in these tools is a software point of view, in order to understand the model not as the representation of an SoC, but as a piece of software. Besides, some of the above mentioned tools need to instrument the model, which is generally not possible in the case of huge and complex models.

V. Example Usage

SycView is not only a software engineering tool, the information given are also useful to characterize a model theoretically. Initially, this tool was made to help the set up of a parallel SystemC kernel on an existing industrial model. We have used SycView outputs in [7] to illustrate the fact that the existing parallelization approaches are not applicable in the case of a loosely-timed TLM model, based on the analysis of the profile of our industrial test case.

VI. Conclusion

In this paper, we have presented SycView, for visualization and profiling of SystemC simulations. The main advantage of

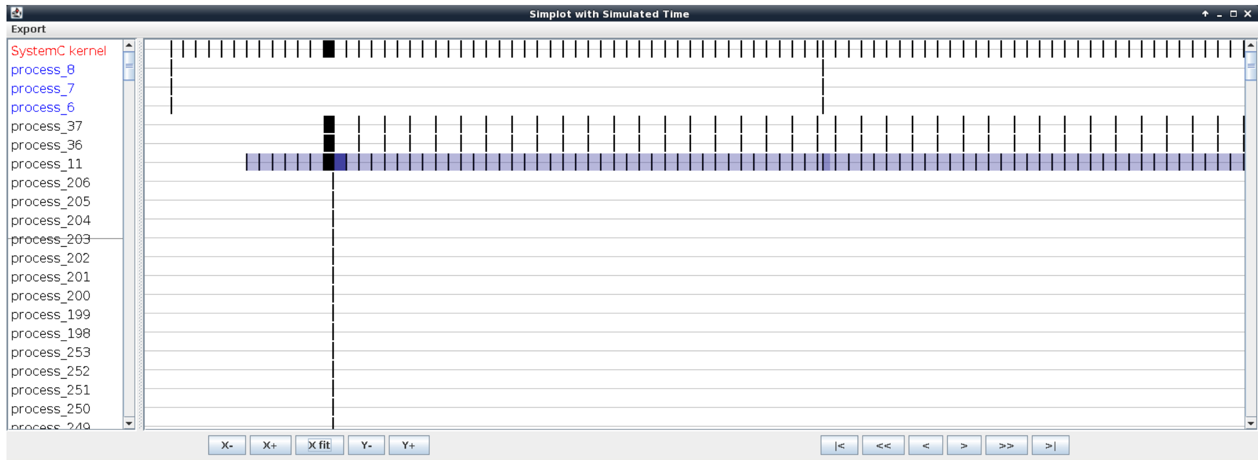

Figure 2: Simulated time axis

this tool compared to classic C++ profiling tools like `gprof` or `valgrind` is that it understands and exploits SystemC constructs. Currently the data collected are limited to SystemC processes, simulation cycles, simulated and wall-clock times. However the tool can be extended to add data collection about transactions or event processing for example, that could also be interesting for analysis purposes.

From the data gathered during the execution of a simulation, the tool can display the following views:

- Simulation timing diagram indexed by either wall-clock or simulated time.
- Wall-clock time consumption per SystemC process.
- Partition of delta cycles depending on the number of runnable processes.

The views of the first item are helpful to understand the sequence of transitions for a simulation. The two other ones presented in this list can quickly identify which parallelization approach is likely to work efficiently, and which one cannot be efficient.

References

[1] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient Automatic Visualization of SystemC Designs," in *FDL*, 2003, pp. 646–658.

[2] D. Berner, J.-P. Talpin, H. D. Patel, D. Mathaikutty, and S. K. Shukla, "SystemCXML: An Exstensible SystemC Front end Using XML," in *FDL*, 2005, pp. 405–409.

[3] A. Donlin and T. Lenart, "Performance Analysis and Visualization of SystemC Models," 2006.

[4] C. Albrecht, C. J. Eibl, and R. Hagenau, "A Loosely-Coupled Graphical User Interface for Run-Time Control of SystemC Simulation Models," *IJSSST*, 2006.

[5] C. Genz, R. Drechsler, G. Angst, and L. Linhard, "Visualization of SystemC Designs," in *Circuits and Systems (ISCAS), IEEE International Symposium on*, 2007, pp. 413–416.

[6] J. U. Stoppe, M. Michael, M. Soeken, R. Wille, and R. Drechsler, "Towards a Multi-dimensional and Dynamic Visualization for ESL Designs," in *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*, 2014.

[7] D. Becker, M. Moy, and J. Cornet, "Challenges for the Parallelization of Loosely-Timed SystemC Programs," in *IEEE International Symposium on Rapid System Prototyping (RSP)*, 2015.

Consumptions

Export

Name	Part	Cumulated time (µs)	Min	1stQ	Med	3rdQ	Max	# of exec	Mean
process_1	12.858 %	10801025	14	1107	1123	1222	18372	10111	1068.245
process_101	[illegible]	7219857	73652	75911	77535	78731.5	84082	93	[illegible]
process_100	7.969 %	6693732	395202	484460	486617	494005	495737	14	478123.72
process_5	5.91 %	4964331	1	4	5	9	983	635129	7.816256
process_22	2.882 %	2420655	43	46	409.5	764	876	5966	405.7417
process_21	2.79 %	2343369	11	88	95	111	81827	21430	109.34993
process_15	2.768 %	2324868	12	13	132	243	377	17896	129.90993
process_18	2.713 %	2279319	14	15	135	239	304	17896	127.364716
process_141	2.696 %	2265016	11	12	129.5	238	386	17896	126.56549
process_41	1.864 %	1565956	846	2730	222992.5	225330	226691	10	156595.6
process_42	1.66 %	1394581	11	74	87	109	4470	14444	96.55089
process_43	1.158 %	972725	14	22	25	26	66518	8922	109.025444
process_44	1.126 %	946173	11	64	82	104	37264	9844	96.11672
process_45	1.042 %	875359	9	10	55.5	86	106	17896	48.91367
process_47	1.03 %	865512	11	12	74	83	102	17896	48.363434
process_48	1.027 %	862644	11	11	74.5	85	106	17896	48.203175
process_58	0.975 %	819206	3	5	5	5	281	164286	4.9864626
process_49	0.934 %	784960	13	14	31	32	230	33308	23.566711
process_50	0.891 %	748067	3	4	5	5	270	164286	4.5534434
process_79	0.885 %	743772	3	4	4	5	45	164286	4.5273
process_78	0.758 %	636525	26	28	40	42	75	17896	35.568005
process_76	0.725 %	609143	14	15	29	53	76	17896	34.03794
process_74	0.716 %	601154	17	19	36	47	96	17897	33.589653
process_72	0.715 %	600779	10	11	41.5	55	114	17896	33.570576
process_71	0.711 %	597516	12	13	26	53	834	17897	33.38638

Figure 4: Wall-clock time usage of SystemC processes