jTLM: an Experimentation Framework for the Simulation of Transaction-Level Models of Systems-on-Chip

Giovanni Funchal[*,†]

[*] *STMicroelectronics*
12, rue Jules Horowitz
38019 Grenoble, France
first.last@st.com

Matthieu Moy[†]

[†] *Verimag*
2, avenue de Vignate
38610 Gières, France
first.last@imag.fr

Abstract

Virtual prototypes *are simulators used in the consumer electronics industry.* Transaction-level Modeling (TLM) *is a widely used technique for designing such virtual prototypes. In particular, they allow for early development of embedded software.*

The SystemC *modeling language is the current industry standard for developing virtual prototypes. Our experience suggests that writing TLM models exclusively in SystemC leads sometimes to confusion between modeling concepts and their implementation, and may be the root of some known bad practices.*

This paper introduces jTLM, *an experimentation framework that allow us to study the extent to which common modeling issues come from a more fundamental constraint of the TLM approach. We focus on a discussion of the two modes of simulation scheduling:* cooperative *and* preemptive. *We confront the implications of these two modes on the way of designing TLM models, the software bugs exposed by the simulators and the performance.*

1. Introduction

Today's industry constantly faces the challenge of staying competitive in spite of the complexity and the time-to-market pressure of designing high-tech consumer electronic devices. Most of the functionality of these devices is often grouped into a single integrated circuit, which is then called a *system-on-chip* (SoC).

Register-transfer level. RTL models are the traditional entry point in the SoC design flow. They specify precisely the hardware logic needed for manufacturing the physical chip.

However, the design of SoCs is not limited to the development of custom hardware: Software (drivers, etc.) is also an important part of the system and must be developed conjointly. The simulation of RTL models is too slow for software development, because they have too much detail (cycle-accuracy, micro-architecture, etc.) [1]. On the other hand, high-level simulators such as that included in the iPhone SDK [2] are too coarse for low-level software development.

An alternative consists of using a *virtual prototype*: a model of the hardware specifically intended for simulation and development of software before the real, physical hardware is available.

Transaction-level modeling. TLM is a widely used technique for designing virtual prototypes [3]. This approach tries to provide the "right" abstraction level in the sense of keeping just enough details so as to maintain the behavior of the hardware as perceived from a software programmer's point-of-view in what concerns the functionality. Thus, they effectively address the aforementioned complexity and time-to-market issues.

2. SystemC: the industry standard

SystemC [4] is the current industry standard language for developing transaction-level models. Strictly speaking, SystemC is a C++ library that includes a simulation kernel and data-types specially designed for describing hardware structures such as wires and registers.

Time in SystemC. SystemC includes a notion of *simulated time* that represents or approximates the time by which actions happen on the concrete system. Simulated time is completely disconnected from the *wall-clock time*, i.e. the time taken by the simulation when running on an ordinary computer.

Concurrency in SystemC. For describing concurrency, SystemC includes a notion of *process* and a cooperative, discrete-event simulation *scheduler*. During simulation, each process is either running, ready or waiting. At each step, the scheduler chooses

978-3-9810801-7-9/DATE11/©2011 EDAA

one process among those that are ready and puts it to run. This process then either runs to completion or calls at some point a primitive that yields control back to the scheduler. In other words, the scheduler is not able to *preempt* the running process. If the process enters an infinite loop at some point, the rest of the processes will starve. Therefore, the overall progress of the simulation depends on global cooperation.

3. Overview and contributions

It is very hard to actually distinguish the modeling concepts of the TLM approach from their implementation in the SystemC language. A contributing factor to this issue is that SystemC was originally designed for RTL modeling and has evolved to support TLM. In the process, many primitives designed for RTL modeling are now used for TLM modeling. Our experience shows that this raises many questions about the adequacy of such and such primitive to accomplish common modeling needs.

Previous works have identified some *bad modeling practices* in TLM and partially linked them to the usage of SystemC primitives. These include using non-persistent events when the intent was to model persistent events, having hard-coded fixed delays when the intent was to model inaccurate timing [5] or yielding at the wrong places [6].

To measure the extent to which the modeling issues listed above come from SystemC or from a more fundamental constraint of the TLM approach, we have developed and performed some experiments in a custom simulator built from scratch for this purpose. This simulator should diverge from SystemC as much as possible, in an attempt to identify new primitives that best suit the programmers' intents. We call the resulting framework *jTLM*.

Two main differences between jTLM [7] and SystemC are the handling concurrency and the modeling of time. This paper presents jTLM with an emphasis on the first: while SystemC is cooperative, jTLM proposes both a preemptive and a cooperative execution modes. We show that this feature allows writing more robust models, avoids having to manually specify preemption points, and allows to better exploit the parallelism of the host machine.

Related works. The closest related work is actually SpecC, a dedicated language for high-level modeling of SoCs. From the SpecC LRM [8] (2.4.2.i), the scheduler is theoretically allowed to be either preemptive or cooperative, but the reference implementation is cooperative. To our best knowledge, they do not further discuss the effects of such kind of choices from a modeling point of view, which is the main focus of this paper.

Structure of the paper. Section 4 presents a brief summary of the contributions of jTLM and compares them to SystemC. Section 5 discusses the implications of each of the choices we made in jTLM.

4. Summary of the contributions

- While the cooperative semantics of SystemC are deeply rooted in the language, jTLM allows both preemptive and cooperative execution. The user can either choose one of the two modes, or alternatively design the model so that it works in both, which we strongly recommend.
- In cooperative simulators, preemption points must be manually specified and, because context switching is expensive, there is a tendency towards having as few as possible. The preemptive mode of jTLM provides an alternative, and we discuss its implications in Section 5.2.
- The preemptive mode of jTLM places no artificial constraints on the parallelism of the model being simulated. As a result, we are naturally able of exploiting multiple physical processors if available on the machine that hosts the simulation. In comparison, previous attempts at automatic parallelization of SystemC either introduce additional cooperation points on communications [9], making them non-conforming w.r.t. the standard (which we think is unreasonable), or require heavyweight analysis techniques [10].
- On the down side, the preemptive mode is non-reproducible because it relies on the host OS scheduler that we do not control. This is a significant drawback with respect to the cooperative mode.
- Transaction-level models omit many micro-architectural details for performance and lesser modeling effort. This is transparent most of the time, but some particularly nasty low-level software bugs depend on these details. Cooperative models are not able to detect such bugs without heavy instrumentation (and the slowdown that goes with it). The preemptive mode of jTLM can expose some of them and we give the details in Section 5.3.

5. Discussion

5.1. jTLM from the user point of view

Simulated time in jTLM is, like in SystemC, completely disconnected from wall-clock time. The scheduler increments the simulated time in discrete steps by keeping an agenda of deadlines.

Similarly to SystemC, jTLM includes the following primitives: **awaitTime** pauses the caller for the amount of simulated time specified as a parameter; and **awaitEvent** pauses until another process calls **signalEvent** on the same event.

jTLM's **signalEvent** has slightly different semantics than SystemC's nearest equivalent: **notify**. In jTLM, when a process wakes up from waiting an event, it may immediately start execution. In SystemC, the woken process will only be able to execute after the current process reaches a preemption point (call to **wait** or end of execution), creating an implicit atomicity that some users may not even be aware of. While the semantics of **notify** are useful for encoding synchronous circuits and combinatory feedback in RTL, it is arguable if this makes any sense for TLM modeling. This became more of a concern in the design of the preemptive mode of jTLM where preemption points are not explicit and an immediate awakening seems more intuitive.

Concurrency in jTLM. In the preemptive mode of jTLM, we do not distinguish between preemptive multitasking and multiprocessing. Processes run each on a different Java thread, managed by the host computer scheduler. Therefore, processes can be run on multiple physical processors if the machine that hosts the simulation supports it.

However, jTLM allows parallel execution only when actions are simultaneous in simulated time. To illustrate this, consider the example in Figure 1(a) showing the traces of a cooperative simulation with three processes: B and C execute at simulated time 5 (gray rectangle), but the simulation is cooperative: C runs first in wall-clock time (black rectangle), then B takes over. The task A waits until time 10 before executing.

Figure 1(b) shows the preemptive simulation of the same processes in a machine with several processors, allowing B and C to be effectively executed in parallel. However, this is not the case for A which happens at a different simulated time.

5.2. Granularity and its implications

We define *granularity* as the amount of code between two preemption points. In a cooperative model, a too fine granularity (i.e. placing preemption points after each transaction) will seriously slow down the simulation because of the context switches and may even break the model.

On the other hand, a too coarse granularity may cause unexpected behavior, e.g. missing interrupts.

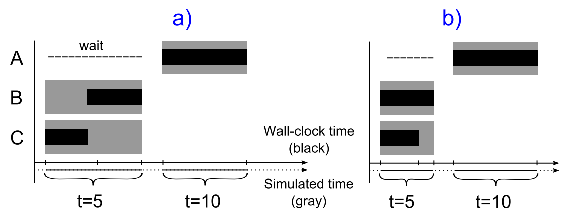

Figure 1. Time and concurrency in jTLM

It may also break the model if processes starve for shared resources. Furthermore, no safe automatic method is known to us for determining the optimal preemption points [11] in a cooperative simulation. Users are therefore currently obliged to manually place preemption points in an attempt to keep the model at an appropriate granularity.

In contrast, a preemptive simulation scheduler lifts the burden of manually specifying preemption points. In this case, the user is required to replace any implicit granularity supposition by an explicit lock. For instance, RMWs cannot anymore be modeled as a read followed by a write and must use some other mechanism that guarantees atomicity w.r.t. other transfers on the same bus. On the bad side, hardware engineers may not be used to the subtleties of writing parallel code, and indeed one may argue that preemption has no meaning in the model of a hardware block.

Nevertheless, we expect that the number of sections of code that must be protected in a preemptive simulation will be small compared to the number of preemption points that should be manually added to obtain a relatively faithful cooperative simulation.

5.3. Observable behavior: exposing bugs

There are several techniques to integrate software within a virtual prototype: *Instruction Set Simulators* (ISS) read instructions one-by-one from the binary code (compiled to the target processor) and simulate their execution. Variants may use dynamic translation [12] techniques. *Native wrappers* may either wrap the source directly into the virtual prototype, link with a binary compiled into native code [3], or use virtual machines [13].

They all have in common the fact that they redirect reads and writes from the software onto a bus, omitting many micro-architectural details (i.e. caches, fifos and pipelines) for performance and lesser modeling effort. As a result, virtual prototypes may be unable to expose some particularly nasty low-level software bugs, such as *race conditions*, that depend on these details. A race condition is when two non-atomic accesses to the same memory location happen concurrently, and at least one of them is a write. In this case, the contents of the memory location may become corrupted with an unexpected value (this bug could be fixed by using a lock to protect the memory location).

Pipelines, for instance, can reorder accesses producing race conditions which would not be observable by a typical transaction-level model, unless the model is changed to take the pipeline into account. Such heavy changes would demand a lot of effort and slow the simulation considerably.

In the preemptive mode of jTLM, processes are naturally exposed to some reorderings because they rely on the Java threads, whose exact semantics is given in the Java Memory Model [14]. If reads and writes to the memory are modeled as simple array manipulations, simultaneous accesses will effectively expose race conditions. Hence, they can then be detected by techniques such as stress testing or model checking.

However, it may be the case that the concrete system allows different reorderings than those of jTLM. If it allows *more* reorderings, the preemptive mode of jTLM may still miss some bugs; If, however, the concrete system allows *less* reorderings than jTLM, the user will need to add synchronization to protect against undesired behavior.

Example. Consider a system executing the following two loops in parallel (initially, **x=y=0**):

```
for(i=0; i<N; ++i) {        for(i=0; i<N; ++i) {
  x++;                        int l_y = y;
  y++;                        int l_x = x;
}                             if(l_x < l_y)
                                error();
                            }
```

This system behaves correctly under interleaving semantics (i.e. adding any number of preemption points between actions), but is still incorrect, since writes to **x** and **y** outside a **synchronized** statement can be re-ordered by the Java compiler or the underlying hardware of the host machine. Here again, the bug can be exhibited in the preemptive mode of jTLM, but not by a cooperative simulation.

6. Conclusion

We have presented jTLM in this paper as a custom simulator that provides an interesting new way to manage the description and the simulation of concurrency by identifying new primitives that best suit the programmer's intents.

jTLM innovates, providing both a cooperative and a preemptive mode. While the cooperative mode is good for reproducibility and ease of modeling of hardware blocks, it requires the manual identification of preemption points and places restrictions that severely compromise attempts at parallelization. On the other hand, the preemptive mode inherently supports parallel simulation but places an important burden on engineers that write hardware models. This is mostly because they need to understand and use synchronization mechanisms. In addition, while writing cooperative models is easier, composing them is harder because of implicit suppositions usually placed on the scheduling.

From the research point of view, a great advantage of jTLM is its simplicity. The complete scheduler implementation including the two modes is just ∼500 lines of code, making jTLM an ideal experimentation framework. Also, as it is a very thin layer on top of the JVM, standard tools like thread-aware debuggers and the Java PathFinder work with jTLM just like with plain Java.

We have made a number of important conclusions during our experiments. and jTLM has provided us some insight about problems that cannot be easily exposed or understood in a cooperative scheduler. In particular, we expect that the modeling of atomic primitives and the placement of yielding calls in jTLM will help our future work understanding how these primitives could better be used, bringing benefits to SystemC in the long run.

References

[1] M. Moy, "Techniques and tools for the verification of systems-on-a-chip at the transaction level," Ph.D. dissertation, INP Grenoble, December 2005.

[2] *iPhone SDK*, Apple, April 2010. [Online]. Available: http://developer.apple.com/iphone/

[3] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems.* Springer, 2006.

[4] *IEEE 1666-2005 Standard SystemC Language Reference Manual*, IEEE Standards Association, 2006.

[5] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz, "Test coverage for loose timing annotations," in *FMICS/PDMC*, 2006, pp. 100–115.

[6] J. Cornet, F. Maraninchi, and L. Maillet-Contoz, "A method for the efficient development of timed and untimed transaction-level models of systems-on-chip," in *DATE*, March 2008, pp. 9–14.

[7] G. Funchal and M. Moy, "jTLM: an experimentation framework for the simulation of transaction-level models of systems-on-chip," Verimag, Tech. Rep. TR-2010-17, 2010.

[8] R. Dömer, A. Gerstlauer, and D. Gajski, *SpecC Language Reference Manual 2.0*, 2002.

[9] B. Chopard, P. Combes, and J. Zory, "A conservative approach to SystemC parallelization," *ICCS 2006*, pp. 653–660, 2006.

[10] Y. Bouzouzou, "Accélération des simulations de systèmes-sur-puce au niveau transactionnel," Diplôme de Recherche Technologique, Université Joseph Fourier, 2007.

[11] J. Cornet, "Separation of functional and non-functional aspects in transactional level models of systems-on-chip," Ph.D. dissertation, INP Grenoble, April 2008.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of the Usenix ATC*, 2005, pp. 41–41.

[13] J. Dike, *User mode linux.* Prentice Hall, 2006.

[14] S. Microsystems, *JSR 133: Java Memory Model and Thread Specification*, 2004.