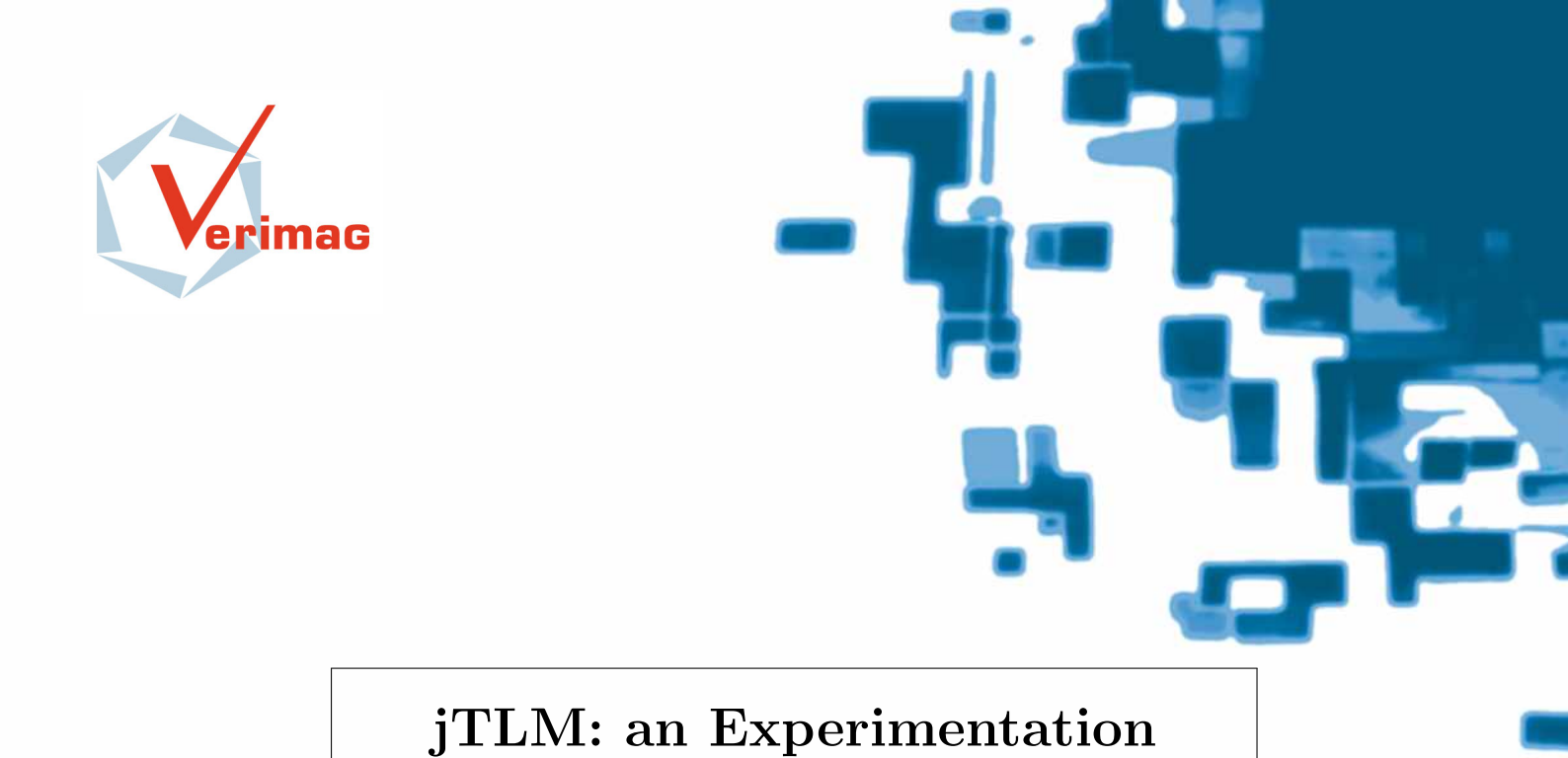

jTLM: an Experimentation Framework for the Simulation of Transaction-Level Models of Systems-on-Chip

Giovanni Funchal, Matthieu Moy

Verimag Research Report n^o TR-2010-17

September 9, 2010

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

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

Giovanni Funchal, Matthieu Moy

September 9, 2010

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.

Keywords: Transaction-level Modeling; SystemC; jTLM; Systems-on-Chip

Reviewers: Matthieu Moy

Notes: Short version of this report will appear at Design, Automation and Test in Europe (DATE'2011)

How to cite this report:

```
@techreport {TR-2010-17,
   title = { jTLM: an Experimentation Framework for the Simulation of Transaction-
Level Models of Systems-on-Chip },
   author = {Giovanni Funchal, Matthieu Moy},
   institution = {{Verimag} Research Report},
   number = {TR-2010-17},
   year = {2010}
}
```

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.) [14]. On the other hand, high-level simulators such as that included in the iPhone SDK [1] 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 [10]. 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.

Conceptually, a virtual prototype in TLM is a set of *components*, which represent hardware blocks (typically: CPUs, DMAs, memories, timers) connected through *interconnections* (bus). An interconnection transports *transactions*, which are abstractions of data exchanges.

2 SystemC: the industry standard

SystemC [12] 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. SystemC is a discrete-event simulator, which means that the state of the model changes at only a discrete set of points in simulated time. In other words, actions do not "take" time, they happen instantaneously w.r.t. the simulated time.

Parallelism in SystemC For describing parallelism, SystemC includes a notion of *process* and a cooperative simulation *scheduler*. SystemC processes can be of two kinds, **SC_THREAD**s and **SC_METHOD**s, which differ only on the type of stack management. During simulation, each process is either running, ready or waiting. At each step, the scheduler chooses 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 and thus 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 [11] or yielding at the wrong places [7].

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 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 [9] (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 features of jTLM and compares them to SystemC. Section 5 discusses the implications of each of the choices we made in jTLM. Section 6 illustrates the algorithm behind the scheduler and each of its primitives. Section 7 provides some experimental results obtained from a synthetic but non-trivial example.

4 Summary of the features of jTLM

- 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.
- Model checking models based on SystemC is a research challenge, because of the complex semantics of C++ [14]. jTLM has a clear advantage because it is a thin layer on top of Java. With little effort, we have been able to successfully use both a standard thread-aware debugger and the Java PathFinder [16] model checker.
- jTLM cannot integrate RTL and mixed-level models. This is a design choice: as jTLM is a research tool and an experimentation framework, its API needs not be as fully-featured as that of SystemC.
- In SystemC (and in the cooperative mode of jTLM), preemption points must be manually specified, a daunting task. In addition, because context switching is expensive, there is a tendency towards having as few preemption points as possible. We discuss in Section 5.2 that this leads to many problems involving accuracy (e.g. missing interrupts), implicit atomicity assumptions and even starvation. On the other hand, the preemptive mode of jTLM allows for no implicit atomicity assumptions. Instead, the user is required to explicitly protect the code using some other mechanism that guarantees atomicity.

- 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 [5], making them non-conforming w.r.t. the standard (which we think is unreasonable), or require heavyweight analysis techniques [4].
- 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.
- For modeling actions with a duration in SystemC, users typically resort to an instantaneous action followed (or preceded) by a wait. jTLM introduces a new primitive **consumesTime()**, that enables modeling actions that take time, offering an advantage in terms of trace debugging and code readability. We shall also see in Section 5.1 that this primitive can be exploited to even better parallelization in the preemptive mode.

5 Discussion

5.1 jTLM from the user point of view

Time in jTLM 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.

Primitives 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.

jTLM also introduces a new primitive to model actions that take time: **consumesTime()**. For instance, the pseudo-code for a DMA transfer that takes time proportional to the length of the transfer would be:

```
1 consumesTime(42 * length) {
2     for(int i = 0; i < length; ++i) {
3         read(...);
4         write(...);
5     }
6 }
```

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 overlap 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 instantaneously 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.

Now suppose that in the previous examples the intention was that task A had a duration. With **consumesTime()**, jTLM provides the means to express this. Better yet, because now there is an overlap between A and B/C, we can exploit this by running the tasks in parallel, as shows Figure 1(c).

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. 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 [6] 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.

Techniques like *quantum keeping* [15] seem to avoid the problem, but they always have some sort of "knob" that must be adjusted manually. Worse than that, making the model work correctly becomes a matter of luck in finding a "good" operating point.

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

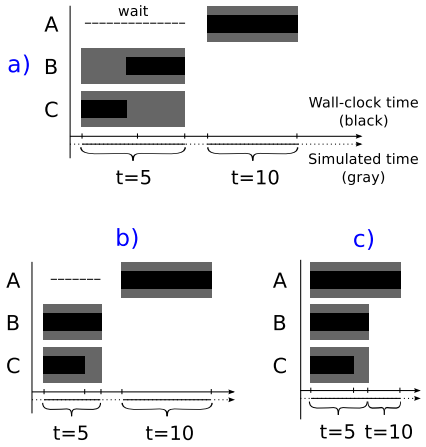

Figure 1: Time and concurrency in jTLM

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 [3] techniques. *Native wrappers* may either wrap the source directly into the virtual prototype, link with a binary compiled into native code [10], or use virtual machines [8].

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*[1], that depend on these details.

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 [13]. 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 1 Consider a model where two processes execute concurrently (i.e. at the same simulation time) the statements **x++** and **x−−**. Obviously, if **x** is a shared variable, it should be protected with some kind of synchronization (like a mutex), and omitting such protection is a bug. In a cooperative simulation, the simulator artificially makes such pieces of code atomic, and the bug remains unnoticed.

A simple experiment shows that jTLM's preemptive mode is able to exhibit the bug both during stress testing (running this example on a loop 100.000 times to increase the probability of an actual race to occur), and using the Java PathFinder state-space exploration tool.

Example 2 Consider a system executing the following two loops in parallel (initially, **x=y=0**):

```
for(i=0; i<N; ++i) {
    x++;
    y++;
}
```

```
for(i=0; i<N; ++i) {
    int l_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.

[1] 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).

6 jTLM algorithm

The algorithm behind jTLM is quite complex. In particular, the preemptive mode requires tricky synchronization at some points. For brevity, we focus here only on the principle of the algorithm, omitting low-level details.

Let $\mathbb{P}$ be the set of all processes, $\mathbb{E}$ be the set of all events, and $\mathbb{T}$ be the time unit. The jTLM scheduler has the following data structures: $AG : \mathbb{T} \to 2^{\mathbb{P}}$ is a priority queue implementing an agenda of processes waiting or consuming time; and $PP : \mathbb{E} \to 2^{\mathbb{P}}$ maps a set of pending processes for each event. Additionally, the scheduler must keep track of the current simulated time ($ct \in \mathbb{T}$), the currently running process ($cp \in \mathbb{P}$), and an integer counter for the number of processes running an instantaneous section of code (pi).

The heart of the scheduler is in the **timeElapse()** method, which is called at the beginning of the simulation and every time pi reaches zero. This method sets the current time to the earliest deadline in AG and then either: (a) in the preemptive mode, wakes up all processes with deadline ct; or (b) in the cooperative mode, picks a random process among those with deadline ct and wakes it up. Here, "waking up" consists of incrementing pi and releasing a semaphore to unblock the process. The simulation ends if **timeElapse()** is called but $AG = \emptyset$.

awaitTime(t) inserts a pair ($ct + t$, cp) into AG, and then decrements pi, checks if $pi = 0$ (in which case **timeElapse()** must be called), and blocks the process by acquiring a semaphore. **awaitEvent**(e) is similar, but increments pe instead of adding an element to AG.

In the preemptive mode, **signalEvent**(e) simply wakes up all $p \in PP(e)$, removing them from the set, and decrementing be by $|PP(e)|$. In the cooperative mode, we must only let one process run among the possible candidates in $PP(e)$, cp and $AG(ct)$. To achieve this, our (unoptimized) algorithm follows a similar procedure but inserts $\big(PP(e) \cup \{cp\}\big) \times \{ct\}$ into AG and then calls **timeElapse()**.

To implement **consumesTime**, we use a technique based on Java's inline unnamed functors, where we intercept the control before and after executing the user code. In the preemptive mode, the implementation of this primitive has only a subtle difference compared to **awaitTime**: before acquiring the semaphore, **consumesTime** first passes control to the user code. While it is running, the pair ($ct + t$, cp) previously inserted into AG acts as a "reservation" ensuring that the simulated time does not advance more than it should. In the cooperative mode, **consumesTime**(t) randomly splits the time interval $[0, t]$ into two calls to **awaitTime**, respectively before and after executing the user code.

7 Experiments

In this section, we briefly present our experiments on a synthetic yet non-trivial example. Our intent is showing to which extent the concurrency described in the model can be exploited in a parallel simulation or is constrained by the simulator.

The system computes hexadecimal digits (nibbles) of π using the BBP formula [2]. The model is organized as one main processor and M processes modeling hardware accelerators, one per task of G nibbles each. The main processor assigns tasks to each accelerator, and waits for them to complete. Each hardware accelerator executes the following pseudo-code:

```
1 while(true) {
2     wait for the task;
3     perform long computation;
4     notify the main processor;
5 }
```

First version In a first version, line 3 is implemented with an instantaneous task followed by a **awaitTime**, similarly to what would have been done for modeling a duration in SystemC. The main processor does the task distribution instantaneously (we discuss this further on). The exact same code works in preemptive and cooperative modes without modifications.

We conducted experiments on a server with 32 Intel Xeon processors. With $G = 200$ and $M = 100$, we compared a pure Java, sequential implementation, and the jTLM cooperative and preemptive versions. While the cooperative mode shows a small performance degradation due to the time taken by the JVM's scheduler and context switches, the preemptive mode can effectively take advantage of the host machine's parallelism and we get an acceleration factor of 24.

Pure Java	jTLM	
Sequential	Cooperative	Preemptive
220s	240s	9s

Second version If we introduce a time delay between each assignment instead of distributing tasks to all accelerators instantaneously, then the parallelization cannot be done anymore. The instantaneous tasks run at different simulation times, and one task cannot start before the previous has completed and let the time elapse: the computation takes 220s.

Preemptive jTLM (with added delay between tasks)	
Without **consumesTime**	With **consumesTime**
220s	20s

The results confirm our intuitive expectations and show the practical benefits of **consumesTime** for parallelization: if we use **consumesTime** instead of **awaitTime**, then the tasks are allowed to overlap although they don't start at the exact same time, and parallelization becomes possible again.

8 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.

Acknowledgments

Many thanks to Nabila Abdessaied, Mohamed El Aissaoui and Rafael Velasquez, who contributed to the initial version of the jTLM implementation.

References

[1] Apple. *iPhone SDK*, April 2010. 1

[2] D. Bailey, P. Borwein, and S. Plouffe. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66(218):903–913, 1997. 7

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the Usenix ATC*, pages 41–41, 2005. 5.3

[4] Y. Bouzouzou. Accélération des simulations de systèmes-sur-puce au niveau transactionnel. Diplôme de recherche technologique, Université Joseph Fourier, 2007. 4

[5] B. Chopard, P. Combes, and J. Zory. A conservative approach to SystemC parallelization. *ICCS 2006*, pages 653–660, 2006. 4

[6] J. Cornet. *Separation of functional and non-functional aspects in transactional level models of systems-on-chip*. PhD thesis, INP Grenoble, April 2008. 5.2

[7] 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*, pages 9–14, March 2008. 3

[8] J. Dike. *User mode linux*. Prentice Hall, 2006. 5.3

[9] R. Dömer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual 2.0*, 2002. 3

[10] F. Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006. 1, 5.3

[11] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz. Test coverage for loose timing annotations. In *FMICS/PDMC*, pages 100–115, 2006. 3

[12] IEEE Standards Association. *IEEE 1666-2005 Standard SystemC Language Reference Manual*, 2006. 2

[13] S. Microsystems. *JSR 133: Java Memory Model and Thread Specification*, 2004. 5.3

[14] M. Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INP Grenoble, December 2005. 1, 4

[15] Open SystemC Initiative. *TLM-2.0.1 User Manual*, July 2009. 5.2

[16] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated software engineering journal*, pages 3–12, 2000. 4