Faithfulness Considerations for Virtual Prototyping of Systems-on-Chip
Giovanni Funchal, Matthieu Moy, Florence Maraninchi, Laurent Maillet-Contoz

To cite this version:
Giovanni Funchal, Matthieu Moy, Florence Maraninchi, Laurent Maillet-Contoz. Faithfulness Considerations for Virtual Prototyping of Systems-on-Chip. 3rd Workshop on: Rapid Simulation and Performance Evaluation: Methods and Tools, Jan 2011, Greece. hal-00559986

HAL Id: hal-00559986
https://hal.archives-ouvertes.fr/hal-00559986
Submitted on 26 Jan 2011

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.
Faithfulness Considerations for Virtual Prototyping of Systems-on-Chip

Giovanni Funchal*,† Matthieu Moy† Florence Maraninchi† Laurent Maillet-Contoz*

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

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

Abstract—Virtual prototypes are simulators used in the consumer electronics industry. They enable the development of embedded software before the real, physical hardware is available, hence providing important gains in speed of development and time-to-market.

Transaction-level Modeling (TLM) is a widely used technique for designing such virtual prototypes. Its main insight is that many micro-architectural details (i.e. caches, fifos and pipelines) can be omitted from the model as they should not impact the behavior perceived from a software programmer’s point-of-view.

In this paper, we shall see that this assumption is not always true, specially for low-level software (e.g. drivers). As a result, there may be bugs in the software which are not observable on a TLM virtual prototype, designed according to the current modeling practices. We call this a faithfulness issue.

Our experience shows that many engineers are not aware of this issue. Therefore, we provide an in-depth and intuitive explanation of the sort of bugs that may be missed. We claim that, to a certain extent, modified TLM models can be faithful without losing the benefits in terms of time-to-market and ease of modeling. However, further investigation is required to understand how this could be done in a more general framework.

I. INTRODUCTION

Developing today’s high-tech consumer electronic devices is a real challenge, mainly because of the complexity and the time-to-market pressure. Most of the functionality of these devices is often grouped into a single integrated circuit, which is then called a system-on-chip (SoC).

The design of such systems is a joint development of custom hardware (the SoC itself) and software (drivers, etc). In this context, a virtual prototype is a model of the hardware intended for simulation. It allows the development of software before the real, physical hardware is available.

A. Virtual prototypes and the design flow

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. RTL models are also executable. They can simulate the behavior of the hardware very precisely.

Nevertheless, the simulation of complex, system-level RTL models is very slow, and they become available too late in the design flow. Using them for software development becomes not feasible in practice [1].

B. Transaction-Level Modeling

Transaction-Level Modeling (TLM) is a widely used technique for designing virtual prototypes [2].

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

The TLM approach relies on the assumption that many micro-architectural details (like caches, fifos, and pipelines) are only optimizations; in other words, they should not impact the behavior of the hardware as perceived from a software programmer’s point-of-view. Consequently, these details are dropped from the virtual prototype, which is expected to remain precise in what concerns the functionality.

Because they are less detailed, TLM virtual prototypes are able to achieve very high simulation speed while requiring far less modeling effort with respect to RTL. Thus, they effectively address the aforementioned complexity and time-to-market issues.

C. Motivation and faithfulness issues

As we shall see in details in Section III, there is a wide variety of modern architectures that implement aggressive optimizations which may change the memory accesses’ semantics (what we call the memory model [3]).

Because the TLM virtual prototypes do not include such architectural details, some behaviors of the software on the real system (and the bugs in particular) may not be reproducible when running on the virtual prototype. We call this a faithfulness issue.

D. The importance of faithfulness

Ensuring that a model is faithful is essential so that software bugs can be found in simulation. Indeed, debugging and reproducibility are some of the major selling points of the TLM virtual prototyping approach.

Many other techniques also require faithfulness. For instance, formal verification [4] and stateless model checking [5] can be used to prove properties of a SoC by analyzing a model composed of the virtual prototype and the embedded software. Of course, if the virtual prototype is not faithful,
any property that has been proven through these techniques cannot be guaranteed to be valid on the real system.

E. Who should be concerned?

Most high-level applications will not suffer from this as they should use APIs that hide the interaction with the hardware. However, when implementing device drivers, low-level synchronization primitives or operating system supporting software, one should be aware of the memory model of the system. It is also the case when implementing lockless data-structures.

Finally, note that the TLM virtual prototyping approach targets low-level software development (there are better techniques to help developing high-level software, like the simulators included in the iPhone [6] and Android [7] SDKs). Hence, the faithfulness of TLM with respect to the memory model concerns the large majority of TLM users.

F. Coping with the issue

At first sight, it could seem that including the relevant micro-architectural details in the virtual prototype is the only way to guarantee its faithfulness. However, this goes against the very principle of Transaction-level Modeling, because it would imply a much higher modeling effort, and lower simulation speed. Moreover, the absence of details in a TLM virtual prototype is also beneficial for software robustness: a piece of software that has been validated on a less-detailed virtual prototype is more likely to run correctly on any real hardware that constitutes a particular implementation of it.

We are currently working on the definition of a methodology that would ensure faithfulness without losing the benefits of TLM in terms of speed of development and time-to-market. We also wish to avoid seriously breaking the way people understand and write transaction-level models by incorporating this methodology within the existing modeling approach.

In this context, we have an initial prototype that we will present briefly later on. However, as a first step, this paper focuses on providing a detailed understanding of the aforementioned faithfulness issues.

G. Contributions and structure of the paper

The main contribution of this paper is related to the study of the faithfulness issues in the current TLM modeling practices. We give precise and simple examples for which a TLM model with respect to the memory model concerns the large majority of TLM users.

Section II presents some technical background on the implementation of virtual prototypes and the current modeling practices.

Section III introduces some common architecture optimizations that are usually not captured in current virtual prototypes.

Section IV shows how this may lead to software bugs being missed.

Section V gives some hints on a promising technique to modify TLM models so that they are faithful with respect to some sorts of architectural details.

II. SOME BACKGROUND ON MODELING

A. The SystemC standard

In this section, we will briefly present SystemC [8], the current industry standard language for developing virtual prototypes. Strictly speaking, SystemC is a C++ library and a discrete-event simulation engine.

In a discrete-event simulation, the state of the model changes only at a discrete set of points in simulated time. Models are composed of processes and events. During simulation, each process can only be running, ready or waiting. A scheduler puts ready processes to run, checks whether waiting processes are ready, and advances the simulated time when all processes are waiting.

There are two kinds of processes in SystemC, SC_THREADS and SC_METHODs. They differ only on the type of stack management and have exactly the same expressiveness [9]. Therefore, and to avoid confusing the term SystemC processes with operating system processes, we will restrict ourselves to SC_THREADS.

SystemC implements cooperative multitasking, i.e. a running SC_THREAD only yields control to others at well-defined points in its execution. This is opposed to preemptive multitasking wherein execution of tasks interleaves on a non-user controlled fashion on uniprocessors and may overlap on multiprocessors [10]. SC_THREADS yield control exactly at the points where they call wait(e) to wait for an event e to be notified by another SC_THREAD; or wait(t) to wait for t units of simulated time.

B. The TLM-2.0.1 library


TLM-2.0.1 itself does not include any modeling guidelines. Instead, it defines an “interoperability layer” which is intended to reduce the engineering effort needed to achieve interoperability. This interoperability layer defines a base protocol, and a generic payload.

The base protocol introduces a communication mechanism called transaction. A transaction transports data between connection points bound to SystemC modules (the sockets). A socket can be either initiator or target depending on whether it initiates the transfers.

The generic payload contains an address, the data, a response status and a command which can be either a read, a write or an ignore (the later is intended for extensions). Additional primitives can be added using an extension mechanism. Interoperability is only guaranteed if these extensions can be ignored by the rest of the components in the virtual prototype.
Transactions are implemented as method calls, which can be of two kinds:

- **Blocking**: the method call returns when the transaction is complete. The code that actually implements the transaction is executed in the context of the calling `SC_THREAD`.
- **Nonblocking**: the method call returns immediately. The code that actually implements the transaction is executed in a different `SC_THREAD`. The caller receives a notification when the transaction is finished.

In this paper, we only consider blocking transactions. The TLM-2.0.1 sockets can automatically convert one type of method call into another, spawning extra `SC_THREAD`s when needed.

### C. Integrating embedded software

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 [2], or use virtual machines [13].

Broadly speaking, all these techniques will:

- transform reads and writes from the software into `read()` and `write()` transactions that are performed by a `SC_THREAD` in a component that corresponds to the processor;
- add calls to `wait()` in order to avoid starvation due to the cooperative nature of the SystemC simulator.

### D. Modeling practices

In this paper, we will focus on virtual prototypes intended for software development. These models, also known as *Programmer’s View* (TLM-PV) [14], allow a fast and accurate execution of embedded software and are widespread in the virtual prototyping community. The main idea is that the embedded software should run, unmodified (or with minimal modifications, depending on the technique), both on the TLM-PV model and on the real chip.

In the sequel, the term TLM will be always referring to TLM-PV. We consider multi-core, bus-mapped, shared-memory systems and we take into account TLM prototypes written in SystemC, with the TLM-2.0.1 library.

A typical virtual prototype is shown on Figure 1. In this figure, the software calls `write()` using one of the techniques highlighted on the previous section. The data and address are packaged into a transaction which is forwarded through the interconnection and to the corresponding implementation in the target component. `wait()` statements may be placed to model the time taken before and after the code that implements the effect of the method call. No details such as arbitration, caches, fifos are part of the virtual prototype, favoring simplicity and simulation speed.

![Fig. 1. A typical virtual prototype](image)

### III. Modern architectures and relaxed consistency

Branch prediction [15], out-of-order execution [16], the pipeline [17] and compiler optimizations [18], [19] are all designed to improve performance of each individual thread by reordering reads and writes to the memory. Such reorderings are invisible to the issuing thread by construction, but can lead to undefined behavior in a multi-threaded system.

On multi-core architectures, when a processor issues a write, the value is usually first kept in a cache. The cache provides a notion of locality, saving time in situations such as when the processor writes to the same address more than once.

The requests are then collected in a hardware `write buffer`, and pushed to memory at a later time. This is often designed to be very efficient, and to exploit the parallelism, but there is a drawback: the order in which requests are issued is not necessarily the order in which they are performed.

Different architectures provide different guarantees on what can be reordered. The next section will illustrate these ideas through an example.

#### A. Example 1

Figure 2 presents part of the code from the implementation of a lock algorithm (Dekker) for critical sections [3]. It involves two processors (P1 and P2) and three variables in the memory (x, y and z). P2’s assertion will be violated if P1 writes to the variable z, that is, if both processors enter the critical section.

Before entering critical section, P1 writes 1 to x and reads y. The code of P1 relies on the assumption that, if the read of y returns 0, then P2 has not yet tried to enter the critical section. P2 operates in a similar way. While this is code works in a sequentially consistent [20] system, implementing systems that provide this view involves serious compromises in performance [3].

A more realistic architecture is depicted in Figure 4. This hardware platform includes a very common micro-
avoid this bug. This can be done, for instance, by enforcing
1 is embedded into two
(described in Section II-C), the software from the Example
Example 1 is constructed: Using a simple wrapper technique
A. Constructing a virtual prototype for Example 1

B. A possible fix to Example 1

Fortunately, when hardware designers introduce such optimi-
zations, they also provide special operations that allow to
avoid this bug. This can be done, for instance, by enforcing
that straightforward modifications that could be proposed to
solve the problem do not lead to practical solutions.

Take for instance the mfence() operation, which is present
in architectures like amd64 and x86 (with the SSE
instruction set extension) [21]. Its semantics guarantees that,
upon completion, all previous (in the program order) opera-
tions are visible to all other processors. Then, Figure 3 shows
a possible fix of the code of Figure 2 using mfence(). Actual
implementations of mutual exclusion algorithms make heavy use of this kind of special operations.

IV. DISCUSSION ON THE EXAMPLE 1

In this section, we discuss what happens when the soft-
ware from the Example 1 is embedded on a SystemC/TLM
virtual prototype. We show that, with the current modeling
practices, this virtual prototype will hide the bug that we
had exposed in this software in Section III-A. We also argue
that straightforward modifications that could be proposed to
solve the problem do not lead to practical solutions.

A. Constructing a virtual prototype for Example 1

First, we will show how a typical virtual prototype for the
Example 1 is constructed: Using a simple wrapper technique
(described in Section II-C), the software from the Example
1 is embedded into two SC_THREADS, respectively $T_1$ and
$T_2$, as shown in Figure 5. Then, these SC_THREADS are put
into two SystemC components that represent the processors.

architectural feature known as write buffers, which is present
in most of today’s processors. In this system, writes are
allowed to be delayed in such a way that both reads by
processors $P_1$ and $P_2$ will return 0. Consider for instance
the execution shown in the same figure.

B. A possible fix of the code of Figure 2 using mfence()

A third component, representing the memory, is modeled
following the guidelines presented in Section II-C. It has an
array storage which effectively stores the data. Calls to
wait() in the implementation are used to model the time
that it takes to read and store data in the memory.

For brevity, we have omitted from Figure 5 the imple-
mentation of the base classes Initiator and Target. The
Initiator class should declare a TLM initiator
socket; transform read()s and write()s, performed by the
software, into transactions; and forward these transactions
through the aforementioned socket. The class Target should
declare a TLM target socket; receive transactions from this
socket; and forward reads to read_impl() and writes to
write_impl().

The last step is to instantiate and connect the components
using a bus model that routes transaction at addresses x, y
and z to the memory.

B. Possible executions of the virtual prototype

Now, we need to understand what happens during the
simulation of this virtual prototype.

We call a step the sequence of instructions executed by a
SC_THREAD from the point the scheduler puts it to execute
to the point where it finishes execution or yields control back
to the scheduler by calling wait().

The set of all possible executions of the virtual prototype is
obtained by interleaving the steps of $T_1$ and $T_2$. The Figure 6
respectively the steps of $T_1$ and $T_2$ with dashed and solid
lines. In this figure, there are paths that lead to either 1) $P_1$,
or 2) $P_2$, or 3) none of them deciding to enter the critical
section. However, there is no possible execution that leads
to both $P_1$ and $P_2$ deciding to enter the critical section.

The conclusion is that, when executing the software on
this virtual prototype, the mutual exclusion property appears
correct. Nevertheless, we have shown in Section III-A that
such software has a bug that shows up on the real system.
This demonstrates that the current TLM modeling practice
may hide software bugs.
SC_MODULE(P_1): public Initiator {
    SC_CTOR(P_1) {
        SC_THREAD(T_1);
    }
    void T_1() {
        write(x, 1);
        if(read(y) == 0) {
            // Crit. section
            write(z, 1);
        }
    }
};

SC_MODULE(P_2): public Initiator {
    SC_CTOR(P_2) {
        SC_THREAD(T_2);
    }
    void T_2() {
        write(y, 1);
        if(read(x) == 0) {
            // Crit. section
            assert(read(z) == 0);
        }
    }
};

SC_MODULE(Mem): public Target {
    data[SIZE] storage;
    void read_impl(addr a) {
        wait(42);
        data d = storage[a]; // effect
        wait(23);
        return d;
    }
    void write_impl(addr a, data d) {
        wait(42);
        storage[a] = d; // effect
        wait(23);
    }
};

Fig. 5. Pseudo-code of a typical virtual prototype implementation of the Example 1

![Diagram of possible executions of Example 1](image)

Fig. 6. Possible executions of Example 1

Let us have a look at the bug again. We have shown in Section III-A that in some architectures, $P_2$’s write to $y$ could be delayed by a write buffer in such a way that $P_1$ will read the previous value ($y == 0$).

The key to understand why the virtual prototype cannot reproduce this behavior is in what we call the effect of a transaction: the portion of code that effectively implements its behavior. The lines that implement the effect of transactions in Figure 5 are marked as such. These lines will be executed somewhere between the call and the return of each transaction. Because of the way we have implemented the memory, the effect of transactions will become visible for all the components at the same time. There is no way to have one component observe a value, and another component observe another (previous) value, which is a necessary condition to reproduce the bug.

V. WRITING FAITHFUL MODELS

The first (non-)solution that we would like to mention is to include all the relevant micro-architectural details in the virtual prototype. This alternative increases complexity, requires a very big modeling effort, and leads to poor performance, contradicting the advantages of using a TLM model in the first place.

A much more reasonable approach is to model the actual semantics of the underlying architecture not in terms of implementation (what a particular cache does), but in terms of specification (what caches may be expected to do). There is a large literature on memory consistency models, a field of research that tries to understand and specify formally how software threads interact with memory on complex architectures. Some of these models have operational semantics which describe the meaning of the program as a non-deterministic sequence of steps, much easier to understand and implement than hundreds of pages of documentation.

Therefore, a practical way to write a faithful model is to have writes and reads implement the operational semantics of a memory model that is equivalent (or weaker) than that of the real system.

We have done some experiments aiming at the implementation of such operational semantics in TLM models. The idea is to enrich the structure of a TLM model by inserting additional components that play the role of write buffers. Their behavior is non-deterministic: they either delay the
Initially \( x = y = 0 \)

\[
P_1 \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad