Comparison of TLM2-Quantum Keeping and TLM$^+$-Resource Modeling with regard to Timing in Virtual Prototypes

Wolfgang Ecker*, Volkan Esen*, Rainer Findenig†, Thomas Leitner‡ and Michael Velten§
*Infineon Technologies AG
85579 Neubiberg, Germany
Email: *Firstname.Lastname*@infineon.com
†Upper Austrian University of Applied Sciences
Softwarepark 11, 4232 Hagenberg, Austria
Email: rainer.findenig@fh-hagenberg.at
‡DICE GmbH & Co KG
Linz, Austria
Email: Thomas.Leitner@infineon.com
§Infineon Technologies AG
Technische Universität München
Email: Michael.Velten@infineon.com

***Abstract*—Virtual Prototypes (VPs) based on Transaction Level Modeling (TLM) have become a de-facto standard in today's SoC design, enabling early SW development. However, due to the growing complexity of SoC architectures, full system simulations (HW+SW) become a bottleneck especially if a high timing accuracy is required. Since timing influences system functionality and since satisfaction of timing constraints is essential in embedded systems, a quite accurate timing representation is important in VP modeling. Hence, it is necessary to apply new modeling styles which allow for further abstraction but provide the required timing accuracy.**

This paper compares two existing modeling concepts with regard to simulation speed and timing accuracy One referred to as TLM2 Quantum Keeper (QK) and the other referred to as the TLM$^+$ Resource Model (RM).

I. Introduction

Transaction Level Modeling (TLM) with SystemC has been adopted widely in the development of embedded systems. Its main use case is to model so-called Virtual Prototypes (VPs). The main intent behind modeling VPs is to parallelize the development of hardware (HW) and the software (SW) which is supposed to be delivered in conjunction with the final product. This is possible due to two major reasons:

1) Early availability of the virtual model due to its abstraction
2) Fast simulation execution speed, with low turn-around times in SW development

Especially, the latter fact is becoming harder to achieve with the growing complexity of embedded systems. More and more components (HW and SW) are integrated into one system and need to be simulated. Hence, it is necessary to counteract the drawback in simulation speed by developing new abstraction techniques, such that the whole process is not endangered in the future.

In this paper we compare two new abstraction techniques with regard to simulation speed and timing accuracy. These abstraction techniques are referred to as Quantum Keeping and the other referred to as the TLM$^+$ Resource Model (RM).

The paper is structured as follows. First, we provide an overview on related work in this field. Following that, we formulate key requirements which shall serve as the basis for the comparison of the mentioned approaches. In connection to that we provide an overview of both techniques followed by the actual comparison.

II. Related Work

Transaction Level Modeling is the de-facto standard for creating Virtual Prototypes. With TLM2, OSCI has released a standard which defines different communication concepts for modeling hardware interfaces and bus protocols [1]. Especially, timing abstraction techniques were introduced to boost simulation performance, e.g., using a time quantum which is explained in more detail later.

The interface concepts of the TLM2 standard are complementary to both approaches compared in this paper.

The new TLM$^+$modeling style for data flow abstraction is presented in [2] and [3].

Other approaches exist, which deal with different abstraction techniques. However, these approaches aim at different goals than the ones discussed in this paper:

SystemQ, an approach presented in [4], provides high-speed simulation models based on queuing networks. These models are mainly used for system performance estimation and cannot

be used for SW development as they do not contain system functionality.

Several approaches as presented in [5], [6], [7] target the development of fast and timed Real-Time Operating System (RTOS) simulation models to increase the simulation speed. These approaches are increasing the simulation speed due to native software execution of the RTOS models in combination with e.g., SW timing annotations or task scheduler models.

The authors of [8] are presenting the generation of timed OS simulation models using delay annotation for the SW execution. These OS models are communicating through bus functional models with the HW model. The OS models presented in [9] are dealing with synchronization problems of SW and HW. A timed HW/SW co-simulation at an early design stage which allows simulation performance up to 3 orders of magnitude faster than using an ISS is presented in [10]. Other approaches target automatic timing annotations of the native SW execution. A compiler based approach is presented in [11]. In [12] the SW execution time is derived from a static analysis and combined with dynamic runtime information in order to achieve Cycle Approximate (CA) simulations of the native SW execution. A combination of instruction set simulation and an abstract RTOS is presented in [13].

However, none of these approaches deal with the abstraction of HW models.

III. Abstraction Requirements

This section discusses requirements an abstraction methodology needs to fulfill. Each requirement is listed and discussed in detail.

As previously described the motivation of this paper is the comparison of two methodologies for modeling high performance embedded HW/SW systems which provide the necessary simulation speed to deal with the growing complexity of today's SoC designs. The focus of the abstraction methodology shall be to provide a platform for developing higher level embedded software (eSW).

R 1: **Faster simulation speed than today's TLM VPs**

From the motivation to simulate complete embedded HW/SW systems follows that the abstraction methodology must provide a faster simulation speed than TLM. This has two reasons:

- The steadily growing complexity of SoCs requires more simulation effort for the hardware model and in connection to that more complex embedded software needs to be simulated.
- The abstraction methodology shall support the development of higher level software, e.g., protocol stacks, operating systems, etc.

R 2: **Sufficient timing accuracy for eSW development**

The new modeling style has to provide a suitable timing accuracy for the virtual prototype of the hardware and for the development of embedded software. Often embedded software has to follow real time conditions for timing critical hardware interactions and hence, it is essential that the timing behavior of the embedded software can be simulated with real time restrictions. Furthermore, originally working SW must not break due to the abstraction of timing behavior of the underlying HW model.

R 3: **Ability to co-simulate with existing TL models**

A co-simulation with an existing TL model shall be provided by the abstraction methodology. A TLM module shall be simulated within an abstracted system and vice versa. Furthermore, it shall be provided to co-simulate different abstracted systems within one simulation context. The ability to co-simulate offers following advantages:

- Integration of third party IP which does not support the new modeling style.
- Step by step abstraction of existing TLM virtual prototypes
- Selective abstraction of bottleneck modules for improving the overall simulation speed.

R 4: **Constant complexity of the abstraction technique**

The abstraction technique shall provide a constant complexity, i. e., the abstraction technique does not introduce further complexity to VP modeling if the system complexity increases, e.g., more initiators, target modules or buses are added to the system.

IV. TLM2 Quantum Keeping

In this section we describe the Quantum Keeping (QK) abstraction technique introduced with the latest TLM2 standard [1].

A. Overview

Context switches induced by SystemC threads suspending and waking up again are very expensive with regard to simulation execution speed. When a SystemC thread suspends, its complete state has to be stored, such that when the thread wakes up again, it can continue from the exact point where it has been suspended. The key principle of QK is to reduce the number of context switches within a simulation in order to speed up the overall simulation execution time.

Basically, using QK abstraction, all processes which initiate transactions are modelled in such a way, that the actual transaction does not consume any time. Instead, a transaction returns a time value, which yields how much time the transaction would have consumed. The corresponding initiator accumulates these times, to a so-called LocalTime. I.e., the initiator stays ahead of the actual simulation time. However, as the initiator accumulates the LocalTime it is the only active process in the simulation, i.e., other processes in the system would starve. Hence, it is necessary to suspend the initiator's process once in a while, to make room for other processes to be executed. Using the QK abstraction a so-called global time quantum has to be set statically for a simulation. Once, an initiator's LocalTime has reached or surpassed the quantum, it needs to suspend by calling a *wait*-statement and passing the current local time to it as argument. This way, other processes

can be executed by the SystemC simulation kernel. Once, the initiating process wakes up again, when the rest of the system has caught up with its LocalTime offset, the initiator resets its LocalTime and the whole procedure starts again.

Figure 1 describes the basic principle of the QK abstraction.

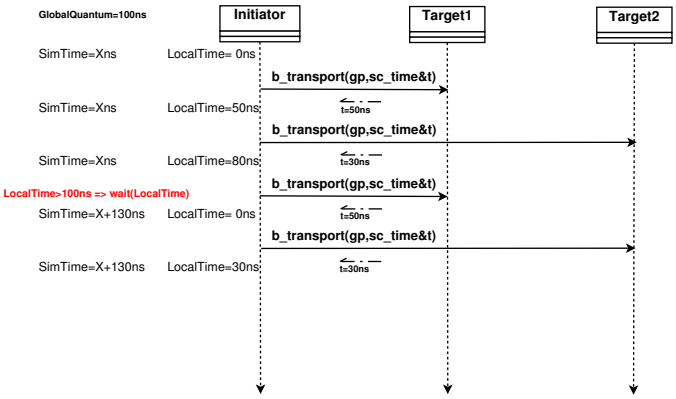

Fig. 1. Quantum Keeping Principle

The TLM2 language reference manual recommends to use the blocking transport API, when employing the QK abstraction. This is clear, because of the necessity of resynchronizing an initiator to the rest of the system by synchronizing its LocalTime with the simulation time. The blocking transport API, already provides the infrastructure for handling the simulation time a transaction would consume, by providing a reference argument in the signature of the blocking transport API's *b_transport*-function. A target which receives the call of *b_transport* is then responsible to write a sensible value to this argument, in order to notify the initiator of the time the transaction needs to consume. This way it is also possible to model state-dependent time consumption for each target.

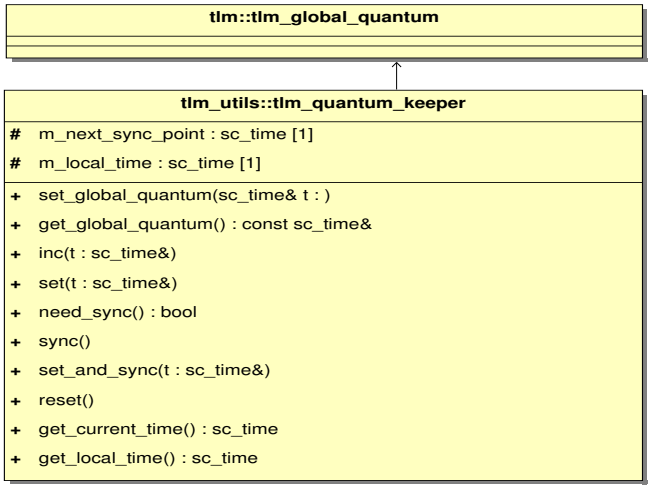

Fig. 2. Quantum Keeper API

B. Quantum Keeper API

To implement the QK abstraction the TLM2 library comes with a utility class called *tlm_quantumkeeper*. Figure 2 depicts the functions and attributes of this class. The class offers an API to manage the quantum keeping. For instance, it allows setting and retrieving the global quantum which is a singleton object and thus valid for any existing initiator. Furthermore, the class offers functions which implement the LocalTime accumulation and retrieve the LocalTime as well as provide means to check if a synchronization is required or to actually carry out synchronization. When modeling with the QK abstraction, the user shall use these provided functions, so that a common management of the LocalTime is guaranteed. The quantumkeeper class also has two attributes, one to hold the current LocalTime value and another to hold the next simulation time at which a synchronization is required.

Figure 3 shows a more fine-grained version of the sequence diagram from Figure 1. The most important calls to the quantum keeper utility class are shown.

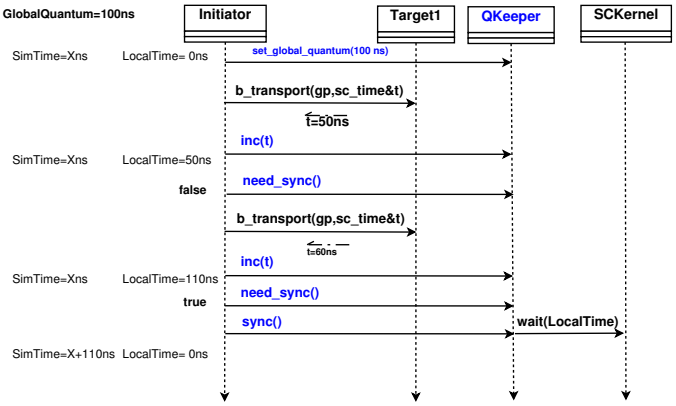

Fig. 3. Using the Quantum Keeper

As depicted in Figure 3, every time a transport function is called the returned time offset needs to be added to the LocalTime by calling the function *inc* of the quantumkeeper class. Directly, after this call it is necessary to check whether a synchronization is required, by calling the function *need_sync*. If required, the synchronization has to be performed by a call to function *sync*, which in turn calls the SystemC function *wait* and also resets the tallied LocalTime.

C. Synchronization Points

The TLM2 language reference manual states, that when using the QK abstraction it is still necessary to introduce other synchronization points than the one that represents reaching or surpassing the time quantum; for instance, when a target needs to wait for another process to be executed in order to determine the time the transaction would consume. Synchronization points are also important for avoiding corrupt timing.

Figure 4 shows a scenario where using the quantum keeper methods can lead to false timing computation if no explicit synchronization is performed.

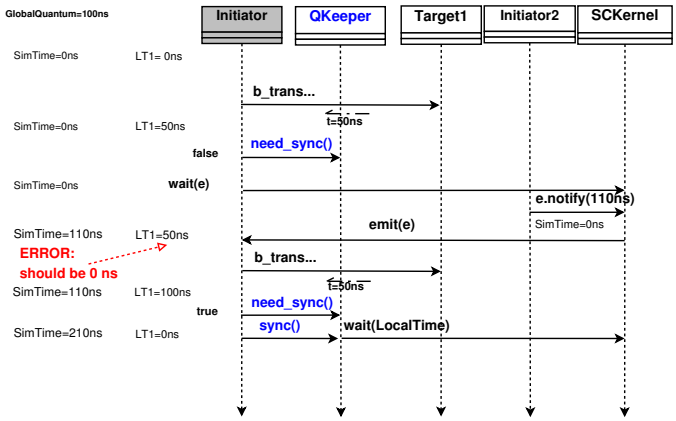

Fig. 4. Using the Quantum Keeper

The issue is that the quantum keeper has no information about a potential progress of the real simulation time. It does not tally the real SystemC simulation time. The local time offset is incremented with every transaction by the associated quantum keeper, which only considers the current value of the local time offset for determining whether the initiating process needs to synchronize. Especially in models, where a process consists of several layers of functions, it is very likely that a suspension takes place in order to synchronize for instance to an event emitted by another process , i.e., the SystemC simulation time can progress without being induced by the quantum keeper. In this case however, as the quantum keeper does not keep track of the SystemC simulation time the accumulation of the local time is continued, although the simulation time has changed with regard to the beginning of the current quantum. Hence, the local time offset is no longer an offset to the correct simulation time, which of course, leads to higher SystemC simulation time values than is expected. In Figure 4 the process *Initiator2* emits an event *e* once it is activated (activation is as soon as *Initiator1* suspends with *wait(e)*). As the event is scheduled at simulation time 0 to 110 ns later, the scheduling logically takes place before the current LocalTime of *Initiator1*. However, as soon as *Initiator1* receives the event it continues performing transactions until it reaches the quantum, which then is basically added to the already progressed simulation time. Hence, in the end the simulation time is later than it should be, as the timing overlap between scheduling of event *e* and the LocalTime of *Initiator1* was counted twice. The solution to this problem would be to make *Initiator1* synchronize once more before the call to *wait(e)*. This however requires, that the quantumkeeper utility class is accessible from anywhere along the layered *Initiator1*. However, it would be more safe to have an automatic way to perform this kind of synchronization. This could be achieved by also tallying the simulation time, with regard to when a new quantum begins.

V. TLM+ Resource Modeling

In the TLM$^+$ modeling style data blocks are transferred instead of single words. In contrast to state-of-the-art TLM modeling style, the data blocks are not related to infrastructure details as e.g., bus transaction burst size. The data blocks are rather related to logical entities such as OS buffer sizes or data content sizes such as pictures. This abstraction technique leads to a huge speedup since the HW/SW and HW/HW interactions are reduced to block accesses at OS level. Nevertheless, the programming technique is quite close to the concepts applied in TLM2. In order to gain high timing accuracy for TLM$^+$ block transfers, computation of functionality and timing is strictly separated. Timing is computed in a so called resource model (RM), which is considering the resources and infrastructure (e.g. buses, CPU) of the complete SoC architecture and the resulting resource conflicts for the computation of time. This RM performs timing corrections and actively reschedules pending transfers to achieve a good timing accuracy at TLM$^+$ level. The TLM$^+$interface abstraction and TLM$^+$timing abstraction concepts are described in the following sections.

A. Interface Abstraction

The TLM$^+$interface abstraction is defined by transferring blocks corresponding to semantic data of the software application instead of transferring the data as sequences of single word transactions to the hardware model. This abstraction reduces context switches depending on the block size of the application specific interface. The application specific interface is defined by merging software and hardware at the device driver interface in order to transfer buffers of the software application as complete block through the device driver instead of splitting the buffer into hardware specific transfer units like words or bus bursts. This abstraction of the HW/SW interface requires that the software is executed natively on the simulation host in order to support block transfers. For this purpose, a generic SystemC EMUCPU module was developed which provides native software execution including support for interrupt handling and SW timing annotation. The behavior of the EMUCPU can be customized for specific processor architectures such as ARM, MIPS, etc.

The HW/SW interface for register and bit field accesses is modeled through bus read and write functions which are wrapped from SystemC to the C-software. Furthermore, the HW/SW interface is extended by two additional read and write functions in order to transfer buffers of the application software as block to the hardware model. However, the configuration of hardware modules still happens through writing their configuration registers using the word mode interface but transferring data is realized through the application specific interface using the abstracted read and write functions for block transfers.

Furthermore, in TLM$^+$the hardware structure of the system is preserved but the transaction interfaces are abstracted to support block transfers. Since the generic payload of TLM2 already defines a pointer for the data value, the OSCI TLM2 library is fully compatible and can be used for TLM$^+$. Only

the generic payload needs to be extended by a *count* member which corresponds to the size of the transferred data block in bytes.

A more detailed description of the TLM$^+$interface abstraction concepts is given in [2].

B. Timing Abstraction

When raising the abstraction from word to block transfers the timing granularity gets reduced to the block size. The transfer timing of a block can be calculated using the protocol information of the generic payload and the block size given in the *count* value. The problem is that a block transfer cannot be interrupted; hence, in case of resource conflicts the data of a transferred block gets valid at a wrong time. Figure 5 shows an example of two conflicting data transfers. In case of the first transfer *T1* the CPU sends 10 words over the bus to the AES module starting at 0 ns. The second transfer *T2* starts at 9 ns when the DMA transfers 10 words to the memory module. Every word transfer keeps each module occupied for 1 ns. Hence, the duration of transferring one word is 3 ns. The transfers *T1* and *T2* result in a bus conflict and the CPU transfer *T1* is interrupted because of the higher priority of the DMA transfer *T2*. Transfer *T1* is resumed when *T2* is finished as it is shown in Figure 5. Hence, the data of *T1* is valid at 60 ns. In case of the TLM$^+$interface abstraction, the 10 words are transferred as a single data block. Hence, *T1* cannot be interrupted and its transferred data gets valid at 30 ns instead of 60 ns also shown in the figure. This timing error can lead to a fatal system behavior.

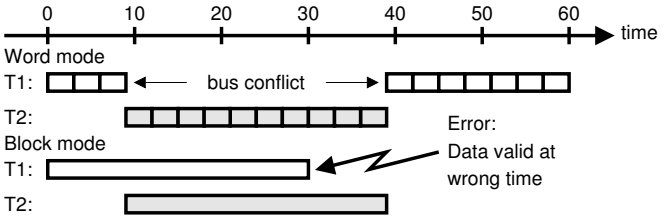

Fig. 5. Timing of resource conflicts for word and block transfers

The TLM$^+$timing abstraction deals with this problem by introducing the concept of the separation of timing and control. The functionality is still modeled by the HW modules of the VP but the timing is handled separately by a so called resource model (RM). This RM takes care of scheduling all data transfers while taking resource conflicts into account. The interface of the RM provides functions for registering initiators and resources during the construction phase. Furthermore, a priority can be assigned to each initiator and an user definable arbitration scheme can be assigned to each resource. Figure 6 gives an overview of the registered initiators and resources of our example system. The CPU initiator *I1(0)* has priority 0 and the DMA initiator *I2(1)* has priority 1. A priority based arbitration scheme is assigned to each resource which defines that a transfer of lower priority is interrupted by a higher transfer.

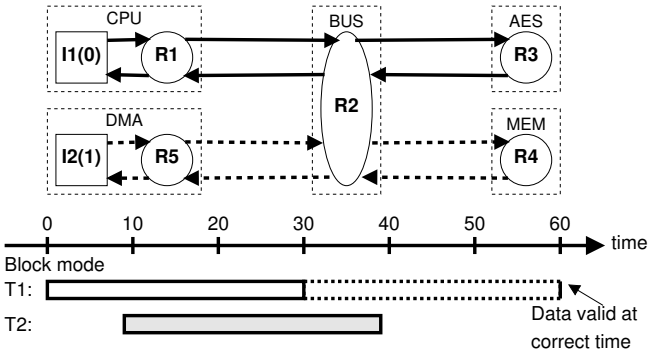

Fig. 6. Resource model and conflict handling

During the simulation, every time an initiator initiates a transfer, the transfer has to be registered at the RM using the *registerTransfer* function which returns an unique transfer ID (TID). This TID is passed as member of the TLM$^+$payload extension. When a transfer passes a resource, the resource has to call the nonblocking *requestResource* function by passing the TID, its registered resource ID, and the requested time as arguments. This function has to be called on both, the forward path and backward path of the transaction. This allows the RM to detect all resource conflicts of simultaneous transfers.

Furthermore, the TLM$^+$methodology defines two synchronization points where the *suspendTransfer* function has to be called. The first point is before the data gets processed by the target module and the second point is on the backward path before the transfer returns to its initiator. The *suspendTransfer* function suspends the specified transfer until its resume event occurs which gets scheduled by the RM. In case of a resource conflict the RM cancels and reschedules the transfer resume events depending on the arbitration scheme of the conflicting resource.

In case of our example the resource model detects the conflict of *T1* and *T2* at the bus resource and reschedules the resume event of the CPU transfer *T1* to 60 ns as it is shown in Figure 6. Hence, the data block becomes valid at the AES target module at the correct simulation time. At the end of a transfer, the initiator has to remove the transfer by calling the *removeTransfer* function of the RM.

VI. Comparison

In this section we provide a comparison of the explained abstraction techniqes based on the requirements introduced in section III.

R1 Faster simulation speed than today's TLM VPs

Quantum Keeping: Generally, the QK abstraction provides an improved simulation execution time when comparing it to regular TLM modeling, such as PV+T using the TLM2-standard. As a rule of thumb it can be stated that the higher the value for the global quantum is chosen, the faster the simulation execution gets. Well, this tendency of course has its limits, because the higher the global quantum is set the more

non-reactive the overall system becomes. This is because the process which handles the quantum keeping will dominate the scheduling of the SystemC kernel, and other processes would starve.

TLM$^+$: TLM$^+$in general provides a speed up in simulation execution time as well. Here, the impact on the execution speed is directly coupled to the number of how many transaction can be grouped to a single transfer. This means that the higher the package size can be set due to SW or simulated OS constraints the higher the speed improvement becomes. However, if transactions cannot be bundled the TLM$^+$abstraction can become even slower than regular TLM modeling, because one transfer requires two process suspensions.

R2 Sufficient timing accuracy for eSW development

Quantum Keeping: The higher the quantum is set the higher the likelihood of introducing timing diversions becomes, as we interfere with the natural scheduling of the SystemC kernel. As an example, let us consider a small system with an instruction-set simulator (ISS) which executes a program and is capable of processing interrupts. The TLM initiator interface of the ISS employs quantum keeping when performing bus accesses and executing instructions. The interrupt can be considered as a concurrent input to the ISS. In this setup however, the Software (SW) being executed on the ISS can only be interrupted at the granularity of the chosen time quantum, as the initiator process which runs the SW execution will only suspend at the given quantum. In other words, the process which would set the interrupt can only be executed once the quantum kept initiator process has suspended. Therefore, the relation of when an interrupt is processed by the ISS and the SW state at that time, will be different if the QK abstraction is used. Furthermore, timing becomes less accurate if more than one initiator exists in a system in combination with resources being shared among initiators. Since, an initiator is active at most until its local time has reached the quantum and another initiator can only start after that, from a resource point of view it is not possible to detect that two initiators have accessed the resource at the same LocalTime (both initiators access the resource when their LocalTimes overlap logically). I.e., arbitration schemes on the shared resources would not take effect and hence, arbitration related timing influences on the initiators would not be considered.

TLM$^+$: In TLM$^+$the usage of a host code execution model for a cpu to run the SW is required. The thread that runs the SW and initiates TLM$^+$- transfers is suspended twice per access to the bus interface. Hence, with this approach, the cpu can only detect incoming interrupts at the granularity of transfers, or if the SW contains explicit synchronization points with the SystemC Kernel. However, this limitation is not as critical as in the case of the QK abstraction, because the TLM$^+$abstraction adapts itself to the current SW state. I.e., if the SW initiates transfers which also involve interrupt based communication to peripherals, these would be covered by the timing correction of the TLM$^+$resource model. With regard to systems with shared resources the TLM$^+$resource model detects when a shared resource is attempted to be used by more than one initiator at the same time. Furthermore, it also employs correction schemes to compute a timing which considers priorities of initiators and validates the transferred data according to the corrected initiator times. I.e., the resource model manages the execution times of initiators over all existing system resources.

R3 Ability to co-simulate with existing TL models

Both described abstraction techniques utilize standard TLM2 interfaces. Therefore, they can, in principle, be connected to regular TLM2 models over TLM2 ports. However, both abstraction techniques would require to insert corresponding transactors which bridge the gap in abstraction when connecting abstract TLM models with regular TLM models.

R4 Constant complexity of the abstraction technique

Quantum Keeping: When employing the QK abstraction extreme care has to be taken, when it comes to synchronizing initiators to rest of the system, if synchronization is required beyond reaching the time quantum. For instance, whenever an initiator needs to react to a SystemC event it has to be made sure that the LocalTime of its quantum keeper is synchronized first. As waiting for an event can consume simulation time, the time that has passed between beginning to wait for an event and the actual emission of the event needs to be considered for the local time as well. Otherwise one might accumulate times which actually would have been in parallel to this time gap. Furthermore, when using the QK abstraction it also has to be considered how targets are modelled. An initiator for instance has to prematurely synchronize its LocalTime when accessing a target where the access leads to temporal side effect, i.e., the target emits events upon an access. If such a target was accessed twice within one quantum, the event emission of the first access would be cancelled, which can lead to unexpected behavior and break the overall system.

TLM$^+$: In TLM$^+$the complexity of the abstraction remains constant. This is due to the generic handling of resources and transfers. The resource model has no limits with regard to managing resources, initiators, or transfer sizes. Furthermore, the process of modeling always follows the same pattern:

- Registration of initiators and resource at elaboration time
- Registration of each new transfer with the resource model, by the initiator
- At every hop (forward and backward) of a transfer a call to *requestResource* is made
- Suspension of a transfer always at the destination target before commiting processing the payload data and immediately before returning to the initiator
- Deregistration of a transfer at the resource model

Arbitration is modeled per resource and reusable among resources. Hence, it is independent from the overall system.

VII. Conclusion and Outlook

In this paper we gave a brief overview on two different abstraction techniques and highlighted their key features. Both

techniques tackle the necessity of improving simulation speed of VPs in order to preserve the usability of virtual prototyping for early development of SW. For comparing both abstraction techniques we defined four main requirements which we deduced from this overall goal. While both abstraction techniques have their strengths and weaknesses with regard to these requirements, the comparison has shown, that the QK abstraction involves more considerations at modeling and induces a high effort to provide a stable and easy to use infrastructure. Furthermore, it has shown that the TLM$^+$abstraction besides providing a more application driven methodology, can handle even resource conflict detection and conflict resolution with regard to timing. Our future work, hence will focus on improving TLM$^+$further into the direction of handling SW-timing and also the use of TLM$^+$models as reference models in the verification of TLM and even RTL models.

The work presented in this paper is partially funded by the German Federal Ministry of Education and Research within the context of the SANITAS project (01M3088).

REFERENCES

[1] *OSCI TLM-2.0 Language Reference Manual*, Jul. 2009.

[2] W. Ecker, V. Esen, R. Schwencker, T. Steininger, and M. Velten, "TLM+ Modeling of Embedded HW/SW Systems," in *Design, Automation and Test in Europe (DATE)*, Dresden, Germany, March 2010.

[3] W. Ecker, V. Esen, R. Schwencker, and M. Velten, "Defining TLM+," in *Design & Verification Conference & Exhibition (DVCon)*, February 2010, pp. 1–8.

[4] S. Sonntag, M. Gries, and C. Sauer, "SystemQ: Bridging the gap between queuing-based performance evaluation and SystemC," *Design Automation for Embedded Systems*, vol. 11, 2007.

[5] A. Gerstlauer, H. Yu, and D. D. Gajski, "RTOS Modeling for System Level Design," in *proceedings of Design, Automation and Test in Europe Conference*, 2003.

[6] R. Le Moigne, O. Pasquier, and J. P. Calvez, "A Generic RTOS Model for Real-time Systems Simulation with SystemC," in *proceedings of Design, Automation and Test in Europe Conference*, 2004.

[7] H. Yu, A. Gerstlauer, and D. D. Gajski, "RTOS Scheduling in Transaction Level Models," in *1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2003.

[8] S. Yoo, G. Nicolescu, L. Gauthier, and A. A. Jerraya, "Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC Design," in *proceedings of Design, Automation and Test in Europe Conference*, 2002.

[9] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. A. Jerraya, "Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer," in *proc. of Design, Automation and Test in Europe Conference*, 2003.

[10] A. Bouchhima, S. Yoo, and A. A. Jerraya, "Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model," in *proceedings of Asia and South Pacific Design Automation Conference*, 2004.

[11] A. Bouchhima, P. Gerin, and F. Pétrot, "Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation," in *Asia and South Pacific Design Automation Conference*, 2009.

[12] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-Performance Timing Simulation of Embedded Software," in *proceedings of Design, Automation and Test in Europe Conference*, 2008.

[13] M. Krause, D. Englert, O. Bringmann, and W. Rosenstiel, "Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation," in *proceedings of 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2008.