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 reducing this benefit. Hence, it is necessary to develop modeling styles which allow for further abstraction beyond the currently applied TLM methodology. This paper introduces such a modeling style, referred to as TLM+. It enables a higher modeling abstraction through merging hardware dependent driver software at the lowest level with the HW interface. Thus, sequences of HW transactions can be merged to single HW/SW transactions while preserving both the HW architecture and the low-level to high-level SW interfaces. In order to maintain the ability to validate timing-critical paths, a new resource model concept is introduced which compensates the loss of timing information, induced by merging HW transactions. Experimental results show a speed-up of up to 1000x at a timing error of approximately 10%.

I. INTRODUCTION

Virtual prototyping based on transaction level modeling (TLM) has gained acceptance in today’s System-on-Chip (SoC) design. Virtual prototypes are used for several purposes:

- Co-Verification of Register-Transfer-Level (RTL) models
- Architecture Exploration
- Early Software (SW) development

Especially, the latter two use models are endorsed by the fact that the simulation speed of a VP is much higher due to its higher level (compared to RTL) of abstraction.

In order to be able to develop SW on a VP and to successfully port it to the actual silicon, it is necessary that a VP provides the same interface to SW as its silicon implementation. The higher simulation speed is gained due to abstraction with regard to communication, timing behavior, and data processing. However, the growing complexity of SoCs - i.e., the integration of more and more subsystems with additional processors, etc. - makes it more and more difficult to maintain sufficiently high simulation speeds for SW development on SoC VPs. Therefore, it is necessary to develop methodologies which go beyond TLM and provide a higher abstraction to improve the simulation speed but also provide the necessary detail level, e.g., timing accuracy and system control. Hence, in this paper a new modeling abstraction - referred to as TLM+ - is introduced which enables modeling at higher levels of abstraction on top of TLM. The key idea of the TLM+ abstraction is to preserve the hardware architecture while abstracting the data and control flow by merging hardware and low level driver software at the HW/SW interface. Furthermore, a new resource model concept is introduced which compensates the loss of timing information, induced by the abstracted communication. This enables a fast simulation performance of the overall system while preserving the effects of hardware timing on the SW execution.

The paper is structured as follows. First a discussion of the presented approach with regard to related work is presented, followed by a definition of the TLM+ abstraction. Subsequent to that, the different steps for obtaining the TLM+ abstraction are introduced and its applicability is discussed based on an example system. Following a brief discussion of the experimental results obtained so far, the overall findings are summarized and are complemented by an overview of future steps.

II. RELATED WORK

Transaction Level Modeling is the de-facto standard for creating Virtual Prototypes. Open SystemC Initiative (OSCI) has released two standards for Transaction Level (TL) modeling up to now [1]. These standards define different communication concepts for modeling hardware interfaces. Especially, with the TLM2 standard timing abstraction techniques were introduced to boost simulation performance. These OSCI standards are complementary to the approach introduced here as the abstraction is obtained by block based transactions which can be modeled using the OSCI standards.

The approach presented in [2] connects the QEMU processor emulator to SystemC TL models to enable driver and SW development. Also Several EDA companies provide high speed processor models for VP design, e.g., VaST or CoWare. These models and QEMU are instruction set simulators (ISS). Our approach is to merge the HW/SW interface to enable further communication abstraction. Due to this merge, an ISS can no longer be used. However, we are providing mixed TLM/TLM+ support. Hence, TLM+ abstracted subsystems can be combined with these ISS based approaches.

The SystemQ approach presented in [3] 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.

Several approaches target the development of fast and timed Real-Time Operating System (RTOS) simulation models to increase the simulation speed. In [4] implementation challenges, e.g., task scheduling, are addressed for applying an RTOS to a
system level design language like SystemC. The approach in
[5] provides a generic RTOS model for real-time simulations in
SystemC using manual timing annotations in the source code
for the SW execution. The authors of [6] are dealing with task
scheduling of the RTOS to increase the system performance
of the SW execution. These approaches are increasing the
simulation speed due to native software execution of the RTOS
models. The authors of [7] 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 [8] 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 [9]. Other approaches target automatic timing annotations
of the native SW execution. A compiler based approach is
presented in [10]. In [11] 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 [12].

None of these approaches deal with an abstraction of the
HW models. In contrast to that, our TLM+ abstraction
considers both, the HW and the SW models due to the merge
of the low-level device driver SW and the HW model. However,
since these other approaches deal with areas of optimization
not or only marginally covered by our approach, it might be
possible to combine them to increase the timing accuracy and
system performance of the native SW execution.

III. TLM+ ABSTRACTION

Prior to describing the various new techniques in detail it is
necessary to explain the exact notion of the TLM+ abstraction
and how it relates to other abstraction levels.

![Fig. 1. TLM+ Dataflow Abstraction](image)

Our view of the abstraction chain starts at RTL, which is
a cycle, bit, and architecture accurate model representation
of the HW. At RTL, communication between different blocks
and components is described in terms of signal protocols. The
protocol in turn is described and simulated at bit level. Due to
the high level of detail, simulation, for instance for application
SW validation on an RTL-SoC model, is not feasible without
emulation techniques or deployment of FPGAs. In addition to
that, having to wait for a stable RTL model would delay SW
development to a high degree.

These disadvantages have led to the development of TLM
and hence, VPs. At TLM, signal based protocols are abstracted
by functions calls, clocked synchronization is replaced by
event based synchronization in conjunction with timing anno-
tations, and instead of bit data types abstract integer types are
used. Together, these techniques yield a more abstract way of
modeling a system and hence, a better simulation performance.

Still, a pure TLM representation of a complete SoC is too
expensive with regard to a complete simulation, especially as
the complexity of systems is continuously growing. Therefore,
it is necessary to raise the level of abstraction in VP-design
even further in order to improve the simulation performance
in such a way that full system simulations can be run in a
feasible amount of time. The TLM+ abstraction presented
in this paper shall represent such a continued abstraction on
top of TLM. The key idea behind our approach is to reduce
sequences of TLM-transactions relating to a single HW feature
or task to single transactions with more complex information
structures (i.e., payloads). Hence, TLM+ represents a data flow
abstraction on top of TLM which leads to a reduction of the
overall number of communication activities, leading to faster
simulation runs.

For illustration purposes, Figure 1 depicts the levels of
communication within a system from the application software
down to particular HW blocks. Data going in between has to
pass several layers.

Starting from the application SW which allocates a buffer of
data to be processed by the HW the corresponding request and
the buffer are passed down to the driver of the required device.
The driver in turn translates the request into control values for
the HW and performs sequences of accesses on the HW/SW
interface in order to initiate the HW. The data to be processed
is also tailored into smaller pieces, depending on the burst
capabilities of the underlying bus system. Finally, the data is
also sent to the HW through several bus accesses. Within the
HW however, the data to be processed is again accumulated
to bigger frames or blocks before starting the actual operation.

As the data blocks at the level of application SW are
reconstructed down in the HW, the separation into smaller
units of data within the device drivers and the HW/SW
interface and the reconstruction into buffers can be avoided.
Moving to TLM+ is achieved through a combination of the
following steps:

- Insertion of an interruptible and timing aware host-
simulation wrapper (EMU CPU)
- Introduction of a new HW/SW interface at both driver
  level and peripheral level
- Introduction of a Resource Model for increased timing
  accuracy

The first step represents an abstraction of the CPU model.
The SW instead of being compiled to the instruction set of a
specific core, is executed within a SystemC thread process.
which is part of a SystemC wrapper module (EMUCPU). Therefore, the SW is executed natively on the simulation host platform and is interruptible at the granularity of bus accesses. This step is a precondition for the TLM\(^+\) data abstraction, as introducing a more abstract HW/SW interface requires a merge of the low-level driver SW with the associated peripheral interface. This cannot be accomplished if the SW was cross compiled to instructions. A detailed description of this step is given in [13] and hence, skipped here. The second step represents the actual TLM\(^+\) modeling techniques. It incorporates techniques on how to cut at driver level and incorporate a new HW API based on block transfers, while preserving the original HW-architecture and providing means for a migration path from TLM to TLM\(^+\). TLM\(^+\) block transfers decrease the timing accuracy of the simulation model even more than TLM2 burst transactions because transfer blocks of TLM\(^+\) are allocated by the software and their size is not restricted by HW limitations. The last step represents a new auxiliary unit for ensuring timing accuracy, despite of the higher TLM\(^+\) abstraction. These steps are described in more detail in Sections IV-A and IV-B.

IV. TLM\(^+\) Modeling Concepts

This chapter describes the aforementioned steps towards implementing the TLM\(^+\) abstraction within the following sections.

A. TLM\(^+\) Interface Abstraction

This section describes the interface abstraction concept of TLM\(^+\), which enables block transactions of complete application data buffers. As previously described this interface abstraction targets mainly the HW/SW interface and the device drivers. Therefore, the EMUCPU model is enhanced by two additional interface functions \texttt{read\_bus\_buf} and \texttt{write\_bus\_buf}. In addition to the original \texttt{read\_bus} and \texttt{write\_bus} functions, these two functions also have a \texttt{char* buf} and \texttt{int length} argument. At the SystemC side these functions are mapped to the TLM bus interface. The block transfer is implemented using the same concepts as applied in OSCI TLM2 for implementing burst transactions using a specific payload.

```c
int i;
for(i=0;i<size;++i) {
  #ifdef TLMPLUS
    // Code for TLM+
  #else
    // Code for TLM2
  #endif
}
```

Figure 2 illustrates the block communication concept and maps it directly to Figure 1 which was previously used to introduce the idea of TLM\(^+\). The software application shown in Figure 2 wants to read 16 bytes from a specific hardware module and store the received data in its buffer \texttt{buf}. Therefore, the application software calls the device driver function \texttt{readDev} passing the buffer and the number of available bytes as arguments. Hence, the device driver interface between application and driver communicates using buffers. The device driver contains a macro to specify whether the TLM or the abstracted TLM\(^+\) interface should be used. In case the device driver was compiled for TLM\(^+\), the driver calls the TLM\(^+\) interface function \texttt{read\_bus\_buf} and passes its own arguments - the buffer and number of bytes. The hardware module which receives a TLM\(^+\) read request appends its processed data to the transaction payload which is returned to the application software. In case the device driver was not compiled with TLM\(^+\) features enabled the driver loops over the buffer and transfers each byte as single transaction using the \texttt{read\_bus} function.

Figure 2 shows the data flow of both the TLM\(^+\) block transaction and the TLM transaction. TLM\(^+\) hardware modules implement both the complete register interface to support TLM accesses and the abstract TLM\(^+\) buffer interface. By analyzing the content of the delivered payload, the hardware modules can dynamically detect whether an incoming transaction is TLM or TLM\(^+\).

There are several reasons to provide this runtime feature:

- Proprietary third party IP: It is common that a VP consists of many third party general purpose peripheral and bus master devices which may not be modified. In the latter case it is to be expected that such an IP does not support the TLM\(^+\) methodology yet. Therefore, the bus traffic will consist of regular TLM transactions as well, which still have to be supported.
- Legacy code: Though being a relatively new development, plenty of devices have already been programmed at TLM. Moving all of them to the TLM\(^+\) abstraction at once would not be possible due to short development time windows. Hence, it is necessary to establish a migration path.
- Co-Verification of TLM\(^+\) and TLM IP: Reusing a VP for verification helps to test TLM\(^+\), as well as TLM modules in a more realistic scenario. Since, it can be expected that a TLM\(^+\) VP will incorporate higher layers of SW, integrating an TLM module to it will help revealing more realistic bugs in the hardware module as the environment is not manually crafted as in testbenches.

As Figure 2 indicates, the TLM\(^+\) interface abstraction cuts at the register interface of the HW/SW interface. This has several benefits. The first advantage is that the original software can be reused since the API of the merged device driver is not changed. This is important since software changes more frequently than hardware. Cutting above the drivers would require to change the simulation model every time the software needs to be changed. Another reason is that control modules like the clock generation unit (CGU) can be used unchanged since the SW access to their hardware registers happens through the classical TLM interface. Hence,
the control flow of the VP can be simulated. This is important for the development of the device drivers in order to get notifications if the clock was not activated or power was not enabled before accessing the hardware module. Another advantage is that the TLM$^+$ abstracted modules are fully backward compatible to TLM and can be directly used within a pure TLM VP using an ISS CPU model.

B. TLM$^+$ Timing Modeling

This section explains how timing inaccuracies provoked by the higher level of abstraction can be compensated by introducing a central resource model.

As the atomicity of system transactions is increased when moving to the TLM$^+$ abstraction, timing inaccuracies in the system behavior are introduced, which may lead to data inconsistency. For instance, an initiator with higher priority requests a system resource during an active transfer of another initiator with lower priority. In a TLM model the less prioritized transfer can be interrupted at least at the granularity of word or burst accesses, in order to process the higher prioritized transfer. At the TLM$^+$ abstraction however, transfers are atomic in nature. Since the less prioritized transfers cannot be interrupted in favor of higher priority transfers, the order of transfers taking place within a system simulation at TLM$^+$ may differ when compared to a simulation at TLM. In some cases such a difference leads to data inconsistencies (e.g., reading old values instead of new values that should have been promoted by a higher prioritized transfer). If the system also has to adhere to real time constraints, the different timing may lead to malfunctions at the system IOs. Hence, it is necessary that timing of a TLM$^+$ model is as accurate as possible.

In the following sections we introduce a central Resource Model (RM) which solves the aforementioned problems by applying a priority based arbitration scheme to the simulated transfers of a system.

1) Resource Model (RM) Overview: The RM is a central component of the complete HW/SW system. Each transfer-initiator within the system is assigned a certain priority, which is considered by the RM when a transfer attempts to use a system resource while it is busy. All resources and initiators in a system are registered with this RM. All transfers can be associated with start and end times, yielding the time interval at which a transfer is active. Whenever a transfer attempts to use a system resource it has to request the resource at the RM first. By using the information passed to the RM with a request, the RM can calculate or respectively update the start and end times of transfers. Since TLM$^+$ transfers are non-blocking, processes which initiate transfers need to be suspended, once the transfer has finished. The RM resumes a process once the calculated end time of the corresponding transfer has passed. Furthermore, by processing the requests, the RM can detect and resolve conflicts on resources. Conflicts are mainly resolved by updating the time intervals of active transfers within the RM, and hence, controlling the resumption times of the initiator processes.

A resource is registered with the following properties:

- Resource ID (RSID): set at elaboration time
- Resource Clock Period: can change at runtime
- Transfer unit Read/Write Cycles: can change at runtime

The RSID is used for tracking resource states and as identifier in resource requests. The latter two attributes are used by the RM to calculate the duration of an access to the resource. When a resource is registered with the RM, the RM creates the following list which at runtime stores the IDs of initiators invoking transfers that request the resource.

- Initiator ID
- Access type (read or write)

Any time a resource is requested by an initiator the corresponding initiator ID and the access type is added to this list. This list is used for conflict handling and timing corrections and is explained in detail in the next section.

Initiators are registered at the RM as well. An initiator has the following properties:

- Unique initiator ID (IID)
- Initiator priority
- Start and end time of the active transfer

The IID is used to request arbitrary resources using the specified initiator priority. The transfer start and end times of the initiator are calculated during each resource request.

The payload of a transfer invoked by an initiator contains also the IID along with the package information. A resource request to the RM contains the initiator ID, resource ID, access type and number of transferred bytes. The RM calculates the new start and end time of the transfer, solves resource conflicts and makes timing corrections.

The RM also provides a synchronization function taking the IID of the active transfer as argument. This function suspends the current transfer by waiting for the resume event which is scheduled at the end time of the transfer. In case of resource conflicts with transfers of initiators with higher priorities the resume event of the suspended transfer gets re-scheduled to the new calculated end time. Hence, the suspended transfer is resumed at a later point of time.

Fig. 3. Resource Request Conflict Handling
2) **Resource Model Conflict Handling:** This section describes the conflict handling of the resource model. A conflict can occur if a resource is requested which was already requested for other initiators. Since the resource model knows resource requests of all active transfers it can perform timing corrections through modifying their timing intervals.

Figure 3 illustrates how resource requests are processed within the RM.

The resource identified by RSID is requested for an initiator identified by IID (1). By retrieving the list of active transfers of the requested resource, the RM determines whether the request leads to a conflict. If not, the new end time of the current transfer is calculated (2), the request information is stored within the RM (4), and the processing of the request terminates. If there is a conflict, the priority of the initiator associated with the request is retrieved (3). Following that, it is checked whether there is a request of a higher priority pending for the current resource. If not, it means that the initiator of the current request has the highest priority. Hence, the end time of the current transfer is increased by the requested time interval (5). If the current request has not the highest priority, the end time of the transfer with the highest available priority is retrieved (6) and the end time for the current transfer is recalculated based on this value (7). Once, this is done, it has to be checked whether there are transfers on the resource with a lower priority. If there are none, the request information of the current transfer is stored (4) and processing has finished. If there are transfers with lower priorities, it is necessary to increase their end times by the requested time of the current transfer (9). Furthermore, the resume events of these transfers are rescheduled to their new calculated end times.

3) **Software Timing:** The execution time of the software is handled by the resource model following the same concept introduced in the previous sections. The native software execution can request the EMUCPU resource for the required number of clock cycles. Depending on the calling thread the EMUCPU resource is requested using the corresponding initiator ID. Hence, the resource model performs its priority based conflict handling. The values of the clock cycles are obtained by measuring the SW execution time on the ISS CPU model.

4) **Resource Dependencies:** Some transactions have dependencies on other resources. For example a package transfer to a target module also requires interrupt control unit (ICU) interaction during the complete transaction time. Furthermore, bus accesses to the memory are not modeled because of the native software. Therefore, the resource model provides a mechanism for handling of resource dependencies. These dependencies can be specified with a configuration file. Each line specifies one dependency relation and has the following syntax:

```
INITIATOR SADDR EADDR RS1 RS2 ... RSn
```

**INITIATOR** is the instance name of the initiator, **SADDR** and **EADDR** specifies the address range for this dependency. This is followed by a list of the instance names of resources which should be requested in case of a block transfer of the specified initiator to the specified address range. The resource model provides a dependency resolution function which requires the IID, address, access type, and number of bytes as arguments. The RM requests all resources specified in the dependency relation, performs conflict handling, and calculates the timing information. In the same way, memory accesses of the SW can be considered, though using the EMUCPU.

V. **APPLICATION EXAMPLES**

This section introduces two HW/SW systems to which the TLM+ modeling concepts are applied. For each system the simulation performance and timing accuracy is compared to the system without abstraction. At the end of this section all experimental results are summarized and discussed.

A. **Example Systems**

In our first example two subsystems are connected to one VP as shown in Figure 4. Each of these subsystems incorporates a CPU core which executes the SW. The SW which is executed on the core of the Application VP (AP-VP) controls the system and communicates with the HW Accelerator VP (ACCVP) over a bridge module. The ACCVP responds according to the requests of the APPVP. The ACCVP compresses/expands data packages using an LZW algorithm implemented in software and encrypts/decrypts the packages using an AES encryption accelerator module.

The software which is employed in the following examples makes use of two data paths - the RX path and TX path. In case of the RX path the firmware of the accelerator VP retrieves compressed encrypted data through the serial interface from a testbench element, decrypts it using the AES accelerator module, and expands the compressed data using the LZW algorithm. The application VP then reads the processed data and displays it to the connected LCD output device. Example 1 is modeled in three different abstractions:

1) Complete VP modeled at TLM which incorporates only ISS CPU cores
2) Application VP modeled at TLM which incorporates one ISS; HW Accelerator VP modeled at TLM+ abstraction
3) Complete VP modeled at TLM+ which incorporates only EMUCPU models

Our second example is a VP including a CPU core, system control units, and a camera interface component. The SW which is executed on the CPU core controls the camera interface and retrieves data from it. This is a communication and control centric example and is modeled as pure TLM and pure TLM+.
Column \textit{Runtime} shows the overall simulation runtime of each system. The next column shows the performance gain compared to the not abstracted TLM systems. The column \textit{Timing SW} shows the timing accuracy of the SW part and column \textit{Timing HW} the timing accuracy of the SW part compared to the TLM systems. The values of the timing accuracy are retrieved by analyzing trace files and comparing the time of specific synchronization points (interrupts, resource accesses, etc.) to equivalent points of the TLM VP.

The experimental results in Table I indicate that the TLM$^+$ abstraction brings a huge performance gain for both examples. Furthermore, the resource model provides a timing accuracy of about 95\% for the HW parts.

In case of Example 1 the total HW execution time is only 4\% of the overall execution time (HW+SW time) since the LZW algorithm is implemented as SW. Therefore, the native SW execution brings a huge performance gain. There is also a lot of communication activity (RX/TX paths) where the block transfer of TLM$^+$ brings a further gain to the simulation speed. Hence, the TLM$^+$ model is more than 940 times faster than the TLM model. The overall timing accuracy is only 87.6\% because of 96\% SW activity and manual timing annotations.

In case of Example 2 the total HW execution time measures 48.5\% of the overall execution time. In this HW/SW system data and control communication is dominant. No algorithms are implemented as SW. Hence, mostly the TLM$^+$ interframe abstraction leads to the huge performance gain factor of 391.6. The resource model provides an overall timing accuracy of about 98\%.

The TLM$^+$ abstracted VPs are more than 390/940 times faster than the original VPs. The TLM/TLM$^+$ mixed VP shows how to combine TLM and TLM$^+$ in one simulation model. Furthermore, the resource model has shown to provide a relatively good timing accuracy.

Based on our examples the following correlations can be observed:

- The more data flow characteristics are present in a system, the more performance benefits can be expected when applying the data abstraction techniques introduced.

## VI. Conclusion and Outlook

In this paper we introduced the TLM$^+$ abstraction, a new abstraction level which goes beyond current TLM abstractions and thus, enables faster VP simulation for early SW validation. We achieved the higher abstraction by merging directly at the HW/SW interface - i.e., by grouping word level accesses to higher-order transfers. By introducing a central resource model we ensured that despite the increased level of abstraction it is still possible to maintain a high degree of timing accuracy. Experimental results have also shown that moving to the TLM$^+$ abstraction can yield up to three orders of magnitude faster simulations when compared to regular TLM platforms with ISS-core models.

Currently, this work is being applied to a mobile phone platform, in order to obtain more results and to analyze the feasibility of the approach in an industrial design environment.

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