Analytical Timing Estimation for Temporally Decoupled TLMs Considering Resource Conflicts

Kun Lu, Daniel Müller-Gritschneder and Ulf Schlichtmann
Institute for Electronic Design Automation
Technische Universität München, Munich, Germany

***Abstract*—Transaction level models (TLMs) can use temporal decoupling to increase the simulation speed. However, there is a lack of modeling support to time the temporally decoupled TLMs. In this paper, we propose a timing estimation mechanism for TLMs with temporal decoupling. This mechanism features an analytical model and novel delay formulas. Concepts such as resource usage and availability are used to derive the delay formulas. Based on them, a fast scheduling algorithm resolves resource conflicts and dynamically determines the timing of concurrent transaction sequences. Experiments show that the delay estimation formulas are capable of capturing the timing effects of resource conflicts. At the same time, the overhead of the scheduling algorithm is very low, hence the simulation speed remains high.**

I. Introduction

Virtual prototypes (VPs) based on SystemC have been widely adopted to handle the growing design complexity of today's embedded systems. VPs are often used as effective simulation platforms by designers. In VPs, the HW and SW components of the designed system need to be modeled. To reduce the modeling effort of VPs, transaction level models (TLMs) have been introduced and standardized. In TLMs, transactions are used to model data flows between HW modules. In a transaction, complex signal protocols are abstracted away. Such abstraction greatly improves the simulation speed, making TLMs suitable for fast and early SW development, system verification and design space exploration. For timing accurate TLMs, timed bus-word transactions are used. This means that a transaction transfers data at the granularity of a bus-word, e.g. a byte, a word or a burst of words, etc. Synchronization is performed at each individual transaction by calling the SystemC *wait()* statement. In multiprocessor simulation, such fine-grained synchronization can capture access conflicts at shared HW modules. The conflicts are arbitrated to ensure the correct timing of the conflicting transactions, as illustrated in Fig. 1(a). However, the problem is that calling *wait()* is very expensive, because it causes time-consuming context switches in the SystemC kernel. In SW simulation, the SW may frequently initiate transactions due to cache misses or I/O communication. The simulation speed can be severely slowed down due to the synchronization overhead of transactions. As a remedy, temporal decoupling (TD) is introduced in TLM 2.0 [1] to reduce the synchronization overhead.

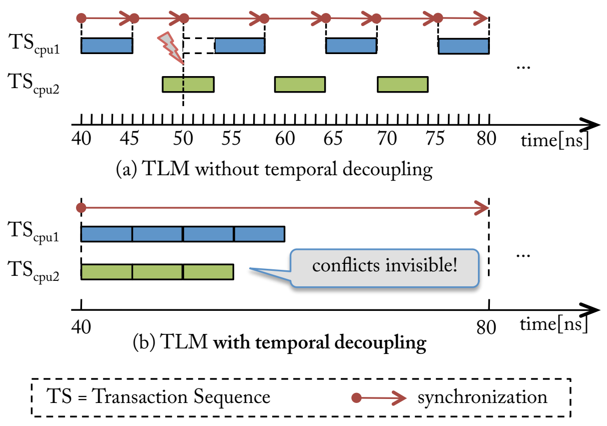

TS_cpu1
TS_cpu2
...
40 45 50 55 60 65 70 75 80 time[ns]
(a) TLM without temporal decoupling
conflicts invisible!
40 80 time[ns]
(b) TLM with temporal decoupling
TS = Transaction Sequence
synchronization

Fig. 1. Introduction of temporal decoupling and its timing problem

A. Temporal decoupling in TLM 2.0

Temporal decoupled TLMs do not synchronize at each bus-word transaction. Instead, one initiator (e.g. a processor) continues its simulation for a long time period. Only then a synchronization request is issued to the SystemC kernel. As shown in Fig. 1(b), only a single synchronization is performed by CPU1 within 40 ns to 80 ns. The problem is that the actual occurrence times of the transactions are lost, therefore access conflicts at shared modules are invisible and the delays due to access conflicts can not be simulated. Conventional arbitration algorithms can not be used in this case, because they work only for individual bus-word transactions. In the TLM 2.0 library, timing accuracy is meant to be sacrificed if temporal decoupling is used. However, this may lead to timing inaccuracy in multiprocessor simulation, in which transactions over the shared bus may be frequently called due to cache misses or I/O communication.

B. Contribution

This paper proposes an analytical timing estimation mechanism to address the timing problem in temporally decoupled TLMs. First, an analytical timing model is introduced, in which each HW module is regarded as a resource. Two concepts, namely *resource usage* and *resource availability*, are used to derive several delay formulas. These formulas model the contention at shared resources. In order to derive realistic delay formulas, we consider several design aspects, such as the arbitration schemes, interrupt handling and advanced bus protocols. Correspondingly, we adjust the calculation of *resource usage* and *resource availability*. Then, we propose a fast scheduling algorithm. This algorithm uses the delay formulas to determine the actual durations of the concurrent synchronization requests. With dynamic event cancellation, only one *wait* statement is needed to schedule one synchronization request. Thus the overhead of the scheduling algorithm is very low. The advantages of our approach are summarized as follows:

- Timing is estimated without arbitrating each transactions.
- The scheduling overhead is very low.
- No global quantum is needed. Each initiator updates its local time variable and synchronizes as it needs.
- It is provided as a library. Existing TLM VPs can be easily ported, by deriving the HW modules from the *resource* class.

Note that our approach does not require any transactions to be stored or simulated. It only needs the timing parameters for applying the formulas. This is advantageous in many cases. For one example, the transactions can be abstracted away [2], which makes transaction arbitration infeasible. For another example, in host-compiled SW simulation, the transactions at cache misses are non-functional. Thus, they do not have to be simulated. Instead, their access time to the resources can be accumulated and used later for timing estimation.

Experiments on a multiprocessor TLM with temporal decoupling show that the scheduling algorithm with the delay formulas successfully estimate the delays due to resource conflicts, while imposing a negligible simulation overhead.

C. Related work

Researchers have explored the idea of faster simulation beyond the abstraction level of TLM [1]–[6]. TLM 2.0 [1] proposes the temporal decoupling technique for TLMs, at the cost of reduced timing accuracy because of the absence of conflict handling. Ecker et al. [2] abstract a long sequence of transactions caused by the transfer of a large data block into a single block transaction. To extract the timing for those highly abstracted block transactions, Lu et al. [7] profile the corresponding driver functions. But they do not consider timing estimation when resource conflicts are present, such as in the multi-processor simulation. Schirner et al. [3] propose a concept of result oriented modeling. A conflict-free optimistic duration is firstly used for a long transaction sequence. Then retroactive timing correction is performed successively, until the actual duration is reached. Similar to this concept, Stattelmann et al. [4] also perform retroactive timing correction, together with quantum allocation. The occurrence times of all transactions are stored in lists. Then the lists are traversed to retroactively correct the timing by arbitrating individual transactions. However, the occurrence times of the bus-word transactions might not be known, e.g. when block transactions are used [2]. Further, [3] and [4] arbitrate each transaction, it may become complex and expensive when the transaction lists are large due to frequent cache misses or I/O accesses. Besides, since the computation and communication models in TLMs often do not provide cycle accuracy, one may argue that it is not always advantageous to perform cycle accurate arbitration in a timing inaccurate system. Sonntag et al. [5] propose SystemQ as a SystemC library for performance estimation. SystemQ is more abstract than TLM in terms of HW modeling and beyond the functional view in terms of SW simulation. It is initially supposed to be used at an early design phase to aid the decision making of message passing architectures. To simulate TLMs with SystemQ, adaptors are needed to collect a group of transactions into a message, which is queued at a server and sent as a whole. SystemQ may offer very rough performance estimation due to its high abstraction of HW and SW modeling. Bobrek et al. [6] train a statistical regression model with samples of resource contention. They employ their approach to simulate software with annotated performance information of the target processor. The training effort may be high to accurately model all the contention scenarios. Besides, the modeling support for applying to TLMs is not investigated in their approach. A more recent approach [8] also investigates the concept for predictive timing estimation. It is used in the scheduling of periodic real-time tasks in host-compiled OS modeling. This approach needs to know the periods of the tasks, so that the preemption time and the duration of the tasks can be predicted. Our approach does not require the resource accesses to be periodic, nor does it require the exact access times of the resources. It extracts several timing parameters and employs analytical delay formulas and a fast scheduler for timing estimation.

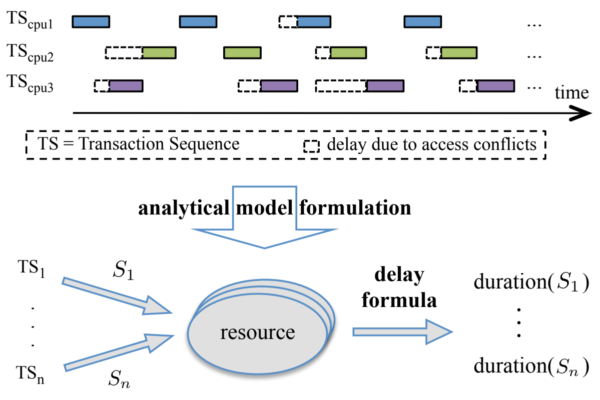


Fig. 2. Problem formulation and the analytical model.

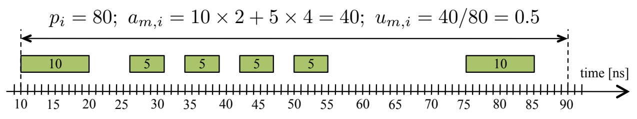
In the following, Sec. II presents our approach. Sec. III shows experimental evaluation. Sec. IV concludes this paper.

II. Analytical Timing Estimation

In the following, we first introduce the analytical model. Then, we present the delay formulas and the scheduling algorithm which performs dynamic timing estimation.

A. Problem formulation

With temporal decoupling, an initiator simulates ahead and updates its local time. After a long period of time, it issues a synchronization request to the SystemC kernel. Unlike a SystemC `wait()` call, such a synchronization request involves not only a time period, but also many transactions during this period for accessing the memory or other peripherals. In



$p_i = 80;\ a_{m,i} = 10 \times 2 + 5 \times 4 = 40;\ u_{m,i} = 40/80 = 0.5$
10
5
5
5
5
10
time [ns]
10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90

Fig. 3. Example of calculating the resource usage.

some case, the transactions within this period are abstracted away [2], thus their occurrence times are not available for arbitration. Temporally overlapping synchronization requests of multiple initiators can have conflicting accesses to shared HW modules. Thus, their actual durations could be delayed. Now, the problem to solve is how to determine those delays without expensively arbitrating individual transactions or when transaction-based arbitration is not feasible.

We propose an analytical model to tackle this problem, as shown in Fig. 2. Each HW module is regarded as a resource, denoted by R. We use S to denote the synchronization request of a long time period including many transactions. For S_i of initiator i, its access time of resource R_m is written as $a_{m,i}$. Thus each synchronization request has following composition:

$$S_i = (p_i,\ a_{1,i},\ a_{2,i},\ a_{m,i}, ...),$$

where p_i is the requested period of S_i, $a_{1,i}$ is the sum of S_i's access time within p_i at resource R_1, etc. For example, assume all the transactions of S_i in Fig. 3 access resource R_m, then $p_i = 80ns$ and $a_{m,i} = 40ns$.

Given a set of overlapping synchronization requests $\mathbf{S}$, we estimate the actual duration ${p_i}'$ of S_i, as shown in

$${p_i}' = p_i + D(i, \mathbf{S}).$$

Here $D(i, \mathbf{S})$ is an estimation function that models the delay due to resource conflicts. This function uses several novel delay formulas, derived in the following.

B. Delay formulas

1) Resource usage and availability: We define two intuitive terms before deriving the delay formulas: resource *usage* and resource *availability*. Resource usage measures the degree of an initiator's access to a resource. Resource availability measures how much a resource is available to an initiator. Formally, for a synchronization request S_i that uses a shared resource R_m, we have:

- $u_{m,i} \in [0, 1]$: **usage** of R_m demanded by S_i, given as:

$$u_{m,i} = \frac{a_{m,i}}{p_i}$$

- $w_{m,i} \in [0, 1]$: **availability** of R_m for S_i.

For example in Fig 3, for the given duration and access time of S_i at R_m, we have $u_{m,i} = 0.5$.

2) Deriving the delay formulas: Because the resource is not fully available, S_i's access time $a_{m,i}$ of R_m is prolonged to $\frac{1}{w_{m,i}} \cdot a_{m,i}$. Thus its delay $d_{m,i}$ at R_m is given as:

$$\begin{aligned} d_{m,i} &= \frac{1}{w_{m,i}} \cdot a_{m,i} - a_{m,i} = \frac{1 - w_{m,i}}{w_{m,i}} \cdot a_{m,i} \\ &= \frac{1 - w_{m,i}}{w_{m,i}} \cdot u_{m,i} \cdot p_i \end{aligned} \qquad (1)$$

The overall delay for S_i is

$$d_i = \textstyle\sum_{m=1}^{r} d_{m,i},$$

with r being the number of shared resources. For example, for a synchronization request S_i at resource R_m, let $a_{m,i} = 0.3ms, p_i = 1ms$, and $w_{m,i} = 0.6$, then it follows $d_{m,i} = \frac{0.4}{0.6} \cdot 0.3ms = 0.2ms$. Assume R_m is the only shared resource, then the duration of S_i will be prolonged to ${p_i}' = p_i + d_i = 1ms + 0.2ms = 1.2ms$. The interpretation of this delay formula complies with the observation in practice: an initiator will experience more delay, if it has less resource availability or it demands more resource usage. More specifically, the delay of an initiator is linear to its resource usage and hyperbolic to its resource availability. This delay formula is simple and yet efficient in terms of complexity and accuracy, as will be demonstrated in the experiments.

The delay estimation formula in (1) requires to know the resource usage and availability. In the following, we show how to calculate these two terms considering several aspects, such as arbitration policy, interrupt handling and bus protocols.

B-1 Consider the arbitration policy

Arbitration policy with preemptive fixed priorities: Let S_1 and S_2 be two synchronization requests issued by initiator 1 and 2 respectively. Assume initiator 1 has higher priority than initiator 2, then the resource availability at R_m for S_1, S_2 are calculated as:

$$w_{m,1} = 1; \qquad w_{m,2} = 1 - u_{m,1}. \qquad (2)$$

Substitute $w_{m,2}$ in (1), we have

$$d_{m,2} = \tfrac{u_{m,1}}{1 - u_{m,1}} \cdot u_{m,2} \cdot p_2,$$

Thus if $u_{m,1}$ increases, delay $d_{m,2}$ will also increase.

Now suppose a third initiator with lower priority issues a synchronization request S_3. To calculate its delay, we compute the combined resource usage $u_{m,(1,2)}$ of R_m demanded by initiator 1 and 2: $u_{m,(1,2)} = {u_{m,1}}' + {u_{m,2}}'$, where ${u_{m,1}}'$ remains as before and ${u_{m,2}}' = \frac{a_{m,2}}{{p_2}'}$. Here we show that $u_{m,(1,2)} \in [0, 1]$:

$$\begin{aligned} 0 < u_{m,(1,2)} &= {u_{m,1}}' + {u_{m,2}}' = u_{m,1} + \frac{a_{m,2}}{{p_2}'} \\ &= u_{m,1} + \frac{u_{m,2} \cdot p_2}{p_2 + d_2} \\ &= u_{m,1} + \frac{u_{m,2} \cdot p_2}{p_2 + \frac{u_{m,1}}{1 - u_{m,1}} \cdot u_{m,2} \cdot p_2} \\ &= u_{m,1} + \frac{1}{\frac{1}{u_{m,2}} + \frac{u_{m,1}}{1 - u_{m,1}}} \\ &\leq u_{m,1} + \frac{1}{1 + \frac{u_{m,1}}{1 - u_{m,1}}} = 1 \end{aligned} \qquad (3)$$

Thus the resource availability for S_3 is $1 - u_{m,(1,2)}$, which can be then used in (1) to compute the delay of S_3. Similar calculation holds for additional initiator with even lower priority.

Arbitration policy with round robin priority scheme: Assume initiator 1 and 2 have the same priority, then the resource availability at R_m for S_1 and S_2 can be expressed symmetrically as:

$$w_{m,1} = (1 - u_{m,2}) + \frac{u_{m,1}}{u_{m,1} + u_{m,2}} \cdot u_{m,2}; \quad w_{m,2} = (1 - u_{m,1}) + \frac{u_{m,2}}{u_{m,2} + u_{m,1}} \cdot u_{m,1}. \tag{4}$$

So, assume $u_{m,1} = u_{m,2} = 0.7$, we have $w_{m,1} = 1 - 0.7 + 0.5 \cdot 0.7 = 0.65$. Correspondingly, the delays and actual durations of S_1 and S_2 can be calculated. If a third initiator with the same priority is added, $w_{m,1}$ is computed similarly as above:

$$w_{m,1} = 1 - u_{m,(2,3)} + \frac{u_{m,1}}{u_{m,1} + u_{m,(2,3)}} \cdot u_{m,(2,3)},$$

where $u_{m,(2,3)} = min(u_{m,2} + u_{m,3}, 1)$ is the combined resource usage of S_2 and S_3.

B-2 Consider interrupt service routines

Some interrupt service routines (ISR) use polling policy, meaning that they keep invoking transactions to check the status register of a HW module until the ready state is set. In this case, the resource usage should be adjusted accordingly. For example, assume the 2nd to 5th transactions in Fig. 3 are due to register polling. Also, assume the HW module being polled will set its status register at $60ns$. Then even if these transactions are delayed by the transactions of other higher priority initiators, the polling in the ISR will still end at $60ns$. Correspondingly, when calculating the delay of a lower priority initiator, the register polling related resource accesses within its ISR do not count as its resource usage.

B-3 Consider the bus protocols

In most common cases, the system bus is a shared resource. When advanced bus protocols are used, the calculation of resource usage for delay estimation should also be adjusted. For example, according to AMBA AHB bus protocols, slow and split-capable slaves may split the transactions and free the bus. Therefore, we do not consider the split slot when calculating the bus usage of higher priority initiators. This gives better delay estimation for lower priority initiators. Further, an initiator can lock the bus and thus can not be preempted after it has been granted to use the bus. Also, address and data phases are pipelined in AHB protocols. Preemption of lower priority initiator happens when the data is ready for the current beat of its burst transaction. To handle these cases, the resource availability of higher priority initiator in (2) is adjusted to $w_{m,1} = 1 - \alpha$, where α is the bus usage of the lower priority's bus-locking transactions plus the average percentage of a beat in a burst transaction. Correspondingly, the resource availability for the lower priority initiator is increased by α.

C. *The scheduling algorithm*

A scheduling algorithm in Alg. 1 is called each time an initiator issues a new synchronization request. This algorithm first updates all ongoing synchronization requests, then determines their durations by employing the delay formulas. Notably, the event cancellation dynamically re-schedules the end of each synchronization request, based on the updated

Algorithm 1 The Scheduling Algorithm

```
$S_{new}$ := the newly received synchronization request
// update ...
for $S_i \in$ S do
    update remaining duration of $S_i$
    update remaining resource access time of $S_i$
end for
// resolve timing ...
for $S_i \in$ S do
    for $R_x \in$ R do
        A := A $\cup$ getAvailability($R_x, S_i$)
    end for
    p := remaining duration of $S_i$
    d := getDelay($S_i$,A)        // Equ. 1
    p := p + d
    $S_i$.endEvent.cancel()       // dynamic re-scheduling
    $S_i$.endEvent.notify(p)
end for
wait($S_{new}$.endEvent)          // the single call to wait()
```

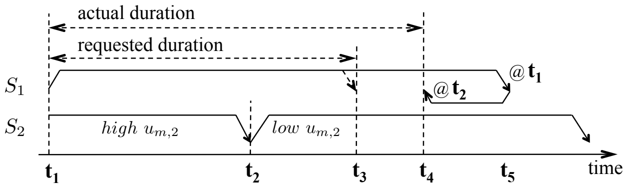


Fig. 4. Dynamic scheduling example.

resource availability. Thus, only one *wait* statement is needed per request, no matter how many re-scheduling is performed. To illustrate the dynamics, consider the example in Fig. 4. We assume the higher priority S_2 has high resource usage before t_2 and low resource usage from t_2 onward. At t_1, S_1 of initiator 1 requests a duration until t_3. The scheduling algorithm is called and schedules S_1 to finish at t_5. At t_2, S_2 finishes and resumes with the next synchronization request. The scheduling algorithm is called again. It updates the remaining duration and the resource access time of S_1. Then it calculates the new delay of S_1. Since now S_2 has lower resource usage, S_1 is delayed less and is re-scheduled earlier to t_4. Finally, S_1 finishes at t_4 and initiator 1 can resume.

D. *Modeling support - integrating the resource model*

The scheduling algorithm is implemented based on a central resource model [2]. It is provided as a library and can be viewed as an additional timing layer to the original VP. To use it, each HW module needs to derive from a resource class:

```
class ModuleX: public sc_module, public rm_resource
 ...
 void thread_1{
   ...
   functionA();
   //synchronize after functionA
   rm_useResource(bus_id, sumBusAccessTime);
   rm_useResource(mem_id, sumMemAccessTime);
   ...
   rm_sync(requestedDuration);
   ...
```

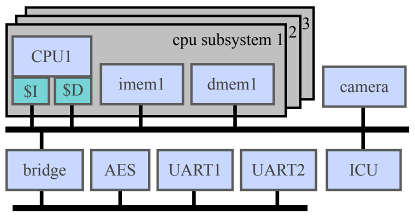


Fig. 5. VP architecture modeled with TLM.

At instantiation, a resource registers itself, receives a resource id, and sets its priority if needed. An initiator executes its thread and accumulates its local time and the access time of the resources. Then it issues a synchronization request to the resource model with a requested duration and the resource access times. The scheduling algorithm is then called to determine the actual duration. When to issue the synchronization request can be determined by the programmer (e.g. after a function is called) or by checking whether the accumulated local time exceeds a threshold.

III. Experimental Results

In the experiments, we firstly perform RTL simulation as the proof of concept. Then we apply the proposed approach to application SW simulation on a multiprocessor TLM. Timing accuracy and simulation speed-up are examined to evaluate the proposed synchronization mechanism. The architecture of the employed TLM VP is sketched in Fig. 5.

A. Proof of concept with RTL simulation

RTL models are clocked, thus temporal decoupling is usually not used. Nevertheless, here we use cycle accurate RTL simulation to validate the proposed analytical formulas for delay estimation. Two initiators are connected to an AMBA AHB bus at a clock period of $10ns$. They use a random traffic generator to transfer data over the bus. Preemptive arbitration is used, with initiator 1 having a lower priority. With no resource conflicts, the number of transactions sent by initiator 1 within $1ms$ is fixed for a given traffic density. We measure the delay to finish those transactions for initiator 1 under various traffic scenarios.

As shown in Fig. 6(a), for a fixed bus usage of initiator 2, the delay is approximately linear to the bus usage of initiator 1. As shown in Fig. 6(b), for a fixed bus usage of initiator 1, the delay is approximately hyperbolic to the bus usage of initiator 2. The linear and hyperbolic curvatures conform to the formula in (1) . These curvatures can be very well estimated using (1) and (2) with adjustment discussed in Section II-B-3.

B. Application SW simulation

Now we apply the proposed approach to SW performance simulation with annotated source code [9], [10]. The source code is annotated with performance information with respect to the target processor, such as estimated execution cycles and cache accesses. Without temporal decoupling, the simulation proceeds in a way as introduced in Fig. 1. The expensive *wait* statement is called to synchronize the estimated cycles before cache line refilling or accesses to non-cachable peripheral registers. Then transactions are evoked for data transfer over the bus. In this way, the occurrence times of the transactions

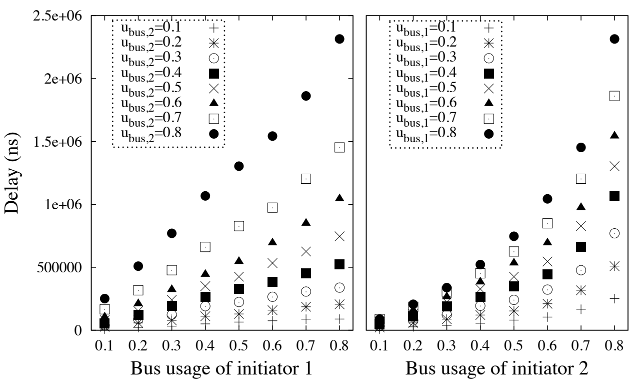


Fig. 6. Delay of the synchronization request of cpu 1 with respect to different resource usage scenarios

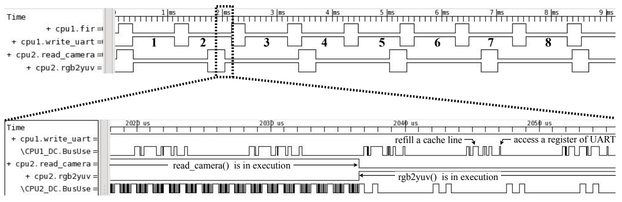


Fig. 7. Traced functions and HW accesses in TLM simulation.

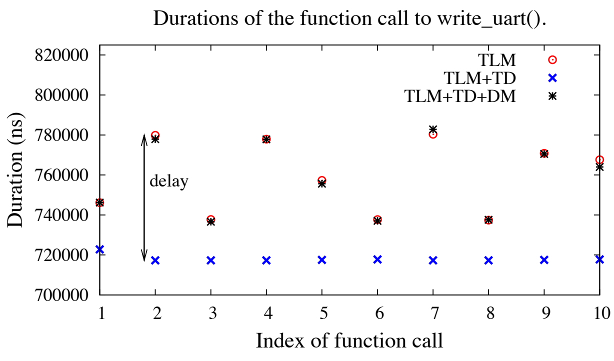


Fig. 8. Timing comparison for *write_uart()*.

are correctly simulated. However, the simulation speed may be severely reduced in the case of frequent cache misses or peripheral accesses. In the following we show that it is necessary to use temporal decoupling to keep high simulation speed and that our approach ensures high timing accuracy when temporal decoupling is used.

The experiment is set up as such: cpu 1 filters a new buffer with the *fir* algorithm, and then writes the results to the *UART* module, so on and so forth. Concurrently, cpu 2 reads a frame from the camera, performs color conversion *rgb2yuv* on each pixel, and then continues with the next frame. A fixed priority scheme is used, with cpu 2 having higher priority than cpu 1. The simulation is performed in 3 modes. In TLM simulation, temporal decoupling is not used. In TLM+TD simulation, temporal decoupling is used. But there is no consideration of the delay due to the conflicts at shared resources. In TLM+TD+DM simulation, temporal decoupling is used. The proposed delay model (DM) and scheduling algorithm are applied for timing estimation. In the following, we first give the overall results and then the in-depth analysis.

The overall simulation results are in Tab. I. In TLM+TD simulation, we can see that the simulation is 19 times faster if temporal decoupling is used. However, the timing is underestimated by 7%, because the transactions of lower priority cpus are not delayed by the resource conflicts. In TLM+TD+DM simulation, the low timing error (-0.8%) indicates that the proposed delay formulas and scheduling algorithm effectively resolve resource conflicts and provide good timing estimation. At the same time, the simulation time is close to that of TLM+TD simulation. This implies that a very small overhead is caused by the scheduling algorithm.

TABLE I
MULTIPROCESSOR SIMULATION RESULTS.

Sim. mode	**TLM**	**TLM+TD**	**TLM+TD+DM**
Cycles	306M	288M	303M
Err(%)	-	-7	-0.8
Exe. time (s)	8.81	0.47	0.48
Speed-up.	-	19	18.4

For in-depth analysis, the concurrent bus accesses of cpu 2 and cpu 1 in TLM simulation are given in Fig. 7. We can see that when executing *read_camera*, cpu 2 evokes transactions more often than executing *rgb2yuv*. Therefore, transactions of cpu 1 are delayed more when cpu 2 is executing *read_camera*. This can be seen by the measured durations of $cpu1$'s 2nd, 4th and 7th calls to *write_uart* in Fig. 8. Further we can also see that the estimated delays in TLM+TD+DM simulation match very well with those in TLM simulation. This means the proposed delay formulas and the scheduling algorithm have successfully modeled the timing related to resource conflicts and determined the dynamic timing of the temporally decoupled synchronization requests.

C. Simulate a hard case

To stress the proposed approach, the data cache is disabled and polling is used in the ISR of I/O driver functions, in order to create heavy traffic and thus access conflicts at the shared bus. The experiment is set up as such: cpu 1 keeps writing new buffers of various lengths to the UART1 module; cpu 2 keeps reading a buffer from the AES module; cpu 3 keeps writing new buffers of variant lengths to the UART2 module. A round robin arbitration scheme is used. For cpu 3, we measure the duration of its call to the *write_uart()* function. The results are given in Fig. 9. As can be seen, timing in TLM+TD simulation is quite underestimated, with the errors around 14%. In contrast, timing in TLM+TD+DM simulation is very well estimated, with the errors fluctuating around 0%. The average absolute error for each call is around 2%. Further it needs to be pointed out that a relatively large fluctuation of timing errors may occur for synchronization requests of relatively short periods, within which the delays are sensitive to the actual occurrence times of the transactions. But the overall timing is well estimated over a long time period. In Fig. 9, the overall timing error for the accumulated duration of those calls is below 1%.

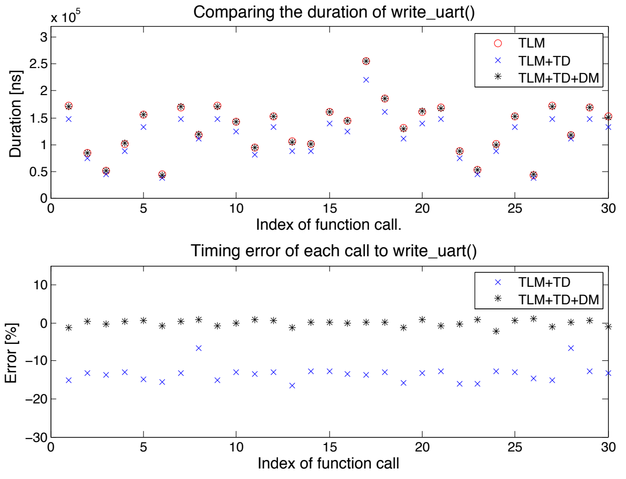


Fig. 9. Durations of cpu 3's calls to write_uart() under round robin arbitration scheme.

IV. CONCLUSIONS AND FUTURE WORK

We have shown an approach that analytically determines the timing for temporally decoupled TLMs. The proposed delay estimation formulas together with a scheduling algorithm achieve high timing accuracy in the experiments while causing little simulation overhead. In the future, we will investigate other communication patterns and interconnect models, such as AMBA AXI or network on chip models. If necessary, we can derive new formulas, for which parameter fitting might be performed.

ACKNOWLEDGEMENTS

This work is party sponsored by the German Federal Ministry of Science and Education (BMBF) in the project SANITAS (16 M 3088). Additionally, the authors thank Michael Velten, Volkan Esen, and Wolfgang Ecker for providing a preliminary version of the resource model.

REFERENCES

[1] OSCI, *OSCI TLM-2.0 Language Reference Manual*, 2009.
[2] W. Ecker, V. Esen, and M. Velten, "TLM+ modeling of embedded HW/SW systems," in *Design, Automation and Test in Europe (DATE)*, 2010.
[3] G. Schirner and R. Doemer, "Fast and Accurate Transaction Level Models using Result Oriented Modeling," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006.
[4] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and Accurate Resource Conflict Simulation for Performance Analysis of Multi-Core Systems," in *Design, Automation and Test in Europe (DATE)*, 2011.
[5] S. Sonntag, M. Gries, and C. Sauer, "SystemQ: A Queuing-Based Approach to Architecture Performance Evaluation with SystemC," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2005.
[6] A. Bobrek, J. M. Paul, and D. E. Thomas, "Shared resource access attributes for high-level contention models," in *ACM/IEEE Design Automation Conference (DAC)*, 2007.
[7] K. Lu, D. Mueller-Gritschneder, and U. Schlichtmann, "Accurately Timed Transaction Level Models for Virtual Prototyping at High Abstraction Level," in *Design, Automation and Test in Europe (DATE)*, Mar. 2012.
[8] P. Razaghi and A. Gerstlauer, "Predictive OS Modeling for Host-Compiled Simulation of Periodic Real-Time Task Sets," *IEEE Embedded System Letters (ESL)*, 2012.
[9] P. Gerin, M. M. Hamayun, and F. Petrot, "Native MPSoC co-simulation environment for software performance estimation," in *International conference on Hardware/Software codesign and system synthesis*, 2009.
[10] K. Lu, D. Mueller-Gritschneder, and U. Schlichtmann, "Memory Access Reconstruction Based on Memory Allocation Mechanism for Source-Level Simulation of Embedded Software," in *Asia and South Pacific Design Automation Conference (ASP-DACA)* , 2013.