Automated Construction of a Cycle-Approximate Transaction Level Model of a Memory Controller

Vladimir Todorov†, Daniel Mueller-Gritschneder‡, Helmut Reinig†, Ulf Schlichtmann‡
† Intel Mobile Communications GmbH ‡ Technische Universität München
vladimir.todorov@intel.com helmut.reinig@intel.com daniel.mueller@tum.de ulf.schlichtmann@tum.de

Abstract—**Transaction level (TL) models are key to early design exploration, performance estimation and virtual prototyping. Their speed and accuracy enable early and rapid System-on-Chip (SoC) design evaluation and software development. Most devices have only register transfer level (RTL) models that are too complex for SoC simulation. Abstracting these models to TL ones, however, is a challenging task, especially when the RTL description is too obscure or not accessible. This work presents a methodology for automatically creating a TL model of an RTL memory controller component. The device is treated as a black box and a multitude of simulations is used to obtain results, showing its timing behavior. The results are classified into conditional probability distributions, which are reused within a TL model to approximate the RTL timing behavior. The presented method is very fast and highly accurate. The resulting TL model executes approximately 1200 times faster, with a maximum measured average timing offset error of 7.66%.**

I. Introduction

Modern system-on-chip (SoC) designs are very complex and, thus, very hard to simulate and verify. The conventional register transfer level (RTL) modeling is of too fine granularity to allow whole system designs to be rapidly simulated or used as virtual prototypes. Therefore, another more abstract level of modeling, namely electronic system level (ESL) [1], has to be applied.

ESL concentrates on abstract models of hardware components, which manage to keep the same or approximately the same timing accuracy as an RTL one, but execute much faster than their RTL counterparts. The ESL models, however, may not be synthesizable. The important factor is the correct representation of the original's device behavior. These models provide a fast and versatile way of design exploration and performance estimation. Therefore, a multitude of design decisions can be evaluated in a short amount of time.

To further reduce the simulation effort, the abstract models use a different communication paradigm. Instead of signals (as in RTL), which are directly mapped onto physical wires, they exchange messages, called transactions [2], [3]. Thus, ESL uses transaction level modeling (TLM) [4]. Each transaction occupies simulation time and represents the whole communication intent as one entity. This removes the need for signals and reduces the number of breakpoints in the simulation by making the models sensitive to a much smaller amount of events.

SoC designs contain significant numbers of devices, some of which may only have RTL models. Hence, transaction level (TL) models have to be created in order to proceed with ESL simulations. However, the task might be a challenging one, as some devices may be an Intellectual Property (IP), provided by external vendors and, thus, no access to RTL description is granted or their RTL description is too complex and obscure and manual remodeling might consume too much time.

978-3-9810801-8-6/DATE12/©2012 EDAA

This work tackles the problem of rapidly creating an abstract, cycle-approximate TL model of an RTL IP block, describing a memory controller, to be used within an industrial virtual prototype on ESL. A memory controller is a device responsible for managing memory media such as DRAM and/or any non-volatile medium and the flow of information to and from it. It takes care of serving the memory as required by its protocol (refreshing, opening/closing of banks and rows) and provides a general interface to the rest of the system for storing and loading data.

The complexity and obscurity of the memory controller's RTL description do not allow for abstracting the model by hand. The functional part is straightforward - the device just passes information between the system and the memory. However, the timing exhibited by the device remains hidden within the RTL logic. To solve the problem, an automatic, fast, and accurate method is devised. It extracts the timing behavior from the RTL model by means of simulations, organizes the results into conditional probability distributions (cycle-approximate timing model) and provides the distributions in a usable form to an abstract generic TL model (functional model), written in SystemC. The TL model uses the conditional probability distributions to recreate the delay and applies it on transactions. This methodology allows for a rapid construction of accurate, low complexity TL models.

The main contribution of this work is a novel methodology for automatic and rapid generation of a cycle-approximate model of a memory controller, where:

- No access to RTL description is required
- The resulting TL model has very high execution speed
- The timing accuracy of the resulting TL model is high (low timing error)

The rest of the paper is structured as follows. Section II presents the related work on abstracting away RTL models and co-simulating such with TL ones. In Section III the core of this work, the modeling methodology, is presented. Section IV provides the obtained results about the accuracy and speed of the approach. Finally, Section V presents conclusions and an outlook.

II. Related Work

Several other approaches for abstracting or translating RTL descriptions exist. The authors in [5] present an automatic way for construction of transactors, i.e., abstract adapter modules that translate the transactions to RTL signals and vice-versa, allowing for co-simulation between different levels of granularity. However, transactors decrease the simulation performance as the RTL implementation remains the bottleneck. There exist several off-the-shelf tools like Verilator [6], V2SC [7] and a tool called V2X presented in [8] that deal with the translation of hardware description languages like VHDL and Verilog to SystemC, which can be adapted to work with transactions. However, this approach translates the code with minor optimizations performed on the structure and there is almost no performance gain. The tools also require access to the RTL structure itself, which may not always be available. Another approach for automatically converting RTL descriptions to TL equivalents is presented in [9], [10]. This approach uses extended finite state machines [11] to create a formal description of a module. The formal description serves as a template for the creation of the TL model with the same functionality. This approach, however, removes timing information and produces untimed models. In comparison, the approach presented in this paper provides an RTL description independent, fast and accurate method for constructing timed TL models.

III. Modeling Methodology

To solve the problem, the RTL description of the memory controller is treated as a black box module with sets of inputs and outputs (Fig. 1). The exact internal behavior of the device remains hidden, but the activity on its periphery is observable in RTL simulations. The intrinsic timing of the black box can be obtained by performing measurements on the results of these simulations.

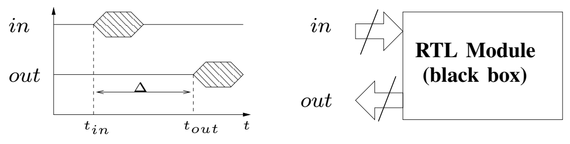

Fig. 1. Black box module

The set of inputs is comprised of all the signals that carry information from the system to the memory controller and the set of outputs contains all the signals that do the opposite. The variable of interest is the delay caused by the RTL module, $\Delta = t_{out} - t_{in}$, or how much time is needed for a given input to produce some output. For example, Δ can be the time between the arrival of a read request (t_{in}) and the appearance of the first data beat (t_{out}) or the time between an arrival of a write access (t_{in}) and the end of its last data beat (t_{out}).

By applying constrained random accesses (distributed over the range of meaningful inputs), the behavior of Δ for many different requests to the RTL component is extracted. The results are then used to construct a statistical profile for Δ, which is reused within a functional model on TL. The functional model just forwards transactions between the memory and the rest of the system.

A. Statistical Timing Model

During the RTL simulation of the memory controller, each provided request x_i (read or write) triggers a response y_i of some kind. The delay Δ_i marks the difference between the arrival time of the request (t_{x_i}) and the one of the response (t_{y_i}). The goal is to gather the delays for the different request and then reuse them within the TL model. However, the delay Δ_i resulting from x_i is not stationary and has more than one realization (Fig. 2). Due to dynamical processes occurring within the controller, providing the same input x_i at two different time points t_1 and t_2 may result in $\Delta_i^{t_1} \neq \Delta_i^{t_2}$. Thus, an access x_i may produce different Δ_i, depending on circumstances, such as the internal state of the controller. By producing a large number of observations, a profile of Δ_i in the form of histogram is constructed, yielding a statistical distribution of the variable.

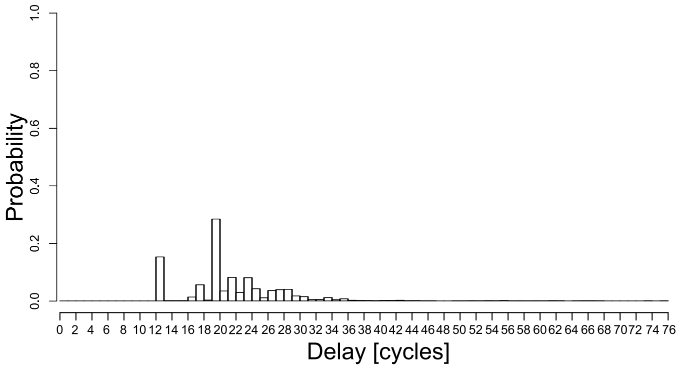

Fig. 2. Delay distribution for read accesses

To minimize uncertainty and, respectively, the variations of the resulting distributions, dependencies are included. They enable the splitting of a single statistical model of the delay into several smaller ones, represented in the form of conditional probabilities.

The results of the RTL simulations of the memory controller are formatted as waveforms. Waveforms predominantly have illustrative purposes, as they provide a convenient, but cumbersome way for exploring the activity of a particular circuit (Fig. 3).

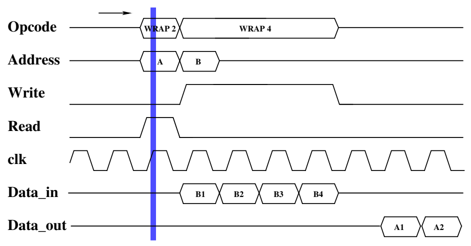

Fig. 3. Waveform scanning

To speed up the process, waveform analysis is done by software. It slides a virtual marker through the simulation results, while observing the values on the wires $\mathcal{W}$ and in the registers $\mathcal{R}$, representing the input/output of the memory controller. As the marker is slid, the pattern ($||\mathcal{W}||$ and $||\mathcal{R}||$) formed by the members of $\mathcal{W}$ and $\mathcal{R}$ changes. The algorithm is informed about the communication protocol and uses $\mathcal{W}$ and $\mathcal{R}$ to identify the beginning of an access and its corresponding response. The identified accesses, together with the responses are encapsulated into data structures. Each instance of the data structure A_i saves all the properties of a particular memory access x_i. Additionally, it has two timestamps. One is the t_{x_i} (start time), indicating the arrival time of the access and the other one is the t_{y_i} (end time), indicating the arrival time of the response. This allows the structure to represent also the delay Δ_i. All extracted accesses are sorted by start times into a queue Q_{access} .

A set of criteria $\mathcal{C}$ is devised according to the specifics of a general memory controller in order to embrace factors that affect the delay. The criteria are used as dependencies in the construction of the conditional probability functions. The set $\mathcal{C}$ used for the memory controller is the following:

- *Type* - Differentiates types of accesses($Read$ or $Write$).
- *Opcode* - Differentiates accesses' lengths in terms of data words. Shorter requests are expected to take shorter time.
- *Predecessor* - Sorts accesses by their predecessors. For example, a read access preceded by a write access is different than a read access preceded by a read access. The possible outcomes grow with number of predecessors considered (taps) by a factor of two (2^{taps}, with $taps \in [1, \infty)$).
- *Successor* - Differentiates accesses according to the type of access that follows the one being classified.
- *Bank* - Differentiates whether an access exhibits a bank change or not. A bank is a portion of the memory, which is specified in the address field of an access. Changing banks is expected to impose higher delay.
- *Row* - Similar to the *Bank* criterion, but tracks changes on sub-portions of a bank, called rows.
- *Distance* - Sorts out accesses according to the distance between them and their immediate predecessors. The distance d is measured in cycles and $d \in [1, M]$, where $M, d \in \mathbb{N}$. M is the maximum considered distance and if $d > M$ d is set to M.
- *Status* - Decides if the controller is busy or not when the current request arrives. Higher delay is expected when the controller is busy.
- *Overlap* - Works in the opposite direction. A larger delay for the current access in service is expected, if the controller has to accept another request meanwhile.

By making use of $\mathcal{C}$ a classification scheme is constructed. The criteria in $\mathcal{C}$ or a subset of them are applied one after the other to construct it. Each criterion $C \in \mathcal{C}$ is a filter with multiple outcomes, the combination of several criteria results in a tree structure $T_{class} = (\mathcal{N}, \mathcal{K}, \rho)$, described by its inner nodes $\mathcal{N}$, leafs $\mathcal{K}$ and the function $\rho : \mathcal{K} \to \mathcal{N}^+$, giving the path from the root to each leaf. Each node $N \in \mathcal{N}$ designates a criterion and each leaf is associated with a delay distribution $D \in \mathcal{D}$ via the bijective mapping $\nu : \mathcal{D} \to \mathcal{K}$. $\mathcal{D}$ is the set of resulting conditional distributions. The accesses from Q_{access} traverse the tree and are classified to the respective leafs.

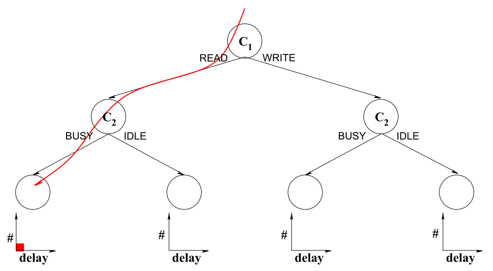

Fig. 4. Tree traversal

For illustrative purposes, only two criteria are utilized, while in the final application the whole $\mathcal{C}$ is used. Fig. 4 is an example for a tree spanned by two criteria, e.g. $C_1 = Type$ and $C_2 = Status$, each with two possible realizations. The depicted access is first identified as a read and the appropriate branch is selected. Next, the controller is identified as busy and the left branch is selected. Finally, the delay of the access is added to the histogram of the leaf, which represents the delay distribution of read accesses that have occurred while the controller has been busy. As an example, Fig. 5 shows the real histograms of the delays of read accesses, differentiated by the controller's state. Algorithm 1 lists the traversal process.C_{MY} is the criterion at the current node,$|C_{MY}|$ denotes the number of realizations of the criterion and $\vec{Q}_{out}$ is the vector of the resulting sub-queues from a criterion.

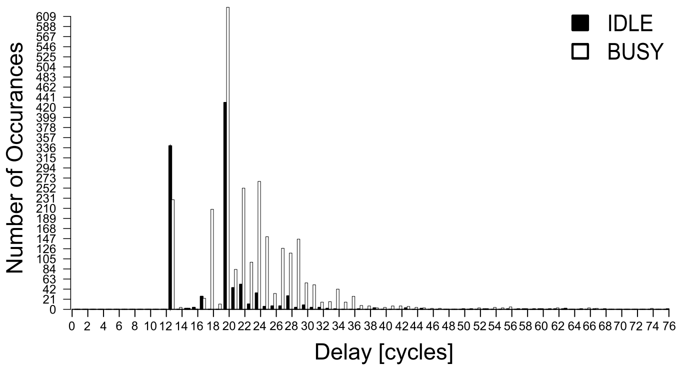

Fig. 5. Delay histograms for read accesses based on controller's state

Fig. 6 shows the example tree after the traversal has finished. To end up with more accurate statistical representation, effects like the additional delay from a memory refresh can be omitted from the original distributions. The refresh of a memory comes with a predefined period and duration. Thus, it can be easily, deterministically included in the TL model. Including the refresh impact in this manner makes the model more accurate as it preserves the periodicity.

Algorithm 1 Propagation Algorithm

```
1: PROCEDURE: $\mathbf{T_{class}.traverse(Q)}$
2: BEGIN
3: $root.propagate(Q, NIL)$ {NIL stands for nothing in list}
4: END
1: PROCEDURE: $\mathbf{N.propagate(Q, Q_{orig})}$
2: BEGIN
3: $\vec{Q}_{out} \leftarrow C_{MY}(Q, Q_{orig})$
4: for $n \in [1, |C_{MY}|]$ do
5:     $child[n].propagate(\vec{Q_{out}}[n], Q_{orig})$
6: end for
7: END
1: PROCEDURE: $\mathbf{K.propagate(Q, Q_{orig})}$
2: BEGIN
3: for $i \in [1, length(Q)]$ do
4:     $D_{leaf}.insert(Q_i.delay\_time)$
5: end for
6: END
```

At the end all $D \in \mathcal{D}$ are converted to cumulative distribution functions (CDFs) by discrete integration. Therefore, each class $K \in \mathcal{K}$ is mapped to a CDF $F \in \mathcal{F}$ with the bijective mapping $\eta : \mathcal{K} \to \mathcal{F}$. The statistical timing model is then represented by $T_{class} = (\mathcal{N}, \mathcal{K}, \rho, \mathcal{F}, \eta)$.

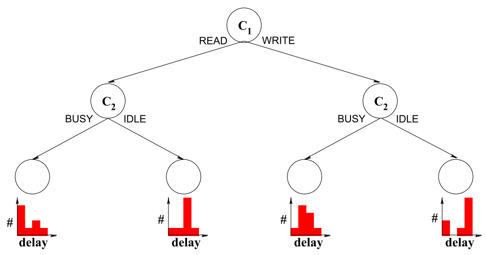

Fig. 6. Established PMFs

B. Transaction Level Model

T_{class} is loaded within a functional TL model and designated as T_{model}. It is used to classify incoming transactions. During the TL simulation, the properties of the controller and of each transaction are examined. Based on them a correct delay is assigned to the transaction. As an example, in Fig. 7 a read transaction occurs, while the controller is idling. Therefore, the `READ` and `IDLE` edges are taken, leading to the second leaf. The CDF associated with this leaf is used to determine the delay of the transaction because it describes the delay distribution of read accesses that occur when the controller is idle. As it is a distribution and not a deterministic value, a random delay is generated as follows.

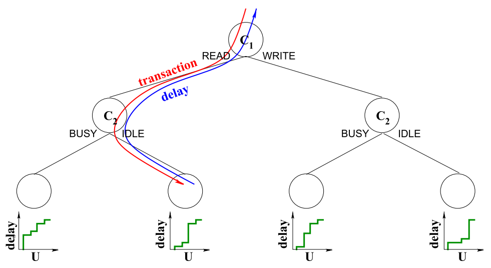

Fig. 7. Delay generation

The regeneration of a random variable X is done by the use of another uniformly distributed one $U \in \mathcal{U}[0, 1)$ and the CDF of X (CDF_X), which has to be invertible [12]. Eq.1-5 show that $x = CDF_X^{-1}(u)$, where x is the realization of X and u is the realization of U. This is a non-parametric approach for reconstructing a random variable [13]. Another approach is to extract characteristics like mean, variance and even higher moments and put them in an already known analytical form. It, however, is only applicable when the resulting CDFs can be described by analytical density functions, which is not the case.

$$U \in \mathcal{U}(0, 1) \;\wedge\; \exists\; CDF_X \tag{1}$$

$$CDF_U(u) = \int_0^u pdf_U(u')\mathrm{d}u' = u \tag{2}$$

$$\text{Find } f : x = f(u) \;\wedge\; u = f^{-1}(x)$$

$$\underbrace{\int_0^{u=f^{-1}(x)} pdf_U(u')du'}_{u} = \int_{-\infty}^{x} pdf_X(x')dx' \implies \tag{3}$$

$$u = f^{-1}(x) = CDF_X(x) \implies \tag{4}$$

$$x = CDF_X^{-1}(u) \tag{5}$$

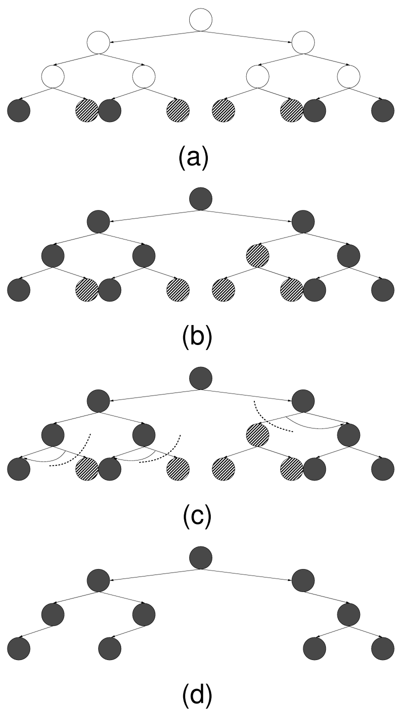

Fig. 8. Reduction process

Some of the leafs in $\mathcal{K}$ can be associated with empty distributions in $\mathcal{F}$ because no memory access has reached them during the measurements on the RTL results. Their delay is, thus, undefined. Due to differences between the levels of abstraction, a transaction during a TL simulation might reach such a leaf. To deal with this, T_{model} is pruned and all leafs associated with empty distributions are removed.

Fig. 8(a)-8(d) shows the pruning process. The approach is bottom-up. All empty leafs are identified and marked (stripes). Next, the nodes that have all their children empty are marked also as empty. The empty elements are cut out and links to their closest siblings are placed instead. This approach ensures that if not the correct one, then a close guess is returned as a delay.

The final TL model is composed of a scheduling algorithm, the timing model T_{model}, based on the whole $\mathcal{C}$, and a process, which deterministically adds the impact of a memory refresh. The refresh in this case is not included in the statistical part. It has a known period ($\tau_{refresh}$), duration ($d_{refresh}$), resulting in a penalty ($\Delta_{refresh}$) and is deterministically reconstructed (Fig. 9). Algorithm 2 shows how the refresh penalty is added to transactions (Θ).

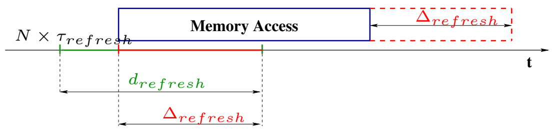

Fig. 9. Refresh mechanism

Algorithm 2 Refresh Engine Algorithm

```
PROCEDURE: $add\_refresh\_impact(\Theta)$
BEGIN
$t_{after} \leftarrow t_{sim} \mod \tau_{refresh}$ {Time after the occurrence of the last refresh}
{If the refresh occurs before or at the start of a transaction}
if $t_{after} \leq d_{refresh}$ then
    $\Delta_{refresh} \leftarrow d_{refresh} - t_{after}$
    return $\Theta.t_{wait} + \Delta_{refresh} \times \tau_{clk} - t_{after}$
end if
{If the refresh occurs during the transaction}
if $\tau_{refresh} - t_{after} \leq \Theta.t_{wait}$ then
    $\Delta_{refresh} \leftarrow d_{refresh}$
    return $\Theta.t_{wait} + \Delta_{refresh} \times \tau_{clk}$
end if
return $\Theta.t_{wait}$ {No hit by a refresh}
END
```

Fig. 10 depicts the memory controller model together with the memory module. The model has two processes, one for receiving transactions from the system (*Input*) and one for sending transactions, such as responses to read accesses, to the system (*Output*). The timing of the memory is, by construction, contained within the model of the controller. Thus, the model is attached to an untimed memory, which further reduces the simulation effort.

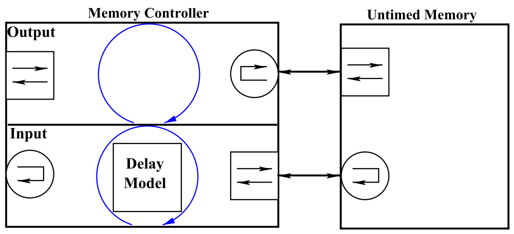

Fig. 10. TL model

On *Input* reside the delay model and the scheduling algorithm (Algorithm 3), which takes care of queuing up the transactions in a correct fashion. $\mathbf{E}_{new}$ is the event, notifying the presence of a new transaction. Lines 6 and 30 track the state of the controller. Lines 16-22 ensure that there is no out of order serving of read transactions (the RTL module also does not support it). When a transaction Θ_Z enters the model at t_Z it gets assigned a delay Δ_Z and enters the waiting queue Ω. It waits until its ending time is reached and gets pushed to the memory model. If meanwhile at t_{Z+1} another transaction Θ_{Z+1} enters the model, Θ_Z gets pulled out of the queue and gets assigned a successor Θ_{Z+1} (line 11). Its delay gets recomputed and is pushed back to Ω (lines 12-14). Θ_{Z+1} is assigned a delay Δ_{Z+1} and also inserted in the queue. When the simulation time reaches an ending time of a transaction, the transaction gets dispatched to the memory model (line 26).

Algorithm 3 Scheduling Algorithm

```
1: PROCEDURE: $KernelProcess()$
2: BEGIN
3: wait($\mathbf{E}_{new}$)
4: while true do {Main loop}
5:     if $\exists\Theta_{Z+1}$ then {If a new transaction has entered}
6:         $idle \leftarrow$ false
7:         $\Theta_{Z+1}.t_{wait} \leftarrow T_{model}.get\_wait(\Theta_{Z+1})$
8:         $\Theta_{Z+1}.t_{wait} \leftarrow add\_refresh\_impact(\Theta_{Z+1})$
9:         $\Theta_{Z+1}.t_{end} \leftarrow t_{sim} + \Theta_{Z+1}.t_{wait}$
10:        if $\exists\Theta_X \in \Omega \wedge overlap(\Theta_Z, \Theta_{Z+1})$ then
11:            $\Theta_X.successor \leftarrow \Theta_{Z+1}.type$
12:            $\delta \leftarrow \max(0, T_{model}.get\_wait(\Theta_X) - \Theta_X.t_{wait})$
13:            $\Theta_X.t_{wait} \leftarrow \Theta_X.t_{wait} + \delta$
14:            $\Theta_X.t_{end} \leftarrow \Theta_X.t_{wait} + \Theta_X.t_{begin}$
15:        end if
16:        if $\Theta_{Z+1}.type = Read$ then
17:            if $\exists\Theta_{LR} \in \Omega \wedge \Theta_{LR}.t_{end} \geq \Theta_{Z+1}.t_{end}$ then
18:                $\Theta_{Z+1}.t_{end} \leftarrow \Theta_{LR}.t_{end} + \tau_{clk} \times \Theta_{LR}.datawords$
19:                $\Theta_{Z+1}.t_{wait} \leftarrow \Theta_{Z+1}.t_{end} - t_{sim}$
20:            end if
21:            $\Theta_{LR} \leftarrow \Theta_{Z+1}$
22:        end if
23:        $\Omega.insert(\Theta_{Z+1})$
24:        $\Omega.sort()$
25:    else {A timeout has occurred}
26:        $dispatch(\Omega[0])$
27:        $\Omega.pop()$
28:    end if
29:    if $|\Omega| = 0$ then {Wait on a new transaction event}
30:        $idle \leftarrow$ true
31:        wait($\mathbf{E}_{new}$)
32:    else {Wait on a new transaction event with a timeout equal to $\Omega[0].t_{end} - t_{sim}$}
33:        wait($\mathbf{E}_{new} \vee \max(0, \Omega[0].t_{end} - t_{sim})$)
34:    end if
35: end while
36: END
```

Output contains a simple process, which returns responses of transactions of type read.

IV. Experimental Results

The modeling approach is applied to a memory controller used in industry. The timing accuracy of the TL model is compared to the one of the RTL model. For the construction of T_{class} all criteria from $\mathcal{C}$ were utilized. Additionally, 250 RTL simulations each consisting of 2000 random accesses with random sizes were used for constructing the delay distributions. The total amount of time required for obtaining all the simulation results and constructing the timing model is approximately 100 hours. In contrast, the estimated time for manual construction of such a model is 3 months. For the comparison step, both devices were stimulated with constrained random accesses/transactions, produced by equivalent generators. Each simulation was performed with 2000 accesses/transactions. The average cumulative delays for both models were then compared for accesses of different lengths. Fig. 11 presents the results. In the figure $SINGLE = 1$ dataword, $BURST2 = 2$ datawords, $BURST4 = 4$ datawords, and $BURST8 = 8$ datawords. $MIXED$ represents the test-case with all lengths combined and equally probable.

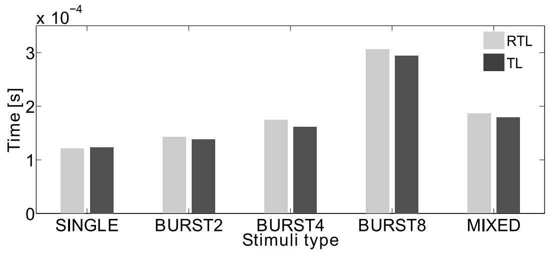

Fig. 11. Average cumulative delay for 2000 accesses/transactions

Fig. 12 shows the delay distributions of $BURST8$ accesses/transactions of the RTL and the TL models. The delays exhibited by the TL model mimic the ones by RTL one.

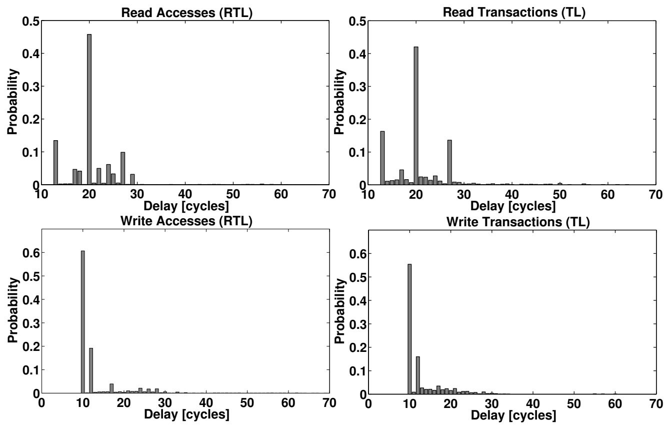

Fig. 12. Distributions of assigned delays

Table I shows the averages of the cumulative delay for all 2000 accesses/transactions to the memory controller for the RTL reference simulation and the TL simulation. Except for the $SINGLE$ dataword accesses, the delays estimated by the TL model are slightly smaller as compared to the ones of the reference RTL simulation. The worst deviation of $7,66\%$ is observed for BURST4 accesses. The average execution time to simulate a test case on TL is approximately 6 seconds. The average execution time to simulate a test case on RTL is approximately 2 hours. Hence, a speed gain of approximately $1200\times$ is observed for the simulation of the memory controller. As simulating a SoC on RTL is infeasible, the time required for constructing the TL model using this methodology is reasonable, given its simulation speed.

TABLE I
COMPARISON OF CUMULATIVE DELAYS

	RTL [s]	**TL [s]**	**Error [%]**
SINGLE	1.2173e-04	1.2365e-04	-1.58
BURST2	1.4283e-04	1.3775e-04	3.55
BURST4	1.7453e-04	1.6116e-04	7.66
BURST8	3.0652e-04	2.9455e-04	3.90
MIXED	1.8644e-04	1.7900e-04	3.99

V. Conclusion

The presented approach for abstracting a cycle-approximate TL memory controller model from RTL description is fast, highly accurate and results in a model with low complexity and excellent simulation speed. It is automated and RTL specifics independent, producing a cycle-approximate, timed model in less than a week. The low error margin between -1.58% and 7.66% and the huge speed-up of $1200\times$ make the resulting model usable in system level simulations and virtual prototypes. As future work, the methodology can be extended to other devices beyond memory controllers. Furthermore, the timing model can be changed to a more sophisticated one, e.g. by using neural networks.

References

[1] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*, 2nd ed. Springer, 2010.
[2] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara, "From VHDL Register Transfer Level to SystemC Transaction Level Modeling a Comparative Case Study," in *16th Symposium on Integrated Circuits and Systems Design (SBCCI'03)*, 2003.
[3] G. Stehr and J. Eckmüller, "Transaction Level Modeling in Practice: Motivation and Introduction," in *IEEE International Conference on Computer-Aided Design (ICCAD), 2010*, 2010.
[4] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," in *CODES+ISSS'03*, October 2003.
[5] N. Bombieri, N. Deganello, and F. Fummi, "Integrating RTL IPs into TLM designs through automatic transactor generation," in *Design, automation and test in Europe*, 2008.
[6] "http://www.veripool.org/wiki/verilator."
[7] "http://www.mazdak-alborz.com/v2sc.html."
[8] Y.-H. Liaw, S.-H. Hung, and C.-H. Tu, "V2X: An Automated Tool for Building SystemC-based Simulation Environments in Designing Multicore Systems-on-Chips," in *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2010.
[9] N. Bombieri, F. Fummi, and G. Pravadelli, "A Methodology for Abstracting RTL Designs into TL Descriptions," in *4th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, 2006.
[10] N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions," in *IEEE Transactions on Computers*, 2010.
[11] C. Kwang-Ting and A. Krishnakumar, "Automatic Functional Test Generation Using The Extended Finite State Machine Model," in *30th Design Automation Conference*, 1993.
[12] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, A. V. Balakrishnan, G. Dantzig, and L. Zadeh, Eds. McGRAW-HILL KOGAKUSHA, LTD., 1965.
[13] F. Flaamennt, S. Guilley, J.-L. Danger, M. A. Elaabid, H. Maghrebi, and L. Sauvage, "About Probability Denstiy Function Estimation for Side Channel Analysis," in *COSADE 2010 - First International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2010.