2018
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE

Performance modeling and timing verification for DRAM memory subsystems

DRAFT PAPER v1

Thomas Schuster, Peter Prüller, Christian Sauer

Cadence Design Systems GmbH, Munich, Germany (thomschu @ cadence.com)

***Abstract* - Contemporary System-on-Chips (SoCs) comprise multiple processing elements, such as CPUs, DSPs and custom accelerators. All these components are competing for shared resources, most of all memory bandwidth. It is undisputed that orchestrating the memory accesses in a modern SoC is one of the biggest challenges today. Due to the high complexity designers often resort to emulation using dedicated hardware for identifying bandwidth and latency issues. An alternative is the use of abstract models trading a certain amount of accuracy for a substantial gain in simulation performance. The latter has many obvious advantages, especially in the early stages of system exploration, but is often not applicable, because the required simulation-IPs are not available and development would take too much time. We can address this issue by providing a framework for the modeling of DRAM memory subsystems for performance exploration, based on SystemC and the Approximately-Timed (AT) coding style of TLM2.0. The models provide the same programming interface and a significant subset of the features of a widely used industrial memory controller RTL IP. Users may explore various command re-ordering strategies, different internal fifo layouts, port arbitration and more.**

In a case study instances of the simulation model targeting LPDDR4 have been compared against the RTL reference. We can achieve a speedup of 100x in full load simulation. The tolerated timing deviation, in terms of simulated time and average latency, is in a pessimistic corridor of 0 – 15%. We present our modeling approach, verification environment, and achieved results.

Keywords—SystemC, TLM, Approximately-timed, DDR, DRAM, Memory controller

I. Introduction / related work

Today Dynamic Random Access Memory (DRAM) is part of almost every SoC package. Since the introduction in the late 1990s many generations and flavors of this cheap high capacity memory have been successfully brought to market. Latest addition to the family are DDR4 and its low-power sibling LPDDR4. Structure and interfaces of DRAM have become an industry standard maintained by the Joint Electron Device Engineering Council (JEDEC) [1]. DRAM performance is crucial to modern systems, because it is usually shared amongst different processing elements like CPUs, DSPs, or custom accelerators. Moreover, it often constitutes the lowest level of the volatile memory hierarchy, below first and secondary caches, hence being responsible for the worst-case memory access latency. Latency of DRAMs natively is higher and throughput is lower compared to on-chip RAMs. Additional complications for system designers are caused by the rather complex internal state of DRAMs, which makes access performance highly traffic dependent. Usually the memory is spread over multiple ranks (chip selects) that connect to the chip through common address and data busses. Each rank consists of multiple memory banks, each of which holds a grid of memory rows and columns. Accesses to a certain memory location must be partitioned into bursts of a constant length and need to be prepped by exactly timed memory control commands (e.g. precharge, activate). More details about DRAM memory architecture and memory access timing can be found in [2].

The importance of estimating and optimizing memory timing has sparked numerous research activities. Many of them focus on on-chip cache optimization, ignoring DRAM timing, or assigning a static penalty per access. In [3] Cheung et. al presents a method where memory accesses in an MPSoC are identified at software compilation time, and statically annotated onto SystemC models. An approach for the automatic construction of timing models for DRAM controllers using statistical measurement from an RTL reference is presented in [4]. Although very promising it remains unclear if statistically or neural learning techniques can be efficiently applied for modeling advanced DRAM controllers with complex internal state and extensive command reordering. Practical application requires models to be reprogrammable, preferable without having to re-train, and a proof-of-correctness within a parameter corridor that cannot be easily obtained using statistic methods. Opposite to statistical timed models resides the direct translation of an RTL memory controller into a C/C++ simulation model, using tools like ARM's Cycle Model Studio (former Carbon) [5]. Such models offer the desired verification coverage, but are only slightly faster the RTL simulation, hence of limited use for system-level exploration.

Efficient exploration of memory access timing requires reliable models with guaranteed correctness within a defined parameter range. Correctness should be defined as a corridor of tolerated error concerning metrics like simulated time / throughput and latency. Ideally this error corridor should be in the pessimistic range, so that predicted performance is never above the actual performance of the reference design. The tolerated error corridor allows for a certain amount of abstraction in the behavior that can be used to speed up simulation.

II. Application

We advocate an approach mixing explicit modeling of algorithms and abstraction similar to [6]. Our implementation is based on SystemC, consequently follows the approximately-timed (AT) coding style proposed by TLM 2.0 [7], and is therefore more widely applicable (e.g. Virtual Prototypes). The TLM is generated from a generic template, which can be retargeted for multiple DRAM types. Until now we have applied our flow to LPDDR2, DDR3, DDR4, and LPDDR4. Internal command and data processing are represented by a data flow model built on SystemC ports and exports. External interfaces are extended TLM2.0 compliant sockets that allow for independent read/write channels and data interleaving as required for AXI. To our best knowledge no comparable design exists. In this work we will focus on the chosen modeling approach, and the verification environment including selected performance metrics and measurement.

A. *Modeling Approach*

A block diagram of the SystemC data flow architecture is shown in Figure 1. The design is partitioned into the register control interface that can be automatically generated from an IP-XACT or RDL description, and the generic core, which can be parametrized using a Configuration, Control and Inspection (CCI) interface [8]. Condensed lists of hardware and software controllable features are given in Table 1 and Table 2.

The generic core of the model (`class ddr`) contains the data path of the controller with all its subcomponents. Most of these subcomponents are C++ traits, allowing features and algorithms to be conveniently added, replaced, or modified. In the following we describe the flow of a transaction through the system touching all relevant bases.

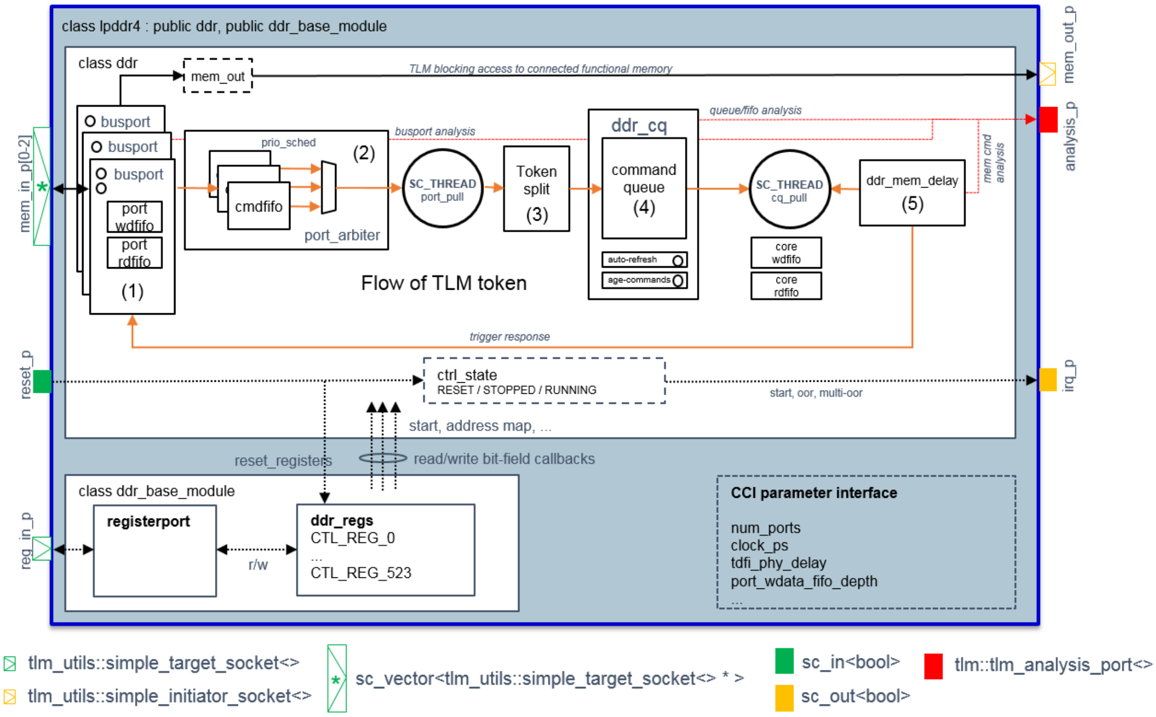


Figure 1 - TLM data flow

1) TLM interface / busport

The path of a transaction through the controller's core starts at one of the `mem_in_p[N]` TLM target sockets. Each socket is handled by a private bus port class (`busportX`). The interface is fully TLM2 compliant, but for best performance initiators need to provide additional information in form of 'ignorable' payload extensions. Amongst these extensions are AXI ID, burst information (length, width, type), priority (AXI QoS), cacheability, exclusiveness, explicit precharge request, and various debug fields (e.g. global id, time stamps). Another payload extension enables the generation of split responses, allowing the controller to extend the TLM2 base protocol by two additional phases signaling begin and end of read data splits. The granularity of these splits is limited to memory burst boundaries, which are translated into AXI bursts of the selected streaming width. To further reflect the nature of AXI, the model allows the user to go beyond standard by accepting reads and writes on a single socket in parallel. This way independent channels can be simulated without loosing backward compatibility and standard compliance. Next to handling the TLM protocol, the `busport` class generates one or multiple dataflow tokens for each incoming transaction (`begin_req`). Token objects are collected and reused in a memory pool. The number of generated tokens depends on the size of the burst, and the capacity of the port data holding queues. A token contains a pointer to the originating transaction, and various decoded information, such as row-bank-column address, number of memory bursts, command priority, and more. The bus port is ready to accept the next transaction of the same type after the last token associated with the current transaction has been accepted into the core (`end_req`). The core will then call back the bus port whenever a response (or a split response) is ready. In case of read operations, `begin_resp` is sent at the estimated time of the first beat of read data available on the AXI read data channel. In a split response scenario `begin_resp` moves to the first beat of the last split to keep the base protocol intact. Write response generation is programmable. Bufferable write responses are generated after write data acceptance, co-bufferable responses after multi-port arbitration, and non-bufferable responses after final memory command execution.

Table 1 - Selected HW parameters (CCI)

Parameter	Description	Runtime
clock_ps	core clock rate	Yes
def_freq_set	default frequency set	No
num_ports	number of bus ports	No
chip_selects	number of chip selects	No
banks_per_chip	banks per chip select	No
datapath_width	width of datapath in bits	No
priority_width	width of priority field	No

Parameter	Description	Runtime
portX_clock_ps	port X clock rate	Yes
high_speed	mem-to-core clock ratio	Yes
cq_depth	depth of command queue	No
sel_depth	depth of selection window	No
other fifos	various other fifos / queue depths	No
other analysis	on/off various instrumentations	Yes
other timing	various timing correction and tuning	Yes

Table 2 - Selected SW programmable features

Bitfields	Description
row/bank/col diff	define layout of address map
max col/row/cs	layout of address map
irq stat/mask/ack	interrupt control
out-of-range	oor signaling (addr, src id, type, ..)
age_count	cmd age counter control
Bstlen	burst length

Bitfields	Description
placement_en	enable placement logic
in_order_accept	control ooo command selection
various reorder controls	priority, grouping & splitting schemes
various coherency ctrl	address, source id, port id, cmd type
various port arbiter ctrl	port bdw limits and monitors, fixed prio,…
various DRAM timings	TRP, TRCD, TCCD, CASLAT, .. (> 50)

2) Multi-port arbitration

Each busport pushes accepted command tokens into a colored array of fifos (`cmdfifoX`). The multi-port arbiter is driven by an sc_thread (`port_pull`), pulling commands from those fifos, at the rate of the controller clock. On each pull request bandwidth monitor (bw_mon) and priority scheduler (`prio_sched`) decide which port and command to prioritize. Port priorities can be programmed as fixed, giving one port a higher priority than another, or mapped on a command-by-command basis to the AXI QoS extension. Independently of priority the bandwidth monitor may skip ports if their bandwidth limits are reached. To avoid active idle threads `port_pull` is put to sleep whenever the input fifos deplete. Wakeup is implemented using a SystemC `sc_event_or_list`.

3) Token split

After port arbitration the `port_pull` thread pushes the command tokens into the command splitting stage. Main purpose of this component is the expansion of read-modify-write sequences, which is needed for memories without write masking. Additionally, checks on memory boundaries are carried out that might also cause a command to split. All required extra commands are clones of the original command token, hence also reference the same originating TLM transaction. A unique sequence ID is used to sustain the command order. An ongoing command split stalls the previous pipeline stages, namely puts the multi-port arbiter on hold. Tokens are pushed out towards the command queue (`ddr_cq`) with minimum distance of one cycle delay.

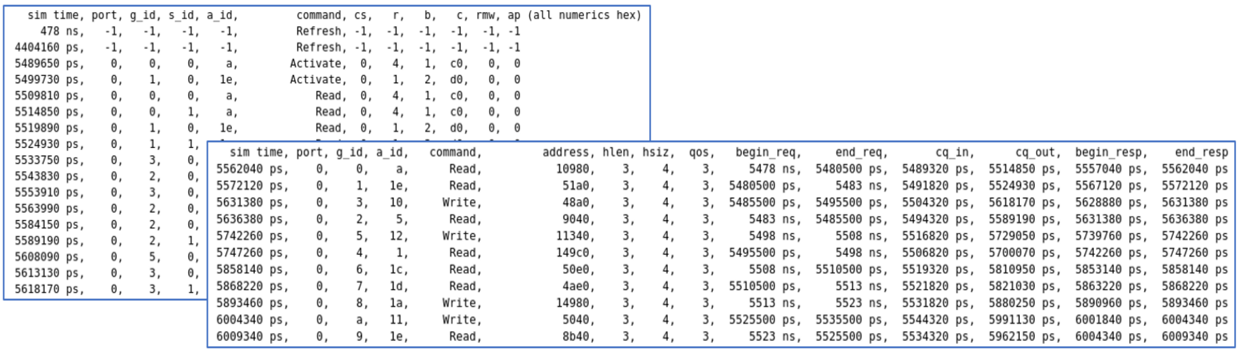
4) Command queue

Most industrial relevant DDR memory controllers feature an internal command queue [9] [10]. The queue is used to deduct the optimal command execution order based on memory state and command priority. The modeled controller uses out-of-order command insertion and out-of-order command selection to establish this optimum. All reordering rules, such as bank splitting, read/write grouping, write-to-read splitting, page grouping, or chip select grouping, can be enabled/disabled by software. These algorithms must be modeled very detailed. In our work we found only little room for abstraction at this point. Even with great care deviations may occur, because multiple unsynchronized threads are working on the same shared data structure. Our TLM implements the reordering queue as a C++ `std::list` optimized for fast insertion and removal of elements. As mentioned, new upstream commands are pushed in from the command splitting stage. Command selection/execution and memory preparation are driven by thread `cq_pull`. If the command queue is not empty, this thread triggers with an interval of two clock cycle (for lpddr4), and otherwise goes to sleep. On each activation the queue checks if one of the commands within the selection window is ready to execute by comparison against timestamps calculated during memory preparation. Multiple memory bursts may be associated with each command token. A command token remains in the queue until the last of its bursts has been committed. Therefore, ddr commands (e.g. cas2) and memory bursts for different user commands may be interleaved at the memory interface. Resource allocation is also handle on a burst-by-burst basis. A read burst can only be requested if the read data fifos of the core have enough space to accommodate at least one burst of data. In a similar way write requests need to be able to allocate sufficient space in the core write data fifo to be allowed to run. If no user command has a burst that is ready to go, memory preparation tries to schedule bank open (activate) and bank closing (precharge) commands for all tokens within a programmable bank prep window. Auto-refresh and command aging are implemented in infrequently running `sc_methods`.

5) DDR mem delay

For each committed ddr read or write command a clone of the respective user command is forwarded to the `ddr_mem_delay` stage, which will take care of write delay and read round-trip delay. Both delays are internally calculated as sums of software programmable parameters (caslat, wrlat), and configurable PHY latency. Payload event queues are used for synchronization. For each arriving burst the internal queues are offloaded/deallocated. Moreover, it will be decided whether a callback to the bus port for response generation is required or not. The latter depends on command type, sequence ID (first or last burst of transaction), and response type.

Most of the mentioned components are instrumented for analysis and broadcast their data over a TLM analysis socket implemented on top-level. Generation of analysis information is controlled per block via CCI. Analysis logs for the ddr memory interface, and the axi tlm interface are shown in Figure 2.



```
  sim time, port, g_id, s_id, a_id,        command, cs,   r,   b,   c, rmw, ap (all numerics hex)
    478 ns,   -1,   -1,   -1,   -1,        Refresh, -1,  -1,  -1,  -1,  -1, -1
4404160 ps,   -1,   -1,   -1,   -1,        Refresh, -1,  -1,  -1,  -1,  -1, -1
5489650 ps,    0,    0,    0,    a,       Activate,  0,   4,   1,  c0,   0,  0
5499730 ps,    0,    1,    0,   1e,       Activate,  0,   1,   2,  d0,   0,  0
5509810 ps,    0,    0,    0,    a,           Read,  0,   4,   1,  c0,   0,  0
5514850 ps,    0,    0,    1,    a,           Read,  0,   4,   1,  c0,   0,  0
5519890 ps,    0,    1,    0,   1e,           Read,  0,   1,   2,  d0,   0,  0
5524930 ps,    0,    1,    1,
5533750 ps,    0,    3,    0,
5543830 ps,    0,    2,    0,
5553910 ps,    0,    3,    0,
5563990 ps,    0,    2,    0,
5584150 ps,    0,    2,    0,
5589190 ps,    0,    2,    1,
5608090 ps,    0,    5,    0,
5613130 ps,    0,    3,    0,
5618170 ps,    0,    3,    1,
```

```
  sim time, port, g_id, a_id,   command,        address, hlen, hsiz,  qos,   begin_req,     end_req,       cq_in,      cq_out,  begin_resp,    end_resp
5562040 ps,    0,    0,    a,      Read,          10980,    3,    4,    3,     5478 ns,  5480500 ps,  5489320 ps,  5514850 ps,  5557040 ps,  5562040 ps
5572120 ps,    0,    1,   1e,      Read,           51a0,    3,    4,    3,  5480500 ps,     5483 ns,  5491820 ps,  5524930 ps,  5567120 ps,  5572120 ps
5631380 ps,    0,    3,   10,     Write,           48a0,    3,    4,    3,  5485500 ps,  5495500 ps,  5504320 ps,  5618170 ps,  5628880 ps,  5631380 ps
5636380 ps,    0,    2,    5,      Read,           9040,    3,    4,    3,     5483 ns,  5485500 ps,  5494320 ps,  5589190 ps,  5631380 ps,  5636380 ps
5742260 ps,    0,    5,   12,     Write,          11340,    3,    4,    3,     5498 ns,     5508 ns,  5516820 ps,  5729050 ps,  5739760 ps,  5742260 ps
5747260 ps,    0,    4,    1,      Read,          149c0,    3,    4,    3,  5495500 ps,     5498 ns,  5506820 ps,  5700070 ps,  5742260 ps,  5747260 ps
5858140 ps,    0,    6,   1c,      Read,           50e0,    3,    4,    3,     5508 ns,  5510500 ps,  5519320 ps,  5810950 ps,  5853140 ps,  5858140 ps
5868220 ps,    0,    7,   1d,      Read,           4ae0,    3,    4,    3,  5510500 ps,     5513 ns,  5521820 ps,  5821030 ps,  5863220 ps,  5868220 ps
5893460 ps,    0,    8,   1a,     Write,          14980,    3,    4,    3,     5513 ns,     5523 ns,  5531820 ps,  5880250 ps,  5890960 ps,  5893460 ps
6004340 ps,    0,    a,   11,     Write,           5040,    3,    4,    3,  5525500 ps,  5535500 ps,  5544320 ps,  5991130 ps,  6001840 ps,  6004340 ps
6009340 ps,    0,    9,   1e,      Read,           8b40,    3,    4,    3,     5523 ns,  5525500 ps,  5534320 ps,  5962150 ps,  6004340 ps,  6009340 ps
```

Figure 2 - Analysis logs for memory and axi tlm interface

6) Functional Memory

Delay estimation is solely done by the controller TLM. The DDR memory is an untimed TLM simulation memory that must be bound to socket `mem_out_p`. Different memory types can be explored by loading appropriate delay information into the register bank. Clock rates, PHY delays, and other relevant timings are configured via CCI. The model detects changes on these parameters to keep all access delays up to date.

Verification Environment and Accuracy Measurement

The presented TLM aims for the architecture exploration use case. Therefore, validation of functional correctness and timing are needed. For the first we use a set of self-checking tests focusing on: datapath operations, exclusive access, interrupt generation / out-of-range detection, reset behavior, register bank, placement, and port arbitration. Some of these tests contain sequences of random test patterns, but most of them are explicitly stressing features and corner cases (directed). This approach is currently deemed sufficient. Upgrade to a full coverage driven flow is in progress. Timing verification on the other hand is far more complicated. The approximately-timed nature of the design under test (DUT), and the necessary abstractions at the bus interfaces make a comparison on a transaction-by-transaction basis against the RTL reference impractical. The TLM intends to predict the behavior of the RTL for various traffic scenarios within a defined configuration space, thereby trading a certain amount of accuracy for simulation speed. Hence, verification metrics must be arithmetic means or extreme values within a time interval (min/max), rather than one-to-one pass/fail conditions. For the start we decided to focus on the overall runtime of a test, as well as minimum/average/maximum latency for read and write operations within a full test. For the near future we are planning to extend this approach towards smaller timing windows.

Our verification setup is shown in Figure 3. It is used for both: functional and timing verification. Simulation model and RTL reference are alternatively bound to the same SystemC test bench. The test bench configures and programs both designs from the same input. DUT data sockets are directly bound using an `sc_vector` of sockets. The RTL reference is connected through AHB and AXI Verification IPs (VIP) in TLM-to-RTL transactor mode.

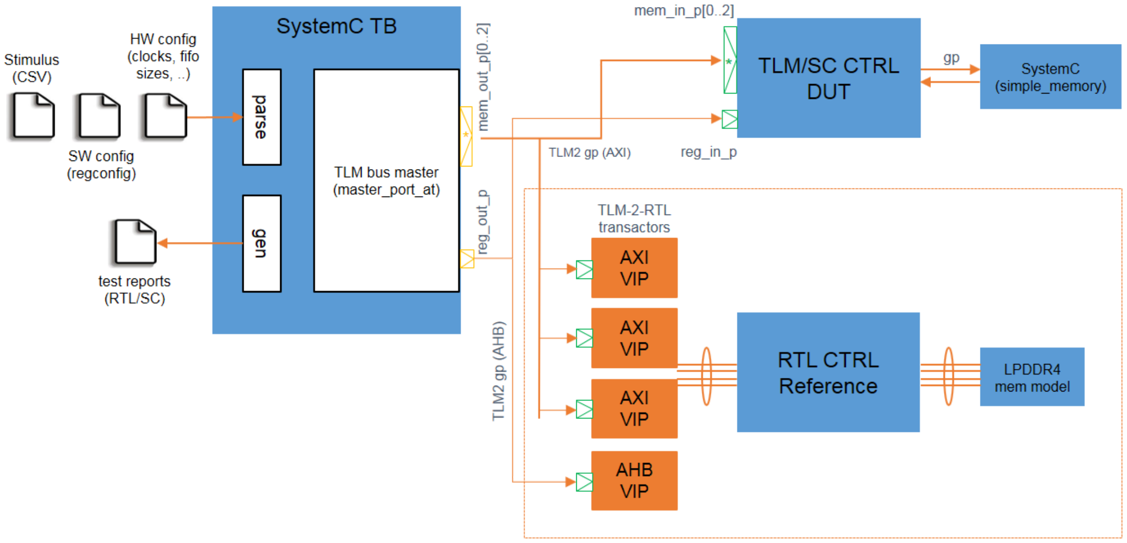
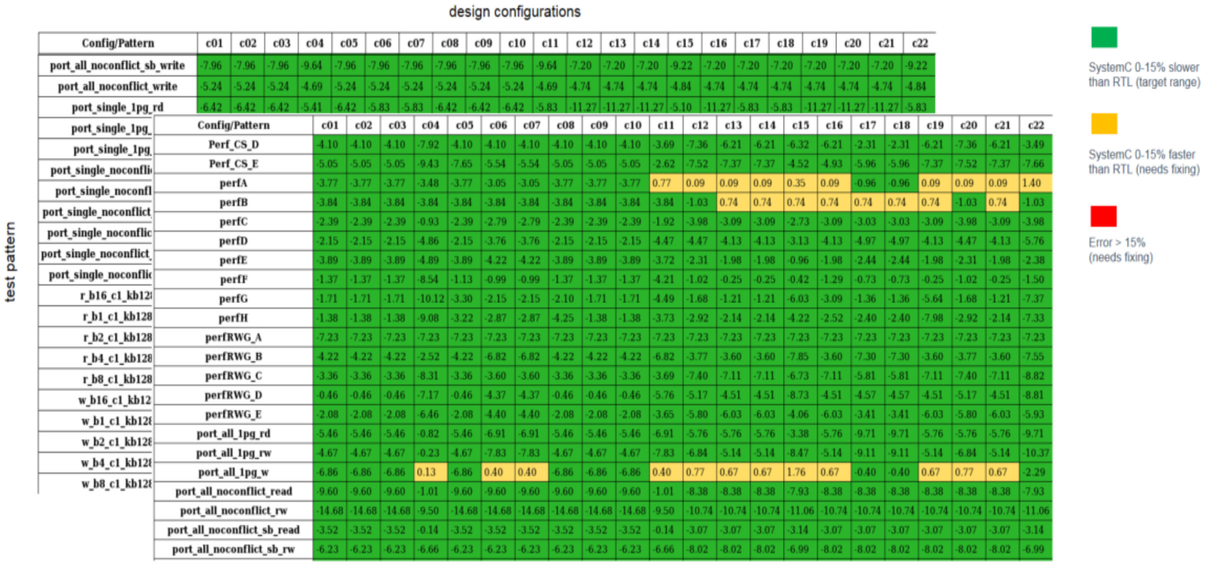


Figure 3 - Verification setup

III. RESULTS

Performance of the model was verified in multiple software and hardware configurations. Software configurations being programmable features, such as timing and command reordering options, and hardware configuration being queue sizes, data widths and memory layouts. Our test deck comprises almost 500 constructed patterns sweeping explicit traffic properties, accompanied by a lower number of random tests, and real-life traffic from a CPU recorded on RTL. Figure 4 illustrates the measured error in simulated time (equivalent to throughput) for a subset of tests (rows) and configurations (columns). Results are given as deviation in % from the RTL reference. Most of the test cases lay within the desired pessimistic prediction corridor (0-15% slower than RTL). Usually, random or real-life traffic tests achieve significantly higher accuracy than constructed tests, which often contain repetitive patterns that cause small deviations to accumulate.


design configurations
test pattern
🟩 SystemC 0-15% slower than RTL (target range)
🟨 SystemC 0-15% faster than RTL (needs fixing)
🟥 Error > 15% (needs fixing)

Config/Pattern	c01	c02	c03	c04	c05	c06	c07	c08	c09	c10	c11	c12	c13	c14	c15	c16	c17	c18	c19	c20	c21	c22
port_all_noconflict_sb_write	-7.96	-7.96	-7.96	-9.64	-7.96	-7.96	-7.96	-7.96	-7.96	-7.96	-9.64	-7.20	-7.20	-7.20	-9.22	-7.20	-7.20	-7.20	-7.20	-7.20	-7.20	-9.22
port_all_noconflict_write	-5.24	-5.24	-5.24	-4.69	-5.24	-5.24	-5.24	-5.24	-5.24	-5.24	-4.69	-4.74	-4.74	-4.74	-4.84	-4.74	-4.74	-4.74	-4.74	-4.74	-4.74	-4.84
port_single_1pg_rd	-6.42	-6.42	-6.42	-5.41	-6.42	-5.83	-5.83	-6.42	-6.42	-6.42	-5.83	-11.27	-11.27	-11.27	-5.10	-11.27	-5.83	-5.83	-11.27	-11.27	-11.27	-5.83
port_single_1pg_																						
port_single_1pg_																						
port_single_noconfli																						
port_single_noconfl																						
port_single_noconflict_																						
port_single_noconflic																						
port_single_noconflict_																						
port_single_noconflic																						
r_b16_c1_kb12																						
r_b1_c1_kb128																						
r_b2_c1_kb128																						
r_b4_c1_kb128																						
r_b8_c1_kb128																						
w_b16_c1_kb12																						
w_b1_c1_kb12																						
w_b2_c1_kb12																						
w_b4_c1_kb12																						
w_b8_c1_kb12																						

Config/Pattern	c01	c02	c03	c04	c05	c06	c07	c08	c09	c10	c11	c12	c13	c14	c15	c16	c17	c18	c19	c20	c21	c22
Perf_CS_D	-4.10	-4.10	-4.10	-7.92	-4.10	-4.10	-4.10	-4.10	-4.10	-4.10	-3.69	-7.36	-6.21	-6.21	-6.32	-6.21	-2.31	-2.31	-6.21	-7.36	-6.21	-3.49
Perf_CS_E	-5.05	-5.05	-5.05	-9.43	-7.65	-5.54	-5.54	-5.05	-5.05	-5.05	-2.62	-7.52	-7.37	-7.37	-4.52	-4.93	-5.96	-5.96	-7.37	-7.52	-7.37	-7.66
perfA	-3.77	-3.77	-3.77	-3.48	-3.77	-3.05	-3.05	-3.77	-3.77	-3.77	0.77	0.09	0.09	0.09	0.35	0.09	-0.96	-0.96	0.09	0.09	0.09	1.40
perfB	-3.84	-3.84	-3.84	-3.84	-3.84	-3.84	-3.84	-3.84	-3.84	-3.84	-3.84	-1.03	0.74	0.74	0.74	0.74	0.74	0.74	0.74	-1.03	0.74	-1.03
perfC	-2.39	-2.39	-2.39	-0.93	-2.39	-2.79	-2.79	-2.39	-2.39	-2.39	-1.92	-3.98	-3.09	-3.09	-2.73	-3.09	-3.03	-3.03	-3.09	-3.98	-3.09	-3.98
perfD	-2.15	-2.15	-2.15	-4.86	-2.15	-3.76	-3.76	-2.15	-2.15	-2.15	-4.47	-4.47	-4.13	-4.13	-3.13	-4.13	-4.97	-4.97	-4.13	-4.47	-4.13	-5.76
perfE	-3.89	-3.89	-3.89	-4.89	-3.89	-4.22	-4.22	-3.89	-3.89	-3.89	-3.72	-2.31	-1.98	-1.98	-0.96	-1.98	-2.44	-2.44	-1.98	-2.31	-1.98	-2.38
perfF	-1.37	-1.37	-1.37	-8.54	-1.13	-0.99	-0.99	-1.37	-1.37	-1.37	-4.21	-1.02	-0.25	-0.25	-0.42	-1.29	-0.73	-0.73	-0.25	-1.02	-0.25	-1.50
perfG	-1.71	-1.71	-1.71	-10.12	-3.30	-2.15	-2.15	-2.10	-1.71	-1.71	-4.49	-1.68	-1.21	-1.21	-6.03	-3.09	-1.36	-1.36	-5.64	-1.68	-1.21	-7.37
perfH	-1.38	-1.38	-1.38	-9.08	-3.22	-2.87	-2.87	-4.25	-1.38	-1.38	-3.73	-2.92	-2.14	-2.14	-4.22	-2.52	-2.40	-2.40	-7.98	-2.92	-2.14	-7.33
perfRWG_A	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23	-7.23
perfRWG_B	-4.22	-4.22	-4.22	-2.52	-4.22	-6.82	-6.82	-4.22	-4.22	-4.22	-6.82	-3.77	-3.60	-3.60	-7.85	-3.60	-7.30	-7.30	-3.60	-3.77	-3.60	-7.55
perfRWG_C	-3.36	-3.36	-3.36	-8.31	-3.36	-3.60	-3.60	-3.36	-3.36	-3.36	-3.69	-7.40	-7.11	-7.11	-6.73	-7.11	-5.81	-5.81	-7.11	-7.40	-7.11	-8.82
perfRWG_D	-0.46	-0.46	-0.46	-7.17	-0.46	-4.37	-4.37	-0.46	-0.46	-0.46	-5.76	-5.17	-4.51	-4.51	-8.73	-4.51	-4.57	-4.57	-4.51	-5.17	-4.51	-8.81
perfRWG_E	-2.08	-2.08	-2.08	-6.46	-2.08	-4.40	-4.40	-2.08	-2.08	-2.08	-3.65	-5.80	-6.03	-6.03	-4.06	-6.03	-3.41	-3.41	-6.03	-5.80	-6.03	-5.93
port_all_1pg_rd	-5.46	-5.46	-5.46	-0.82	-5.46	-6.91	-6.91	-5.46	-5.46	-5.46	-6.91	-5.76	-5.76	-5.76	-3.38	-5.76	-9.71	-9.71	-5.76	-5.76	-5.76	-9.71
port_all_1pg_rw	-4.67	-4.67	-4.67	-0.23	-4.67	-7.83	-7.83	-4.67	-4.67	-4.67	-7.83	-6.84	-5.14	-5.14	-8.47	-5.14	-9.11	-9.11	-5.14	-6.84	-5.14	-10.37
port_all_1pg_w	-6.86	-6.86	-6.86	0.13	-6.86	0.40	0.40	-6.86	-6.86	-6.86	0.40	0.77	0.67	0.67	1.76	0.67	-0.40	-0.40	0.67	0.77	0.67	-2.29
port_all_noconflict_read	-9.60	-9.60	-9.60	-1.01	-9.60	-9.60	-9.60	-9.60	-9.60	-9.60	-1.01	-8.38	-8.38	-8.38	-7.93	-8.38	-8.38	-8.38	-8.38	-8.38	-8.38	-7.93
port_all_noconflict_rw	-14.68	-14.68	-14.68	-9.50	-14.68	-14.68	-14.68	-14.68	-14.68	-14.68	-9.50	-10.74	-10.74	-10.74	-11.06	-10.74	-10.74	-10.74	-10.74	-10.74	-10.74	-11.06
port_all_noconflict_sb_read	-3.52	-3.52	-3.52	-0.14	-3.52	-3.52	-3.52	-3.52	-3.52	-3.52	-0.14	-3.07	-3.07	-3.07	-3.14	-3.07	-3.07	-3.07	-3.07	-3.07	-3.07	-3.14
port_all_noconflict_sb_rw	-6.23	-6.23	-6.23	-6.66	-6.23	-6.23	-6.23	-6.23	-6.23	-6.23	-6.66	-8.02	-8.02	-8.02	-6.99	-8.02	-8.02	-8.02	-8.02	-8.02	-8.02	-6.99

Figure 4 – Error in simulated time for tests in %

Exemplary results for latency measurement are shown in Figure 5. The plot shows the latency per transaction in bus clock cycles for random traffic from three ports. The transaction latency rapidly increases soon after simulation start. Reasons are: 1) the high number of concurrent requests from the ports, and 2) the random nature of the traffic giving the model only little room for optimization (high number of bank activations required). Hence, for the given test case, the internal queues quickly stall the data path, as can be seen in Figure 6 on the example of the command reordering queue. Another observation that can be made here is the effect of the auto-refresh, which blocks access to memory in regular intervals. The refreshes add additional latency causing worst case response delays of up to 600 cycles.

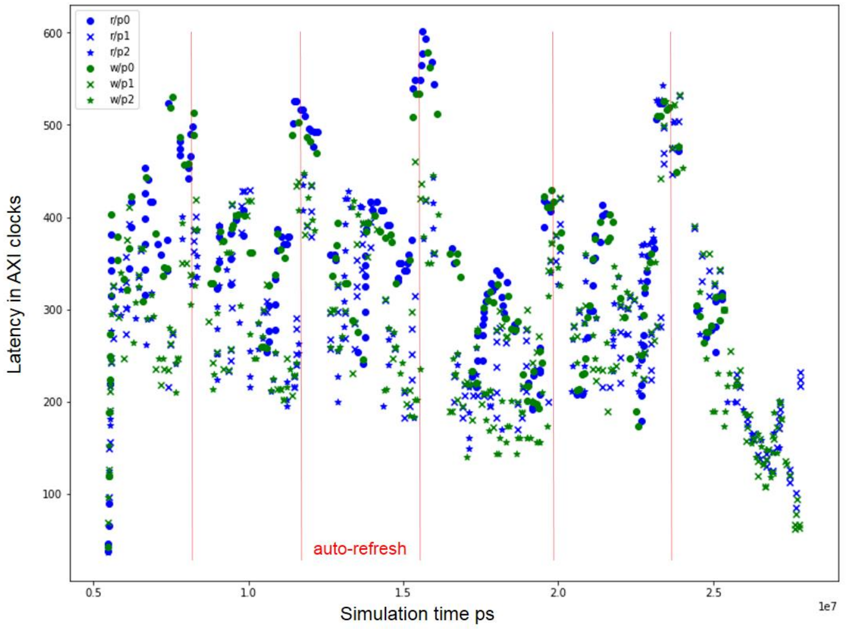


Figure 5 - Latency per transaction

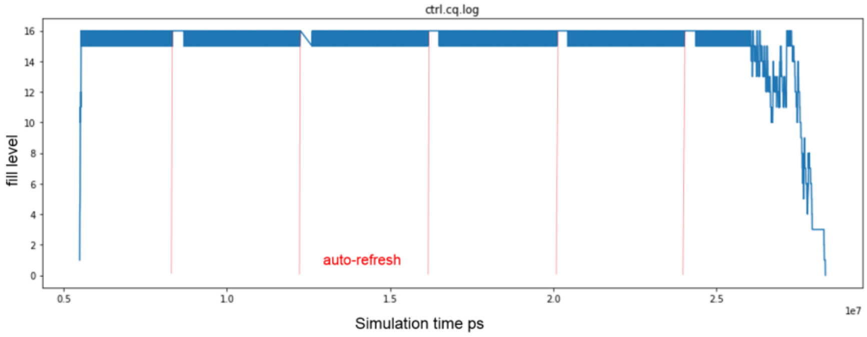


Figure 6 - Fill level trace of command reordering queue

CONCLUSION

We presented a framework for performance modeling of state-of-the-art memory controllers targeting various flavors of DRAM such as DDR3/4, and LPDDR3/4 using SystemC and TLM2.0. The design is based on a data flow concept. It facilitates carefully selected TLM extensions to adapt to advanced multi-channel bus protocols and at the same time remain standard compliant. The models are equipped with software interfaces equivalent to their RTL references and can be parametrized using an interface for Configuration, Control and Inspection. Functional verification and timing verification was done in TLM/RTL co-simulation, using a TLM test bench and appropriate TLM-to-RTL transactors. Timing accuracy in terms of simulated time (throughput) and average transaction latency for most of our tests is within a pessimistic corridor of 0-15% (TLM slower than RTL). The TLM simulates approximately 100x faster than the RTL design. This speed up and the analysis instrumentation of relevant design components greatly helps identifying architectural bottlenecks, saving a lot of time compared to waveform-based analysis and debug at RTL.

References

[1] JEDEC, „Joint Electron Device Engineering Council,“ 2018. [Online]. Available: www.jedec.org.

[2] J. Bruce, S. W. Ng und D. T. Wang, Memory Systems, Cache, DRAM, Disk, Burlington: Morgan Kaufmann, 2008.

[3] E. Cheung, H. Hsieh und F. Balarin, „Memory subsystem simulation in software TLM/T models,“ in *Asia and South Pacific Design Automation Conference*, Yokohama, 2009.

[4] V. Todorov, D. Mueller-Gritschneder, H. Reinig und U. Schlichtmann, „Automated construction of a cycle-approximate transaction level model of a memory controller,“ *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1066-1071, 2012.

[5] ARM Inc., „User Manual Carbon Model Studio 8.1,“ 2016. [Online]. Available: infocenter.arm.com.

[6] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch and A. N. Udipi, "Simulating DRAM controllers for future system architecture exploration," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, 2014.

[7] accellera systems initiative, „SystemC/TLM2.0,“ 2018. [Online]. Available: www.accellera.org/downloads/standards/systemc.

[8] Accellera, „CCI draft standard - Requirement specification for configuration interfaces,“ December 2009. [Online]. Available: http://accellera.org/images/downloads/drafts-review/configuration_requirements-091218.pdf.

[9] S. Biswas und H. Chen, „Reordering in the memory controller“. US Patent US8510521B2, 16 09 2010.

[10] S. Shrader, W. Bishop und A. Matta, „Method and apparatus for multi-port memory controller“. US Patent US7054968B2, 16 09 2003.