MCXplore: An Automated Framework for Validating Memory Controller Designs

Mohamed Hassan and Hiren Patel
{mohamed.hassan, hiren.patel}@uwaterloo.ca
University of Waterloo, Waterloo, Canada

***Abstract*—This work presents an automated framework for the validation of dynamic random access memory controllers (DRAM MCs) called *MCXplore*. In developing this framework, we construct formal models for memory requests interrelation and DRAM command interaction. The framework enables validation engineers to define their test plans precisely as temporal logic specifications. We use the NuSMV model-checker to generate counter-examples that serve as test templates; hence, *MCXplore* uses these test templates to generate memory tests to validate the correctness properties of the memory controller. We show the effectiveness of *MCXplore* by validating various state-of-the-art MC features as well as hard-to-detect timing violations that often occur. We also provide a set of predefined test plans, and regression tests that validate essential properties of modern DRAM MCs. We release *MCXplore* as an open-source framework to allow validation engineers and researchers to extend and use.**

I. Introduction

While the complexity of computing systems is increasing, their time-to-market is decreasing. As a consequence, the validation process of these systems becomes a major challenge that consumes a considerable portion of the design cycle. Companies spend millions of dollars annually on the validation process of all components of the computing system [1]. Researchers have proposed methodologies to validate CPU designs [2]. However, with the increase in memory requirement demands from applications, main memory subsystem is a vital component in almost all computing systems. Therefore, the validation of the memory subsystem is as crucial as validating other components; thus, it is the focus of this paper.

There are many techniques to validate computing systems. We target simulation-based validation since it is the most commonly used technique nowadays [2]. To validate any new feature or debug failures in the memory subsystem using the simulation-based approach, validation engineers adopt a simulation model. They provide stimulus inputs to the model and study its responses. Consequently, the effectiveness of this approach is heavily dependent on the ability of input tests to cover necessary execution scenarios to be validated. Different approaches exist for generating these tests. The straightforward approach is to use available benchmarks as the input stimuli, which saves time and cost required to develop test suites. This approach is extensively used by researchers to evaluate and validate their novel memory controller (MC) designs, though it has many shortcomings. First, some of the benchmarks may not be memory intensive. Furthermore, they may be so complex that they do not have easy-to-analyse memory patterns, which are vital to diagnose MC responses and to check for correctness. Second, these benchmarks do not explore the state space of the memory subsystem properties. For instance, they have specific locality and read/write switching percentages. However, exploring this state space is paramount for validating the design under all possible scenarios. To avoid these shortcomings, validation engineers either manually develop their own synthetic test suites or use random test generators [1] [2]. Manually-generated tests are time consuming and prone to human errors. On the other hand, randomly-generated tests may not cover all necessary test properties. In addition, MC designs are becoming complex with different performance optimizations such as multiple reordering levels, adaptive policies and priority-based arbitration. Therefore, test generation for memory subsystem validation is becoming an increasing challenge.

Contributions– We address this challenge by making the following contributions. (1) We present *MCXplore*, an automated framework for the validation of MCs. *MCXplore* enables validation engineers to precisely specify the properties required in the test suite in temporal logic specifications. Then, it automatically generates tests with the optimal number of memory requests that satisfy these properties to validate the correctness of the MC. To our knowledge, this is the first effort to automate the validation process of MCs. We release *MCXplore* as an open-source framework [3] to allow validation engineers and researchers to extend and use. (2) We introduce two formal models for the generation process of memory tests. The first model represents the interrelation amongst memory requests, while the second model resembles interactions between memory commands. These models allow us to encode the test generation process as a symbolic finite state machine (FSM), and use model checking techniques [4] to explore the state space for MC test suites and generate counter-examples that serve as test templates. *MCXplore* uses these test templates to generate property-driven test suites. (3) We highlight interesting sequence patterns that a test suite should encompass to test and evaluate various MC features. Consequently, we provide a set of predefined test plans as well as regression tests that validate essential functionalities of modern dynamic random access memory (DRAM) MCs. (4) Finally, we show case studies on applying our automated framework to validate the correctness of several state-of-the-art MC features and debug for any timing violations.

A. Main Memory Background

As Figure 1 illustrates, a DRAM is a three-dimensional array of memory cells arranged as *banks*. Cells in each bank are organized in *rows* and *columns*. A DRAM *rank* is a group of banks. For multi-channel DRAMs, each *channel* has its own buses and consists of one or more ranks. Accesses to different

978-3-9815370-6-2/DATE16/

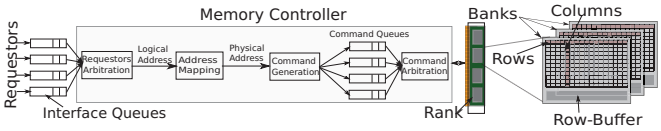


Fig. 1: DRAM subsystem.

channels, ranks or banks can be interleaved to reduce their access latency. On the other hand, accesses to different rows in the same bank suffer from *row conflicts* and encounter larger latencies. The *row buffer* caches the most-recently accessed row in each bank. DRAM accesses are controlled by the MC, which implements an arbitration scheme, an address mapping, and a page policy. The arbitration scheme arbitrates amongst different requests. The address mapping translates request addresses into 5 segments: channel (CH), rank (RNK), bank (BNK), row (RW), and column (CL). The page policy controls the liveness of the row in the row buffer. Open-page policy keeps the row in the row buffer until another row is requested. Contrarily, close-page policy writes back data in the row buffer after each access. Usually, modern MCs implement neither a strict open- nor close-page; instead, they implement adaptive policies that dynamically switch between the two. Finally, the MC issues one or more of the following commands to access the DRAM: ACTIVATE (A), READ (R), WRITE (W), PRECHARGE (P), and REFRESH (REF). A fetches the row to the row buffer. R (W) reads (writes) the required columns in the row buffer. P writes back the data in the row buffer to the corresponding row of cells. Finally, REF activates and precharges DRAM rows to prevent charge leakage. The DRAM JEDEC standard [5] imposes strict timing constraints on these commands (Table I). All MC designs must satisfy these constraints to ensure correct DRAM behaviour.

TABLE I: Important JEDEC timing constraints (DDR3-1333) [5].

Const.	Meaning	Cyc.
tRC	Minimum time between A commands to same bank.	34
$tCCD$	Column-to-column delay.	4
tRP	Row pre-charge time	10
$tBUS$	$\frac{\text{request size}}{\text{data bus size} \times 2}$: Time required to transfer a data burst.	4
$tRAS$	Minimum time between A command and P command.	24
tWL	Minimum time between W and the start of data transfer.	9
tRL	Minimum time between R and the start of data transfer.	10
$tRCD$	Minimum time between activating the row and accessing it.	10
$tFAW$	Four bank activation window in same rank.	20
$tRTRS$	Rank to Rank switching delay.	1
$tRTP$	Read to precharge delay.	5
$tWTR$	Write to read switching delay.	5
tWR	Write recovery delay.	10
$RKtoRK$	$(tBUS + tRTRS)$: Rank switching delay.	
$RtoW$	$(tRL + tBUS + tRTRS - tWL)$: R to W delay.	
$WtoR\_B$	$(tWL + tBUS + tWTR)$: W to R in same rank delay.	
$WtoR\_RK$	$(tWL + tBUS + tRTRS - tRL)$: W to R in different ranks delay.	
$RtoP$	$(tBUS + tRTP - tCCD)$: R to P delay.	
$WtoP$	$(tWL + tBUS + tWR)$: R to P delay.	

II. Related Work

Researchers have proposed several novel features to reduce the large DRAM access latency. These efforts include providing simulation environments to help in the process of evaluating new ideas [6], proposing new features in all memory controller subcomponents such as address mapping [7], page policy [8] and arbitration [9]. However, researchers usually validate their novel features using benchmarks such as in [8], or manually-written directed tests or a combination of both such as in [7], [9]. We propose an automated process of validating new features in the DRAM subsystem that can be used both by researchers and industry.

In industry, our automated framework can be used in the pre-silicon validation of MCs. Pre-silicon validation engineers often use hand-written directed tests or randomly-generated tests [1]. Compared to both methods, our proposed framework: would achieve better coverage, is less error-prone and reduces validation complexity through automation.

Model checking has proven its success as a test-generation engine for validating both software [10] and hardware [2]. This work is the first to incorporate model checking techniques in the test generation process for the memory subsystem.

III. MCXplore

Figure 2 represents the steps of our methodology. The process consists of three phases: test template generation, test suite generation and diagnosis and reporting results. Thus, the process separates the test generation step from the test plan step. This is an important requirement from validation engineers to simplify the validation process [11].

Phase** 1**: Test Template Generation– In this phase, *MCXplore* turns the test plan into a test template in three steps.

Step 1: A test plan is a list of behaviours whose correctness needs to be validated. Usually, design engineers provide this list in a highly-abstracted human language.

Step 2: The big challenge for validation engineers is to turn the test plan into meticulous rules that generated tests must follow [12]. We promote leveraging model checking capabilities to address this challenge. Model checking automates the state-space exploration of the test generation, and provides a formal methodology to define test properties. We create two abstract models to express the stimulus test of the MC: a request model and a command interaction model, and we encode them as FSMs in the NuSMV model checker [13]. Accordingly, validation engineers are able to encode test properties as specifications expressed in temporal logic formulas. Formulas are negated such that they are true if required test properties do not exist. We accompany *MCXplore* with regression suites and a pre-defined set of temporal logic specifications that encode most of the basic test properties required to stress MC designs. Table II tabulates these properties.

Step 3: The model checker explores the FSM to determine the truth or falsity of the specifications. For a false specification, it constructs a counter-example, which is a trace of states that falsifies the specification. This trace represents the test template that encompasses test properties specified by validation engineers. We use bounded model checking to obtain the trace with minimum number of states, which results in tests with the optimal (minimum) number of memory requests satisfying specified properties. Minimizing the number of requests is mandatory to reduce the time and complexity of the validation process.

Phase** 2**: Test Suite Generation– *Step 4:* We provide a parser script to parse the test template produced by phase 1 and generate test suites with actual memory requests. Validation engineers drive this parser with the address mapping of the

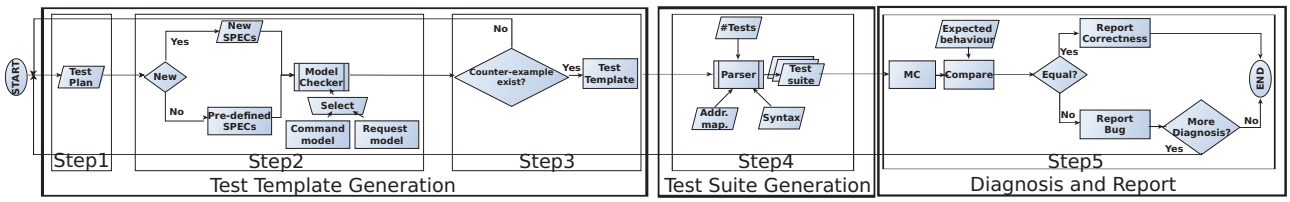


Fig. 2: Proposed pre-silicon validation process of MCs.

Property	Configuration
Address pattern	Linear Random Customized
Type pattern	All reads All writes Switching Random Switching% Reads-to-writes%
Address mapping	Any
Transaction size	Any
Address length	32

Customized Address	
Segment	Configuration
Row	Hit (for any row) Conflict Random Locality%
Bank/ Rank/ Channel	No interleave Fully interleave Random Interleave%
Column	Same Successive Random

TABLE II: Currently pre-defined configurations.

MC, number of desired tests in the suite, and test syntax.

***Phase** 3**: Diagnosis and Report–** Step 5:* Validation engineers invoke the MC under validation with the generated test suite and compare responses with the expected behaviour. If results match, then they report correctness. Otherwise, they report their diagnosis results and conduct more detailed investigation if required. Section IV provides lemmas and a methodology to construct test plans with certain expected behaviours to use as a golden metric to compare the MC response.

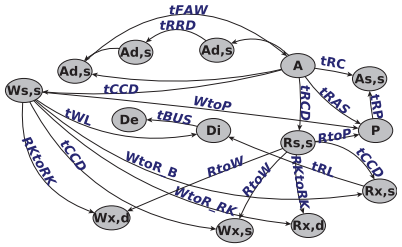
A. Proposed Models

We propose two models that are at different granularities to facilitate the test generation process.

Request Interrelation Model – To fully cover the state-space of a test of n memory read/write requests and a 32 bit address, $2^{33} \times n$ tests are needed. Clearly, such a large number of tests is prohibitively time consuming. We argue that the important factor in the coverage is not the input stream pattern. Instead, it is the MC's response to this stream. For instance, if a request to a row RW_1 is followed by a request to RW_2 in the same bank and rank, then the MC behaviour depends on whether $RW_2 = RW_1$ or not. This is regardless of the actual values of these rows. The same observation holds for banks, ranks and channels. Hence, a state graph constructing these relationships is sufficient to represent a model for the test template generation step. Based on this observation, we model the interrelation between memory requests as the Kripke structure in Model 1. Recall that a DRAM request has an address ($Addr$), and an operation type (ty) where the address consists of 5 segments (row, column, bank, etc.), and the type

Model 1 : Kripke structure for the MC input

$MCin = \{S_{in}, I_{in}, R_{in}, L_{in}\}$ where:
$P_{in} = \{ty, e_{rw}, e_{ch}, e_{rnk}, e_{bnk}, e_{cl}\}$
$S_{in} = \{s_i : \forall i \in [0, 63]\}$ is the set of all possible states.
$I = \{s_0\}$ is the set of initial states.
$R = \{(s_i, s_j) : \forall i, j \in [0, 63]\}$ is the transition relation between states.
$L = \{(si, \langle e_{cl}, e_{bnk}, e_{rnk}, e_{ch}, e_{rw}, ty\rangle)\}$ is the labelling function where all the sets cannot be empty sets, and
ty=BIN$(i, 0)$, e_{rw}=BIN$(i, 1)$, e_{ch}=BIN$(i, 2)$, e_{rnk}=BIN$(i, 3)$, e_{bnk}=BIN$(i, 4)$, and e_{cl}=BIN$(i, 5)$.



Subscripts reflect the targeted bank and rank respectively. d: different, s: same, x: do not care. Di: start of the data transfer. De: end of the data transfer. P is for same bank.

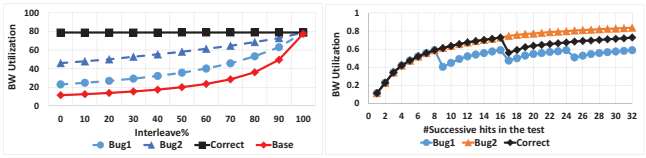
Fig. 3: DRAM commands and timing constraints interaction.

is a read or a write operation. We define the proposition e for each address segment such that $e = 1$ means that the request has the same segment as its previous request, and $e = 0$ otherwise. Similarly, if $ty = 1$, then the operation is a read, and a write otherwise. To exhibit all possible relations between successive requests, we have 64 possible states. For instance, for state s_{39}, $(s_{39}, \langle 1, 0, 0, 1, 1, 1\rangle)$ denotes a read request that targets the same channel, row and column as its previous request, while it targets different rank and bank. We also maintain a set of counters to track the address pattern such as total number of requests, row hits, and bank interleavings, which we use to encode the test specifications. Note that BIN(x, y) returns the y^{th} bit of a positive integer x's binary equivalent number.

Command Interaction Model– Validation engineers can use the request interrelation model to validate properties related to timing constraints ruling command interactions. However, in this case, *MCXplore* requires them to find out the request patterns that expose these timing constraints. This is because, using the request interrelation model, *MCXplore* allows specifications to be at the request level and not the command level. Therefore, we propose the command interaction model to facilitate the validation of properties related to MC command generation. This model enables validation engineers to specify the timing constraints to be validated and *MCXplore* automatically generates the test sequence that exercises these constraints. Figure 3 depicts the state graph of this model that we build based on the timing constraints imposed by the JEDEC standard [5]. The vertices represent DRAM commands and the edges represent timing constraints. For example, the time between A and a P to the same bank must be at least $tRAS$. In section IV-C, we use this model to generate test suites for validating the correctness of command generation, and checking for any timing violations.

IV. Evaluation: Validating state-of-the-art MC features

We use DRAMSim2 [6] with DDR3-1333 DRAM to validate several state-of-the-art MC features. We also insert



(a) XOR address mapping. (b) Maximum hits using utilization.

Fig. 4: Address mapping and page policy validation.

common design bugs in the functionality of these features to determine whether the proposed methodology can discover them. Although we have carried out these evaluations for an extensive number of features, due to space limitations, we only show a subset of them. We use bandwidth utilization defined as ($Uti = \frac{\text{Data transfer cycles}}{\text{Total DRAM access cycles}}$) as our metric to validate the features. The advantage of using Uti for validation is that it does not require an engineer to observe internals of the MC. Instead, existing inputs and outputs of the MC are sufficient.

A. XOR Address Mapping Validation

Modern MCs reduce row conflicts by using XOR address mapping where the bank bits are bitwise XOR-ed with the least significant three row bits [7].

Test plan. To generate a test suite $Suite_{XOR}$ that represents the optimal memory pattern for the XOR mapping. It is a stream of read accesses where we change the bank interleaving ratio per test, $intr$. In addition, requests targeting the same bank are accessing different rows. Since the plan is related to the interrelation between requests, we use the request model for test generation.

Specifications. Each test has its corresponding specification. The following formula encodes a test plan with $intr = 40\%$, where t_x represents the total counts of the event x:

$$\text{LTLSPEC G}((t\_requests = 6 \ \wedge \ t\_hit = 0 \ \wedge t\_bank\_interleave = 0)) \rightarrow \text{!F}(t\_requests = 10 \ \wedge \ t\_hit = 4 \ \wedge \ t\_bank\_interleave = 4))$$

The intuition behind the specification is that out of 10 total requests in the test, the first 6 requests target different rows in the same bank, while the last 4 requests target the same row but in different banks.

Test template. Step 3 in *MCXplore* produces a counter-example for each specification, which forms the test template that we formalize as below. Each test has an interleaving percentage between 0% and 100%. $nbnk$ is the number of banks per rank (usually 8). The conditions ensure that $intr\%$ of requests in the test interleave across different $nbnk$ banks. They also ensure that in these $intr\%$ requests, each $nbnk$ successive requests target same row, which implies that requests targeting different banks have same rw segment, while requests to the same bank have different rw values. Again, the target of this test plan is to achieve the maximum possible utilization of XOR mapping regardless of the $intr$ value.

$Suite_{XOR} = \{Test_{intr} : \forall intr \in [0, 100]\}$
$Test_{intr} = [Req_1, Req_2, ..., Req_n]$, where $Req_k = \langle Addr_k, R\rangle, k \in [1, n]$ and $((rw_l = rw_m)$ iff $((l \text{ MOD } nbnk = m \text{ MOD } nbnk) \wedge (l, m \in [1, \frac{n \times intr}{100}]))$, and $((bnk_l \neq bnk_{l-1})$ iff $l \in [2, \frac{n \times intr}{100}]))$.

Test suite. *MCXplore* parses each test template and generates a test that complies with the test plan (step 4).

Validation. For the sake of comparison, we execute $Suite_{XOR}$ on both the XOR mapping and the base mapping (no XOR operation is performed). As Figure 4a illustrates, increasing the $intr$ ratio on the test, the base mapping achieves better utilization. This is because requests to different banks are serviced in parallel. On the other hand, the correct behaviour of the XOR mapping is to achieve a fixed utilization for all tests in the suite. This is because even for non-interleaved accesses, the XOR address mapping will map them to different banks because of the XOR operation between the bank bits and the corresponding row bits. To further check for correct functionality, this value should be compared to the expected utilization dictated in Lemma 1. Figure 4a shows that the XOR mapping achieves a fixed utilization of 79%, which coincides with the expected behaviour.

Lemma 1: Executing any test in $Suite_{XOR}$ on an MC with XOR mapping results in a utilization that can be calculated as: $\frac{4tBUS}{tFAW}$.

Proof: Since XOR mapping maps successive requests of any test in $Suite_{XOR}$ to different banks, the MC under test repeats the behaviour shown in Figure 5 every 8 requests. Focusing on one repetition, the data bus is busy for $8tBUS$, while the total DRAM latency is $2tFAW$. ■

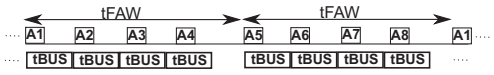


Fig. 5: Command sequence of $Suite_{XOR}$ on XOR mapping.

Bug scenario. To illustrate potential design errors, we inject two bugs in the XOR mapping. In the first one (Bug1 in Figure 4a), we perform the XOR operation between only the first two bits of the bank and row segments, while in the second bug (Bug2), we perform the XOR operation between the least significant bit of row and bank segments. From Figure 4a, both Bug1 and Bug2 do not achieve the expected utilization of Lemma 1; hence, they are detectable.

B. Page Policy and Arbitration Validation

In this section, we validate an MC feature that affects both the page policy and the arbitration deployed by the MC. MCs employing this feature keep the row in the row buffer for a designated number of row hits, that we call maximum row-hits threshold. Thus, the open-page policy is turned into a threshold-based page policy in these MCs. In addition, the threshold limits the number of requests that can be reordered with the first-ready first-come-first-serve (FR-FCFS) arbitration scheme deployed in most conventional MCs nowadays [9]. MC designers select the threshold value that maximizes the performance for targeted applications. We assume the intended threshold to be $thr = 16$.

Test plan. To generate a set of tests, where each test is a stream of read accesses targeting the same bank, while we sweep the number of requests targeting an open row (row hits), hit, per test. We use the request model to generate the test suites.

Specifications. The following formula exemplifies the encoding of the test plan with $hit = 16$, where c_x represents the total successive occurrences of x.

$$\text{LTLSPEC G}(c\_hit = 16 \mapsto \text{!}F(t\_requests = 34 \wedge t\_hit = 33 \wedge t\_bank\_interleave = 0 \wedge c\_hit = 16))$$

Test template. Step 3 in *MCXplore* produces a counter-example

for each specification that we formalize as follows, where we sweep hit between 0 and 32. The conditions ensure that all requests target the same bank, while every hit successive requests target the same row.

$Suite_{thr} = \{Test_{hit} : \forall hit \in [0, 32]\}$
$Test_{hit} = [Req_1, Req_2, ..., Req_n]$, where $((bnk_l = bnk_{l-1})$ and $((rw_l = rw_m)$ iff $(l \text{ MOD } hit = m \text{ MOD } hit)) \ \forall l, m \in [1, n])$.

Validation. We execute the generated tests and compare with the expected behaviour. The correct functionality is to achieve the maximum utilization when $hit = thr$. Lemma 2 calculates this maximum utilization value. Figure 4b shows that the MC under correct functionality (Correct) achieves a maximum utilization of 73% at $hit = 16$, which confirms the conclusion of Lemma 2.

Lemma 2: Executing $Suite_{thr}$ on an MC that implements a maximum row-hits threshold results in a maximum utilization for the test $Test_{hit}$ with $hit = thr$ and this utilization can be calculated as: $\frac{thr \times tBUS}{tRCD+(thr-1)tCCD+RtoP+tRP}$.

Proof: When $hit = thr$, the DRAM under test repeats the behaviour illustrated in Figure 6 every thr requests. During one repetition, the data bus is busy for $thr \times tBUS$, while the total access latency is $tRCD + (thr - 1)tCCD + RtoP + tRP$. ■

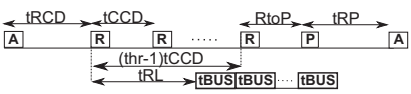


Fig. 6: Command sequence of $Suite_{thr}$ when $hit = thr - 1$.

Bug scenario. We embed two bugs to the logic of the row-hits threshold. The first bug (Bug1 in Figure 4b) reduces the threshold to 8 instead of intended value by the designer (16), while the second bug (Bug2 in Figure 4b) increases the threshold to 32. From utilization graphs in Figure 4b, we directly discover that the maximum utilization value is not the expected value calculated by Lemma 2. In Bug1, the utilization graph repeats a pattern every multiple of 8, where it achieves the maximum utilization. Consequently, we deduce that the bug causes the threshold to be 8. Similar conclusion can be reached for Bug2.

C. Timing Parameters Validation

Using *MCXplore*, we design property-driven tests in order to validate the correctness of the timing parameter values enforced by the MC. The key novelty here is that each test is designed to maximize the impact of the timing parameter under test while eliminating or minimizing the effect of all other parameters. Using the state graph in Figure 3, we exhaustively study all possible command interactions, produce utilization equations to investigate the impact of timing parameters on utilization. Having these equations, we find that not all parameters can be isolated. As a consequence, we introduce the dependency graph in Figure 8. An edge from constraint $constr_1$ to $constr_2$ means that $constr_1$ must be validated before $constr_2$. A bi-directional edge between two constraints means that they have to be validated together. For space limitation reasons, we show the validation process of only a subset of the parameters and summarize our findings in Table IV.

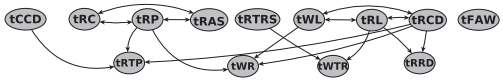


Fig. 8: Validation dependency graph for timing parameters.

Bug scenarios. For the timing parameter under validation, we randomly set one of these parameters to a wrong value in the range: $[0, standard\ value + 20]$, where *standard value* is the value dictated by the JEDEC standard.

1) tRTP: ***Test plan.*** Studying the state graph in Figure 3, a valid command sequence encompassing $tRTP$ would be an A command followed by one or more R commands then a P command to close the row followed by an A to a different row. In addition, the number of R commands must be large enough to dominate the $tRAS$ constraint between A and P. Clearly, this example highlights the importance of the command model. Using the request model, validation engineers have to manually design the request sequence that exposes these details. On the other hand, the command model captures the command interaction details; thus, allows validation engineers to set the specifications as simple as we illustrate below.
Specifications. Specification is as follows, where num_tRTP is the number of occurrences of the $tRTP$ constraint:
LTLSPEC G! $(num\_tRTP \geq 1)$
Test template. Figure 9 delineates the command sequence generated by *MCXplore*. From Figure 9, the $tRCD$, tRP and $tCCD$ parameters must be be validated before $tRTP$, which coincides with the dependency in Figure 8.
Test suite. *MCXplore* parses this command sequence and creates a test, $Test_{RTP}$, consists of five read accesses targeting the same bank where the last request targets a different row than the first four.
Validation. To validate $tRTP$, we compare the observed utilization (Uti_o) from executing $Test_{RTP}$ with the calculated utilization (Uti_c) from Lemma 3.

Lemma 3: Executing $Test_{RTP}$, the BW utilization of the MC under test is: $\frac{4tBUS}{tRCD+3tCCD+tBUS+tRTP+tRP}$.

Proof: Executing $Test_{RTP}$, the MC under test repeats the behaviour shown in Figure 9 every 4 requests. Focusing on one repetition, the data bus is busy for $4 \times tBUS$ cycles. In addition, the 4 requests encounter a total DRAM access latency of $tRCD + 3tCCD + tBUS + tRTP + tRP$. ■

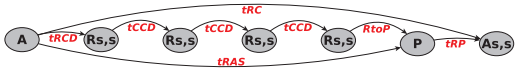
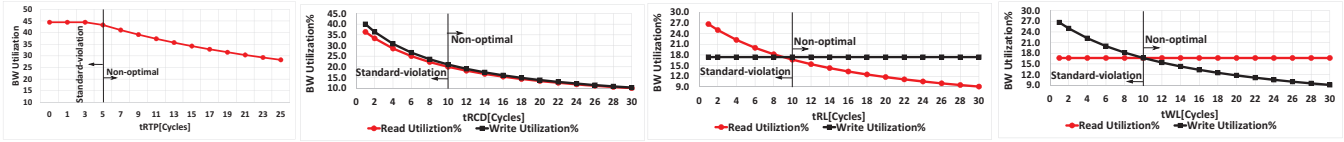


Fig. 9: Command sequence of $Test_{RTP}$.

Based on the comparison, we make the following conclusions:

$Uti_o = Uti_c$	optimal value	Figure 7a at $tRTP = 5$
$Uti_o > Uti_c$	violated	Figure 7a at $tRTP < 5$
$Uti_o < Uti_c$	non-optimal value	Figure 7a at $tRTP > 5$

2) tRCD,tWL,tRL: ***Test plan.*** The target is to validate tRL and tWL parameters, which requires two tests.
Test template. *MCXplore* generates a template for the tRL test as an A followed by a R. Similarly, the template of the tWL



(a) tRTP (b) tRCD (c) tRL (d) tWL

Fig. 7: Timing parameters validation.

$Test_{\{tRCD,tRL\}}$	$Test_{\{tRCD,tWL\}}$	Conclusion	Figure
$Uti_o > Uti_c$	$Uti_o > Uti_c$	$tRCD$ is violated.	7b
$Uti_o < Uti_c$	$Uti_o < Uti_c$	$tRCD$ is not optimal.	
$Uti_o > Uti_c$	$Uti_o = Uti_c$	tRL is violated.	7c
$Uti_o < Uti_c$	$Uti_o = Uti_c$	tRL is not optimal.	
$Uti_o = Uti_c$	$Uti_o > Uti_c$	tWL is violated.	7d
$Uti_o = Uti_c$	$Uti_o < Uti_c$	tWL is not optimal.	

TABLE III: Validating $tRCD$,tRL and tWL.

test is an A followed by a W. Hence, it is not possible to exclude the $tRCD$ parameter (between A and R or W). As a consequence, we validate $tRCD, tRL$ and tWL together.

Test suite. $Test_{\{tRCD,tRL\}}$ is a single read request, while $Test_{\{tRCD,tWL\}}$ is a single write request.

Validation. To validate the parameters $tRCD, tRL$ and tWL, we investigate the utilization observed (Uti_o) from running tests $Test_{\{tRCD,tRL\}}$ and $Test_{\{tRCD,tWL\}}$. If the observed utilization coincides with the calculated utilization (Uti_c) in Lemma 4 for both tests, then all the three parameters are set to the standard value.

Lemma 4: Executing $Test_{\{tRCD,tRL\}}$, the utilization of the MC under test is: $\frac{tBUS}{tRCD+tRL+tBUS}$. similarly, executing $Test_{\{tRCD,tWL\}}$, the BW utilization of the MC under test is: $\frac{tBUS}{tRCD+tWL+tBUS}$.

For the DDR3 module used in our validation, this situation is observed in Figures 7b, 7c and 7d at $tRCD = 10$, $tRL = 10$ and $tWL = 9$. Table III summarizes our debugging conclusions from the utilization graphs. We assume a single parameter is possibly violated at a time.

3) Other parameters: Similar to the aforementioned two instances, for validating other parameters we conduct the following procedure. 1) We execute the corresponding test from Table IV. 2) We compare the observed utilization with the calculated utilization. 3) Based on the comparison, we determine whether the parameter under test is (a) compliant with the standard, (b) violated or (c) set to a non-optimal value. We tabulate calculated utilizations from all tests in Table IV. In Table IV, unless specified, the number of requests is $n \gg 10$.

Test	Conditions ($\forall l \in [1, n]$)	Utilization
tRC	$(bnk_l = bnk_{l-1}) \wedge (rw_l \neq rw_{l-1})$	$\frac{tBUS}{tRC}$
$tCCD$	$(bnk_l = bnk_{l-1}) \wedge (rw_l = rw_{l-1})$	$\frac{tBUS}{tCCD}$
$tFAW$	$(rnk_l = rnk_{l-1}) \wedge$ $(bnk_l \neq bnk_{l-1}) \wedge (rw_l \neq rw_{l-1})$	$\frac{8 \times tBUS}{2 \times tFAW}$
$tRTRS$	$(rnk_l \neq rnk_{l-1}) \wedge (rw_l = rw_{l-1})$	$\frac{tBUS}{tBUS+tRTRS}$
$tRRD$	$(n = 4) \wedge$ $(rnk_l = rnk_{l-1}) \wedge (bnk_l \neq bnk_{l-1})$	$\frac{4 \times tBUS}{3 \times tRRD+tRCD+tRL+tBUS}$
tWR	$(ty_l = W) \wedge$ $(bnk_l = bnk_{l-1}) \wedge (rw_l \neq rw_{l-1})$	$\frac{tBUS}{tRCD+tWL+tBUS+tWR+tRP}$
$tWTR$	$(ty_l \neq ty_{l-1}) \wedge$ $(bnk_l = bnk_{l-1}) \wedge (rw_l = rw_{l-1})$	$\frac{tBUS}{tRL+tWTR+tBUS+tRTRS}$

TABLE IV: Tests of timing parameters.

V. Conclusion

We propose a framework for validating MC designs. We introduce two models for the test input of the MC and enable validation engineers and researchers to specify their test plan as specifications in temporal logic. We use model checking to generate test templates that satisfy this plan. We implement this framework and release it open-source as *MCXplore*, accompanied with a regression test suite for validating basic MC features. Using *MCXplore*, we show how to validate the correctness of state-of-the-art MC features as well as discover timing violations in the DRAM subsystem.

References

[1] "Intel platform and component validation , a white paper," http://download.intel.com/design/chipsets/labtour/PVPT_WhitePaper.pdf, Intel, 2015-08-31.

[2] H.-M. Koo and P. Mishra, "Test generation using sat-based bounded model checking for validation of pipelined processors," in *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, 2006, pp. 362–365.

[3] "Mcxplore." [Online]. Available: https://caesr.uwaterloo.ca/mcxplore/

[4] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[5] "DDR3 SDRAM specification, JESD79," JEDEC, 2010.

[6] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[7] W.-F. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2001, pp. 301–312.

[8] M. Hassan, H. Patel, and R. Pellizoni, "A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems," in *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[9] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 416–427, 2012.

[10] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

[11] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "A unified methodology for pre-silicon verification and post-silicon validation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2011, pp. 1–6.

[12] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, p. 13, 2007.

[13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*. Springer, 2002, pp. 359–364.