MCXplore: Automating the Validation Process of DRAM Memory Controller Designs

Mohamed Hassan, *Student Member, IEEE*, and Hiren Patel, *Member, IEEE*

***Abstract*—We present an automated framework for the validation of memory controllers (MCs) called *MCXplore*. In developing this framework, we construct formal models for memory requests and command interactions. *MCXplore* enables validation engineers to define their test plans precisely using temporal logic specifications. We use the NuSMV model-checker to generate counterexamples that serve as test templates. *MCXplore* uses these test templates to generate memory tests to validate the correctness properties of the MC. We show the effectiveness of *MCXplore* by validating various state-of-the-art MC features as well as hard-to-detect timing violations. We also provide a set of predefined test plans, and regression test suites that validate essential properties of modern MCs. *MCXplore* is an open-source framework to allow validation engineers and researchers to extend and use.**

***Index Terms*—Dynamic random access memory (DRAM), memory controller (MC), model checking, testing, validation, verification.**

I. Introduction

WHILE the complexity of computing systems is increasing, their time-to-market is decreasing. As a consequence, the validation process of computing 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], [3]. 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, this paper focuses specifically on validating the memory controller (MC) component, which manages requests to the main memory.

There are several techniques to validate computing systems. We consider simulation-based validation since it is a commonly used approach [2], [3]. 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 MC designs [4]–[6]; though, it has shortcomings. First, some of the benchmarks may not be memory intensive. Furthermore, they may be so complex that they do not have easy-to-analyze 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 ratios. 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.

An MC is a complex component that has to track the state of all memory banks, check over 12 timing constraints obligated by the JEDEC standard [7], and make several dynamic arbitration decisions. Having the memory as a bottleneck in many computing systems [8], they are becoming even more 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.

A. Contributions

We address this challenge by making the following contributions.

1) We present *MCXplore*, an automated framework for the validation of MCs. Unlike prior efforts to validate MCs, *MCXplore* is design independent such that it can be used to validate any MC design. This is possible because *MCXplore* instead of following the convention by modeling a specific MC design, it models the input stimulus, which is common across different MC implementations. *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. We release *MCXplore* as an open-source framework [9] to allow validation engineers and researchers to extend and use.

Manuscript received August 17, 2016; revised January 8, 2017; accepted April 14, 2017. Date of publication May 18, 2017; date of current version April 19, 2018. This paper was recommended by Associate Editor L. Benini. *(Corresponding author: Mohamed Hassan.)*

The authors are with the University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: mohamed.hassan@uwaterloo.ca; hiren.patel@uwaterloo.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2705123

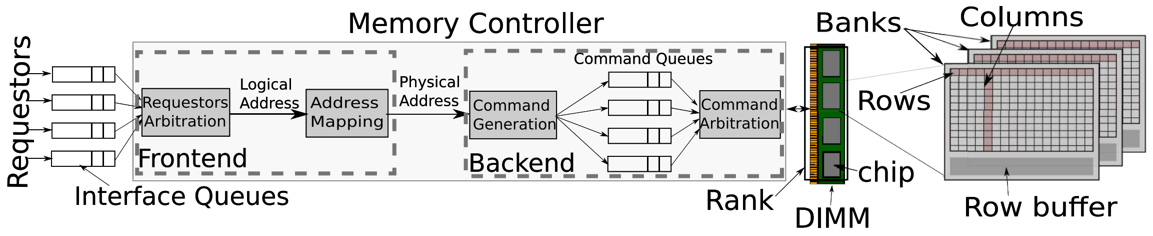


Fig. 1. DRAM subsystem.

2) We introduce two formal models for the generation process of memory tests. The first model represents the interrelation amongst memory requests (Section IV-A1) and is used to validate the MC's frontend (Section VI), while the second model resembles interactions between memory commands (Section IV-A2) and is suitable for validating the MC's backend (Section V). These models allow us to encode the test generation process as a symbolic finite state machine (FSM), and use model checking techniques to explore the state space for MC test suites and generate counterexamples 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. These test suites can be used to conduct trace-based validation for high-level models of MCs (such as DRAM simulators), and they can be utilized to construct test benches to validate hardware register-transfer level (RTL) implementations.
4) Finally, we show a methodology to use high-level statistics such as bandwidth utilization, access latency, and aggregated number of committed commands to validate the correctness of several state-of-the-art MC features and debug for any timing violations. The validated MC features are just examples to show the capabilities of using *MCXplore* for validation. The proposed methodology applies for other features as well.

II. Background

We introduce basics of DRAM, MCs, and model checking that are necessary for this paper.

A. Main Memory

As Fig. 1 illustrates, a DRAM is organized in dual inline memory modules (DIMMs), each DIMM consists of multiple DRAM chips. Each DRAM chip consists 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 multichannel DRAMs, each *channel* has its own buses and consists of one or more ranks. Accesses to different 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. Data is transferred to/from the memory cells via sense amplifiers. These sense amplifiers work as a *row buffer* that caches the most-recently accessed row in each bank. A DRAM request consists of a type and an address. The type is either a read or a write. DRAM accesses are controlled by the MC, which translates the read/write requests into one or more of the following DRAM commands: ACTIVATE (A), READ (R), WRITE (W), PRECHARGE (P), and REFRESH (REF). A fetches the row from the memory cells to the sense amplifiers (row buffer). R (W) reads (writes) the required columns in the row buffer. P closes the activated row, and prepares the cell array for the next memory access by restoring the charge level of each DRAM cell in the row. Finally, REF activates and precharges DRAM rows to prevent charge leakage. The DRAM JEDEC standard [7] imposes strict timing constraints on these commands (Table I). All MC designs must satisfy these constraints to ensure correct DRAM behavior. Typically, an MC 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 five segments: 1) channel (*ch*); 2) rank (*rnk*); 3) bank (*bnk*); 4) row (*rw*); and 5) 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 precharges 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.

TABLE I
Important JEDEC Timing Constraints (DDR3-1333) [7]

Const.	Meaning	Cyc.
tRC	Minimum time between A commands to same bank.	33
$tCCD$	Column-to-column delay.	4
tRP	Row pre-charge time	9
$tBUS$	$\frac{\text{request size}}{\text{data bus size} \times 2}$: 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.	7
tRL	Minimum time between R and the start of data transfer.	9
$tRCD$	Minimum time between activating the row and accessing it.	9
$tFAW$	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
$tREFI$	Refresh Period.	$7.8\mu s$
$tRFC$	Time required to refresh.	$160ns$
$RKtoRK$	$(tBUS + tRTRS)$: 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.	

B. Model Checking

Fig. 2 delineates the basic operation of a model checker. A model checker is a verification tool that takes two

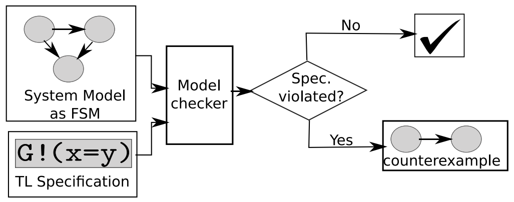


Fig. 2. Model checking operation.

inputs: 1) a system model expressed as an automaton or an FSM and 2) a property expressed in a temporal logic statement. It, then, checks whether the system violates this property or not by exploring the state space of the model. If it detects a violation, it produces a counterexample. This counterexample is a path of states that falsifies the checked property. If the property is carefully crafted, this counterexample can be interpreted as a test case for the system [10]. Since the state space of the system may be exponential, it is useful to bound the number of searchable states by the checker. This approach is known as bounded model checking (BMC) [11]. In BMC, the checker searches for a counterexample in executions whose length is bounded by some integer i. Leveraging BMC, *MCXplore* provides memory tests with optimal (minimum) number of requests. Optimality is achieved by starting with $i = 1$ and increasing i until a counterexample is found. We provide more details about test generation in Section IV.

III. Related Work

Researchers have proposed several novel features to reduce the large DRAM access latency. We divide these efforts into two main categories. The first category is providing DRAM simulation environments to help in the process of evaluating the strengths and weaknesses of new ideas. Examples include DRAMsim2 [12], USIMM [13], DrSim [14], Ramulator [15], and DRAMSys [16]. The second category is proposing novel features in all MC subcomponents, such as address mapping [4], [5], page policy [17], and arbitration [18]–[20]. Validating DRAM MC designs or simulators is a challenging task for many reasons.

1) The MC must carefully track the state of each DRAM bank.
2) The standard specifies more than 12 timing constraints that any MC must satisfy for correct operation.
3) Many dynamic factors impact the MC operation such as the type of requests in the queues and which DRAM cell their addresses target.

For these reasons, most of the proposed DRAM simulators do not fully validate their operation. Ramulator [15] is validated using ten million memory requests that are a mix of random and hand-crafted requests. The authors conduct the validation process by providing these requests to both Ramualtor and a reference Verilog model provided by Micron [21], and comparing the behavior of both. This approach has many shortcomings.

1) The used requests do not necessarily exhibit all the possible request and command interactions. Thus, validation coverage is not guaranteed.
2) The adopted Verilog reference model only validates that there are no violations in the timing parameters. A timing parameter is violated if a corresponding command is issued earlier than the specified timing constraint. However, it does not test for performance penalties that may result from issuing the commands later than the specified constraint.
3) It does not validate all other policies at the MC's frontend, because these policies are not directly related to command generation.

These policies include the page policy, address mapping, and request arbitration. Developers of DRAMsim2 [12] follow a similar validation procedure; thus, they suffer from the same aforementioned shortcomings. DRAMSys [16] also uses a similar approach. It executes testing scripts on the generated commands to check if the timing constraints comply to the JEDEC standard. To our knowledge, neither USIMM [13] nor DrSim [14] are validated. *MCXplore* provides an easy-to-adopt methodology to validate these simulators to avoid these shortcomings. First, it provides a methodology to generate property-driven tests for each policy that can cover all possible behaviors, as we illustrate in Section IV. Second, the methodology can be used to seamlessly validate both backend and frontend policies of MCs, as we illustrate in Section VI. We intentionally insert bugs into several components of DRAMsim2 [12] and show that *MCXplore* is able to detect those bugs.

Upon proposing novel MC's policies or features, researchers usually validate them using benchmarks such as in [4], [5], and [22], or manually written directed tests or a combination of both such as in [17], [18], and [20]. Similarly, in industry, presilicon validation engineers often use handwritten directed tests or randomly generated tests [1], [23]. We propose an automated portable process of validating new features in the DRAM subsystem that can be used both by researchers and industry. Compared to these methods, our proposed framework would achieve better coverage, is less error-prone, and reduces validation complexity through automation.

A. Formal Verification of the Memory system

Some prior works incorporate formal models in the design and validation process of memory systems [24]–[27]. Authors of MSimDRAM [24] model their MC design as a state machine. Accordingly, they encode the correct intended behavior of that MC as linear temporal logic (LTL) specifications and use BMC to verify this behavior. Khalifa and Salah [25] used a universal verification methodology (UVM) environment to verify their proposed generic MC. The environment consists of a test driver that generates test cases stimulating both the design and a reference transaction level modeling (TLM) model. The environment compares results from the design with the reference results to detect any faults. The methodology in [26] automatically transforms the timing constraints from the JEDEC standards into system Verilog assertions, which can be used to verify that the commands generated by the MC comply with these constraints. A timed automata model for the MC's backend is proposed in [27]. It uses UPPAAL model checker to verify the correct behavior of

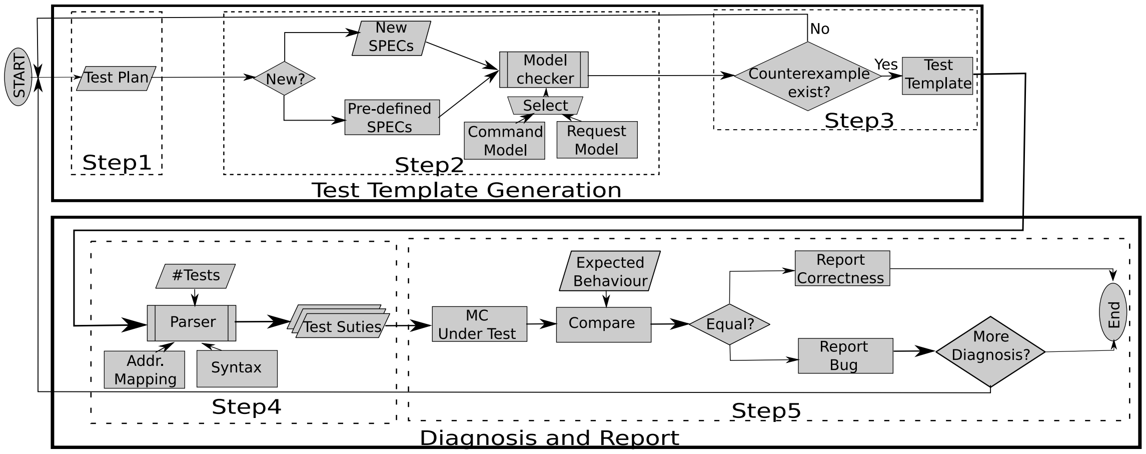


Fig. 3. Proposed validation process of MCs.

it including the timing constraints. All these approaches target to validate a particular MC with specific design instance. The approach in [25] requires a reference TLM model to compare against and hardware probing capabilities to monitor different signals. In addition, it requires a methodology to generate representative test cases to stress the MC design. With the increasing complexity of MC designs, industry reports that the test case generation process is becoming time consuming and requires huge intellectual efforts [23]. The methodology in [26] requires modifications to the RTL implementation of the MC to insert the assertions. In addition, approaches of [26] and [27] focus solely on MC's backend.

MCXplore, on the other hand, focuses on modeling the input stimulus of the MC, which enables *MCXplore* to be design agnostic and can be used to validate both MC's frontend and backend. *MCXplore* can leverage different levels of mentoring capabilities. For instance, in Sections V and VI, we utilize high-level metrics such as bandwidth utilization to validate features from both frontend and backend. *MCXplore*, similar to these works, adopts model checking in its framework. However, previous works use model checking to model and validate a specific design as in [24], or to convert the specifications into RTL assertions as in [26]. In contrast, *MCXplore* uses model checking as a test generation engine. Integrating model checking techniques in the test generation process is not a new idea. Model checking has proven its success as a test generation engine for validating both software [28], [29] and hardware [2], [3]. A comprehensive survey of using model checking in testing can be found in [30]. Using model checking set *MCXplore* as a framework that can generate various type of tests. For instance, it can generate *directed test cases*, where the generated test is based on the temporal logic specification and can be used to stress certain behavior of the MC. In addition, *MCXplore* can generate *constrained random test cases*, where the generated test is randomized but obeys certain rules. This is possible because model checking fully explores the state space of the memory test components (addresses, transaction types, etc.). Accordingly, by specifying the rules as specifications, the model checker randomly picks one of the valid paths with minimum transitions (which can be multiple) that satisfies these rules.

1) Industry Solutions: Unfortunately, industrial solutions are intellectual properties (IPs) with only few information available about them. Synopsys has a verification IP to verify the DRAM and the MC [31]. The IP is implemented in SystemVerilog and uses UVM. This IP requires access to the native RTL implementation. Further, it requires licensing, which may be of unaffordable cost. Keysight Technologies and FuturePlus Systems follow a different approach by using special probes and analyzers to monitor the DRAM signals and verify their correctness [32]. This approach is suitable after the manufacturing process is finished (post-silicon validation); however, it requires special hardware tools to conduct the verification process, which may not be costly effective. *MCXplore*, contrarily, can also be used for post-silicon validation, while it does not require special hardware nor additional costs as it is freely available.

Compared to the preliminary version of *MCXplore* [33], this paper introduces the following contributions.

1) It introduces the validation process of two address mapping techniques: a) address masking and b) rank hopping (Sections VI-A2 and VI-A3).
2) It validates and compares three page policies: open-, close-, and adaptive-page policies (Section VI-B).
3) It verifies seven more timing parameters (Section V-A).
4) It validates an REF management technique called *Smart Refresh*, which targets to reduce DRAM power consumption (Section V-B).
5) It validates command bus contention techniques (Section V-C).

IV. MCXPLORE

Fig. 3 represents the steps of the methodology we propose. The process consists of three phases: 1) test template generation; 2) test suite generation; and 3) diagnosis and reporting. 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 [34].

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 behaviors whose correctness needs to be validated. Usually, design engineers provide this list in a highly abstracted human language.

TABLE II
CURRENTLY SUPPORTED CONFIGURATIONS. (A) INDEXING PERFORMANCE. (B) CUSTOMIZED ADDRESS

(a)

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

(b)

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

Step 2: The big challenge for validation engineers is to turn the test plan into meticulous rules that generated tests must follow [35]. 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: 1) a request model and 2) a command interaction model. We encode them as FSMs in the NuSMV model checker [36]. 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 predefined 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 counterexample, 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 BMC 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 necessary 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 memory requests. Validation engineers drive this parser with the address mapping of the 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 behavior. If results match, then they report correctness. Otherwise, they report their diagnosis results and conduct more detailed investigation if required. Section VI provides a methodology to construct test plans with certain expected behaviors to use as a golden metric to compare the MC response.

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$(i, 5)$.

A. Proposed Models

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

1) 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 behavior depends on whether $rw_2 = rw_1$ or not. This is because the MC decision takes into account whether it is a row hit or a row conflict, 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 five segments (row, column, bank, etc.), and the type 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.

2) 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

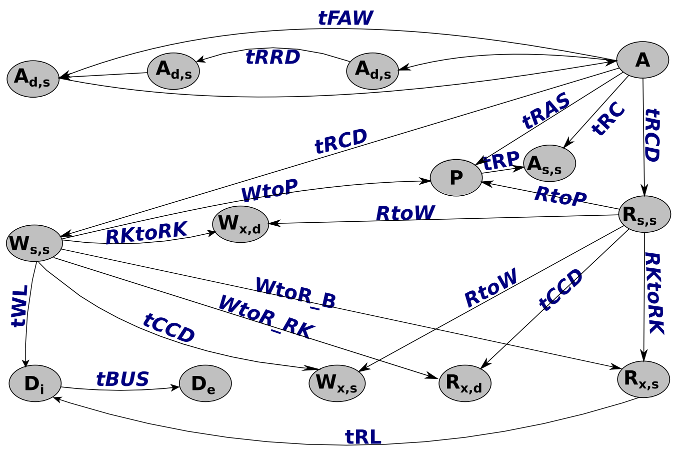


Fig. 4. DRAM commands and timing constraints interaction. Subscripts reflect the targeted bank and rank, respectively. d: different, s: same, x: do not care. Di: start of the data transfer. De: end of the data transfer. P is for same bank. $A_{d,s}$ is an A command to a different bank on the same rank.

to specify the timing constraints to be validated and *MCXplore* automatically generates the test sequence that exercises these constraints. Fig. 4 depicts the state graph of this model that we build based on the timing constraints imposed by the JEDEC standard [7]. 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 V, we use this model to generate test suites for validating the correctness of command generation, and checking for any timing violations. Generally, the request model is better-suited for MC's frontend validation, while the command model well-suits the MC's backend validation. Frontend policies include request arbitration, address mapping, and page policy. MC's backend is responsible for command generation and arbitration.

In the remaining of this paper, we show case studies on applying the proposed methodology to validate the correctness of several state-of-the-art MC policies. We use DRAMSim2 [12] with DDR3-1333 DRAM to conduct the experiments. We use the DDR3-1333 module only as an example. All the lemmas and proofs in this paper are independent of the specific values of the timing constraints; thus, they are applicable to other DRAM modules unless otherwise specified. We also insert common design bugs in the functionality of these features to determine whether the proposed methodology can discover them. We use bandwidth utilization defined in (1) as our metric to validate the MC 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. In (1), the total DRAM access cycles consists of the data transfer cycles and the overhead due to DRAM timing constraints

$$Uti = \frac{\text{Data transfer cycles}}{\text{Total DRAM access cycles}}. \tag{1}$$

It is worth noting that a single metric cannot cover all design bugs. Accordingly, we show the usage of *MCXplore* with other metrics such as the aggregated numbers of issued commands (Section V-B), and the memory access latency (Section V-C).

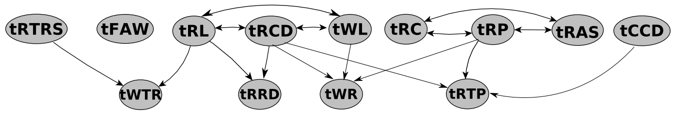


Fig. 5. Validation dependency graph for timing parameters.

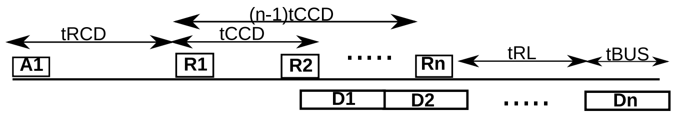


Fig. 6. Command sequence of $Test_{CCD}$.

V. Validating MC's Backend

A. Checking for Timing Violations

Using *MCXplore*, we design property-driven tests 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 Fig. 4, 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 Fig. 5. An edge from constraint $constr_1$ to $constr_2$ means that $constr_1$ must be validated before $constr_2$. A bidirectional edge between two constraints means that they have to be validated together. For space limitation reasons, we only show the complete validation process for the *tCCD* parameter. For *tRTP*, *tRCD*, *tRL*, and *tWL*, we discuss the three major components of the validation process: 1) the test plan; 2) the LTL specifications; and 3) the diagnosis. For all other parameters, we summarize our findings in Table V. Since we validate the timing

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) tCCD (Test Plan): $Test_{CCD}$ formalizes this plan. It is a stream of n read accesses targeting the same bank and row (100% row locality). If $n \gg 10$ in Fig. 6, *tCCD* will dominate the other timing constraints as Lemma 1 proves.

$Test_{CCD}$: Test to validate $tCCD$.

$Test_{CCD} = [Req_1, Req_2, ..., Req_n]$
where $Req_k = \langle Addr_k, R\rangle, k \in [1, n]$
and $(bnk_l = bnk_m) \wedge (rw_l = rw_m), \forall l\ \forall m \in [1, n], n \gg 10$.

Specifications: The LTL in Specification 1 specifies a state, where the number of requests is 100 and the *tCCD* constraint appeared 99 times on the explored path to reach this state.

LTL Specification 1: $tCCD$.

```
G!((num_requests=100) & (num_tCCD=99)&
((num_tRL=1)|(num_tWL=1))&(num_tBUS=1)&(num_Rx_d=0))
```

Test Template and Test Generation: MCXplore invokes the NuSMV model checker to explore the state space to find a counterexample for the specification. The counterexample

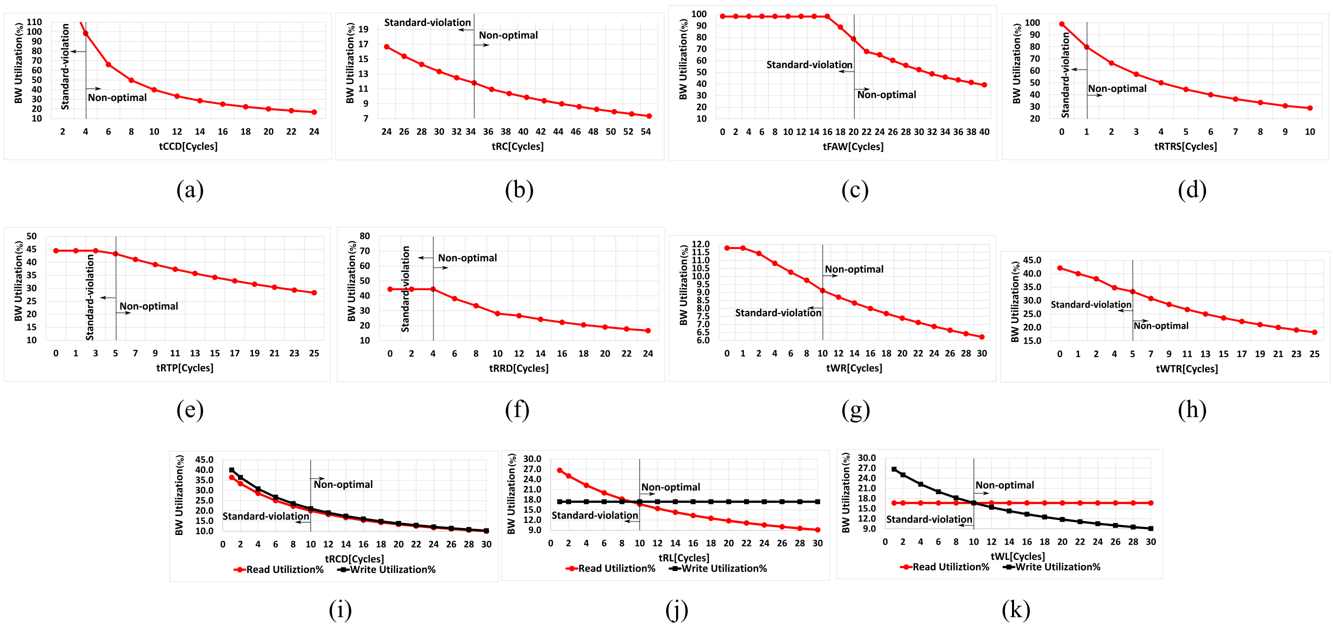
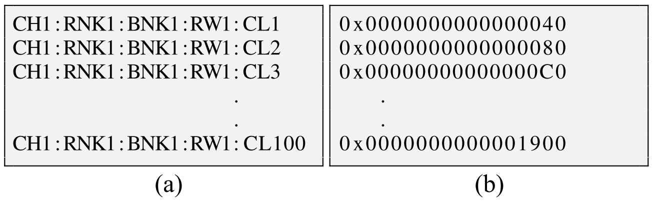


Fig. 7. Timing parameters validation. (a) *tCCD*. (b) *tRC*. (c) *tFAW*. (d) *tRTRS*. (e) *tRTP*. (f) *tRRD*. (g) *tWR*. (h) *tWTR*. (i) *tRCD*. (j) *tRL*. (k) *tWL*.



```
CH1:RNK1:BNK1:RW1:CL1
CH1:RNK1:BNK1:RW1:CL2
CH1:RNK1:BNK1:RW1:CL3
        .
        .
CH1:RNK1:BNK1:RW1:CL100
```
(a)

```
0x0000000000000040
0x0000000000000080
0x00000000000000C0
    .
    .
0x0000000000001900
```
(b)

Fig. 8. Test generation for validating *tCCD*. (a) Test template. (b) Final test example.

represents the test template that exhibits the required test properties. Fig. 8(a) delineates the test template for the specifications in Specification 1. Afterward, the parser parses this template to generate as many tests as required, which conforms with the provided address mapping and syntax. Fig. 8(b) shows one test instance for a 64-bit address machine.

Diagnosis: To validate *tCCD*'s value, we execute the obtained test on the MC under test (DRAMsim2 in this case) and plot the results for various *tCCD* values in Fig. 7(a). Afterward, we compare the observed utilization (Uti_o) with the golden-metric utilization calculated from Lemma 1, Uti_c. Fig. 7(a) illustrates that the observed utilization aligns with the golden metric for $tCCD = 4$, which is the value specified by the standard. For $tCCD > 4$, the optimization is less than the expected value, which implies that the *tCCD* timing is set to a nonoptimal value. Obviously, this leads to a performance degradation as Fig. 7(a) shows. For $tCCD < 4$, the optimization exceeds 100% in DRAMsim2 timings, which implies a corruption in the transferred data.

Lemma 1: Executing $Test_{CCD}$, the BW utilization of the DRAM under test can be approximated to ($tBUS/tCCD$).

Proof: Executing $Test_{CCD}$, the DRAM under test exhibits the behavior shown in Fig. 6. Since $Test_{CCD}$ has n requests, the data bus is busy for $n \cdot tBUS$ cycles. Using Fig. 6, the total DRAM access time can be calculated as: $tRCD + (n - 1) \cdot tCCD + tRL + tBUS$. As a result, the BW utilization of the DRAM under test is: $[n \cdot tBUS/tRCD + (n-1) \cdot tCCD + tRL + tBUS]$. If $n >> 10$, the BW utilization can be approximated to $(tBUS/tCCD)$. ■

2) tRTP (Test Plan): Studying the state graph in Fig. 4, 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. However, this sequence includes the *tRCD*, *tCCD*, and *tRP* constraints as well. Therefore, as the dependency graph in Fig. 5 illustrates, these constraints need to be validated before *tRTP*. In addition, the number of R commands must be large enough to dominate the *tRAS* constraint between A and P. The exact number of required R commands depends on the used DRAM module.

Specifications: Specification 2 shows the LTL property to validate *tRTP*, where `num_tRTP` is the number of occurrences of the *tRTP* constraint. As Fig. 9 illustrates, there are two paths of constraints between the A and the P commands. The first is *tRAS* and the second consists of *tRCD*, a number of *tCCD* constraints that depends on the number of R requests, and finally *RtoP*. Specification 2 ensures that the second path dominates the first one to show the effect of *tRTP* constraint on the utilization.

LTL Specification 2: $tRTP$.

```
G!(((value_READ_TO_PRE_DELAY*num_READ_TO_PRE_DELAY
  +num_tCCD*value_tCCD+value_tRCD)>(value_tRAS)) &
  (num_READ_TO_PRE_DELAY>0) &
  ((num_Rx_d=0) & (num_Rd_s=0))
```

Diagnosis: To validate *tRTP*, we compare the observed utilization (Uti_o) from executing $Test_{RTP}$ with the calculated utilization (Uti_c) from Lemma 2. Based on the comparison, we make the conclusions tabulated in Table III.

Lemma 2: Executing $Test_{RTP}$, the BW utilization of the MC under test is: $(4tBUS/tRCD + 3tCCD + tBUS + tRTP + tRP)$.

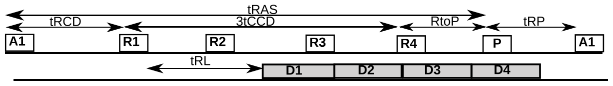


Fig. 9. Command sequence of *Test*$_{RTP}$.

TABLE III
VALIDATING *tRTP*

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

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

3) tRCD, tWL, and tRL (Test Plan): The target is to validate *tRL* and *tWL* parameters, which requires two tests. One request is a read operation and the other is a write operation.

Specifications: The LTL in Specification 3 specifies a state in the FSM, where the explored path encounters an A followed by a R. Similarly, the LTL in Specification 4 describes a state in the FSM, where the explored path encounters an A followed by a W. Since a read or a write from a DRAM row requires first to activate that row, 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.

LTL Specification 3: $tRCD\_tRL$.

```
G!((num_requests=1) & (num_tRL=1) & (num_tBUS=1))
```

LTL Specification 4: $tRCD\_tWL$.

```
G!((num_requests=1) & (num_tWL=1) & (num_tBUS=1))
```

Diagnosis: 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 3 for both tests, then all the three parameters are set to the standard value. For the DDR3 module used in our validation, this situation is observed in Fig. 7(i)–(k) at $tRCD = 10$, $tRL = 10$, and $tWL = 9$. Table IV summarizes our debugging conclusions from the utilization graphs. We assume a single parameter is possibly violated at a time.

Lemma 3: Executing *Test*$_{\{tRCD,tRL\}}$, the utilization of the MC under test is: ($tBUS/tRCD + tRL + tBUS$). similarly, executing *Test*$_{\{tRCD,tWL\}}$, the BW utilization of the MC under test is: ($tBUS/tRCD + tWL + tBUS$).

Proof: In Fig. 10(a), the data bus is utilized for *tBUS* cycles out of $tRCD+tRL+tBUS$ total cycles. Similarly, in Fig. 10(b), the data bus is utilized for *tBUS* cycles out of $tRCD + tWL + tBUS$ cycles. ■

4) Other Parameters: Similar to the aforementioned parameters, for validating other parameters we conduct the following procedure.

1) We execute the corresponding test from Table V.
2) We compare the observed utilization with the calculated utilization.

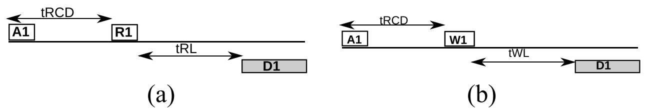


Fig. 10. Validating *tRCD*, *tRL*, and *tWL*. (a) *Test*$_{tRCD,tRL}$. (b) *Test*$_{tRCD,tWL}$.

TABLE IV
VALIDATING *tRCD*, *tRL*, AND *tWL*

$Test_{\{tRCD,tRL\}}$	$Test_{\{tRCD,tWL\}}$	Conclusion	Figure
$Uti_o > Uti_c$ $Uti_o < Uti_c$	$Uti_o > Uti_c$ $Uti_o < Uti_c$	$tRCD$ is violated. $tRCD$ is not optimal.	7i
$Uti_o > Uti_c$ $Uti_o < Uti_c$	$Uti_o = Uti_c$ $Uti_o = Uti_c$	tRL is violated. tRL is not optimal.	7j
$Uti_o = Uti_c$ $Uti_o = Uti_c$	$Uti_o > Uti_c$ $Uti_o < Uti_c$	tWL is violated. tWL is not optimal.	7k

TABLE V
TESTS OF TIMING PARAMETERS

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

3) Based on the comparison, we determine whether the parameter under test is:
   a) compliant with the standard;
   b) violated;
   c) set to a nonoptimal value.

We present all the observed utilizations from all tests in Fig. 7 and tabulate calculated utilizations from all tests in Table V. In Table V, unless specified, the number of requests is $n \gg 10$.

B. Smart Refresh

In this section, we validate the behavior of the *Smart Refresh* technique [37]. This serves three purposes: 1) validating a feature involving the REF command; 2) validating one of the DRAM power reduction techniques; and 3) illustrating the usage of a different metric than the bandwidth utilization with *MCXplore*. The main observation behind Smart Refresh is that a normal access to a DRAM row plays the same role as a refresh command from the data restoration standpoint. Accordingly, Smart Refresh aims to reduce power consumption by skipping refreshing recently accessed rows. The MC maintains a per-row counter and issues a refresh command to a row when its counter value reaches zero. Upon accessing a row, its corresponding counter is set to its maximum value. We implement Smart Refresh in DRAMsim2, where the controller maintains a 2-bit counter per row.

Test Generation: The number of REF commands issued by Smart Refresh is supposed to decrease upon increasing the number of accessed rows in the tests. Therefore, we use *MCXplore* to generate tests with different number of A commands as Specification 5 describes. To capture a considerable number of refreshes, each test has one million requests, and we run the simulation for ten million cycles.

Diagnosis: We use the total aggregated numbers of issued A and REF commands as our metrics in this experiment.

LTL Specification 5: Smart Refresh validation (nACT varies from 8 to 1000000 in a step of 8).

```
G!((num_requests = 1000000)&(num_ACT=nACT)
```

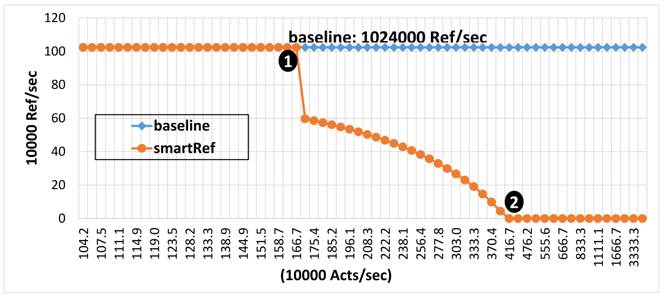


Fig. 11. Smart Refresh behavior with different number of accessed rows.

These numbers can be obtained using performance counters such as those existing in Intel processors [38]. We depict the results for the Smart Refresh technique along with the baseline refresh mechanism in Fig. 11. The baseline issues a fixed number of REF commands agnostically to the access pattern. On the other side, Smart Refresh, as expected, issues less number of REFcommands for tests accessing more rows. Fig. 11 shows that for tests with small number of accessed rows, Smart Refresh acts exactly as the baseline refresh mechanism (until point ❶), which validates the worst scenario. Contrarily, for tests accessing large number of rows (after point ❷), Smart Refresh does not need to issue any extra REF command. This aligns with the ideal scenario represented in [37].

C. Command Bus Contention

All banks of a DRAM rank share the command and data bus. Accordingly, if more than one DRAM command are ready at a specific cycle, the MC must have a policy to select only one command to issue to prevent bus collisions. In this context, a ready command is the command that satisfies all the corresponding timing constraints and can be issued according to the command arbitration policy. For example, commands R2 and R3 in Fig. 12 are ready on the same cycle; however, only one of them can be issued at one cycle. One possible policy is to favor ready requests to different banks to increase DRAM parallelism through interleaving [Fig. 12(a)]. An alternative policy is to favor ready requests to same bank to increase DRAM locality through row hits [Fig. 12(b)].

Test Generation: The target is to generate a test pattern that exhibit a bus contention, i.e., having more than one ready command to be issued on the same cycle. We generate a test with three memory requests, Req_1, Req_2, and Req_3. Req_1 and req_3 target the same row in the same bank, while Req_2 targets a different bank. As Fig. 12 illustrates, both R2 and R3 are ready on the same cycle. Accordingly, one of the commands is issued on cycle ❶, while the other has to be delayed *tCCD* cycles to be issued on cycle ❷.

Diagnosis: There are two possibilities for the test pattern, either R2 is postponed or R3. To verify the policy resolving the command bus contention, we use the request latency as our metric since Req_2 and Req_3 will have different latencies based on the implemented policy. L_i in Fig. 12 is the latency of Req_i. Bandwidth utilization cannot be used in this case since both possibilities lead to the same utilization. Based on the monitored latency, we can figure out which R command is postponed by the MC and compare this to the expected behavior. For instance, in Fig. 12(a), the MC postpones R3; thus, $L_3 > L_2$. On the other hand, in Fig. 12(b), the MC postpones R2 such that $L_3 < L_2$.

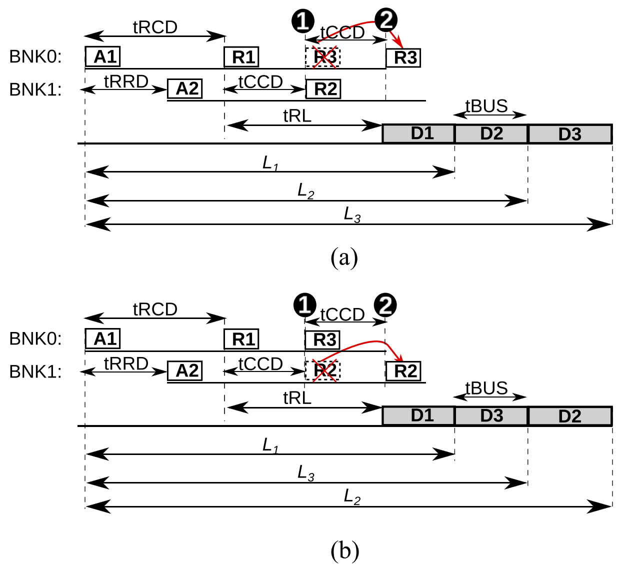


Fig. 12. Policies to resolve command bus contention. (a) Prioritizing accesses to different banks (favor bank interleaving). (b) Prioritizing hit accesses to same bank (favor row hits).

VI. Validating MC's Frontend

We validate features from MC's frontend functionalities including address mapping, page policy, and request arbitration. For space limitation reasons, we show the complete validation process for the XOR address mapping. For the other policies, we discuss the three major components of the validation process: 1) the test plan; 2) the LTL specifications; and 3) the diagnosis. Since all the policies we validate in this section are at the MC's frontend, we use the request model.

A. Address Mapping Policies

We validate three features related to the address mapping component: 1) permutation-based address mapping [4], [6]; 2) address masking [39]; and 3) rank hopping [40], [41].

1) Permutation-Based Page-Interleaving (XOR) Address Mapping: Modern MCs reduce row conflicts by using a permutation-based page interleaving, where the bank bits are bitwise XOR-ed with the least significant row bits [4], [6]. We refer to this technique as simply XOR address mapping.

Test Plan: To generate a test suite 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. $Suite_{XOR}$ formally represents this test plan. Each test has an interleaving percentage between 0% and 100%. *nbnk* is the number of banks per rank (usually eight for DDR3). 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 suite for XOR address mapping.

$Suite_{XOR} = \{Test_{intr} : \forall intr \in [0, 100]\}$
$Test_{intr} = [Req_1, Req_2, ..., Req_n]$, where $Req_k = \langle Addr_k, R\rangle, k \in [1, n]$. $((rw_l = rw_m)$ 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 $l \in [2, \frac{n \times intr}{100}])$.

Specifications: Each test template has its corresponding specification. The LTL in Specification 6 encodes a test plan with $intr = 40\%$, where *t_x* represents the total counts of the event *x*. `t_hit` is the total number of row hits, and `t_bank_interleave` is the total number of bank interleavings. The intuition behind the specification is that out of ten total requests in the test, the first six requests target different rows in the same bank, while the last four requests target the same row but in different banks.

LTL Specification 6: XOR mapping.

```
G ((num_requests=6 & t_hit=0 & t_bank_interleave=0)->
   !F(num_requests=10 & t_reads=10 & t_hit=4 &
      t_bank_interleave=4))
```

Test Template: MCXplore invokes the NuSMV model checker to explore the state space to find a counterexample for the specification. The counterexample represents the test template that exhibits the required test properties. Fig. 13(a) delineates the test template for the specifications in Specification 6.

Test Suite: MCXplore parses each test template and generates a test that complies with the test plan (step 4). Fig. 13(b) shows one test instance generated from the test template.

Diagnosis: For the sake of comparison, we execute $Suite_{XOR}$ on both the XOR mapping and the base mapping (no XOR operation is performed). As Fig. 14 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 behavior of the XOR mapping is to achieve a fixed utilization for all tests in the suite. This is because even for noninterleaved 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 4. Fig. 14 shows that the XOR mapping achieves a fixed utilization of $\sim$ 79%, which coincides with the expected behavior. It is worth mentioning that Lemma 4 assumes that $tFAW \geq 4 \cdot tRRD$ and $2 \cdot tFAW \geq tRC$, which is true for all available DDR chips.

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

Proof: Since XOR mapping maps successive requests of any test in $Suite_{XOR}$ to different banks, the MC under test repeats the behavior shown in Fig. 15 every eight requests. Focusing on one repetition, the data bus is busy for $8 \cdot tBUS$, while the total DRAM latency is $2 \cdot tFAW$. ■

(a)

CH1:RNK1:BNK1:RW1:CL1	R
CH1:RNK1:BNK1:RW2:CL1	R
CH1:RNK1:BNK1:RW3:CL1	R
CH1:RNK1:BNK1:RW4:CL1	R
CH1:RNK1:BNK1:RW5:CL1	R
CH1:RNK1:BNK1:RW6:CL1	R
CH1:RNK1:BNK2:RW6:CL1	R
CH1:RNK1:BNK3:RW6:CL1	R
CH1:RNK1:BNK4:RW6:CL1	R
CH1:RNK1:BNK5:RW6:CL1	R

(b)

0x00000000	R
0x00080000	R
0x00100000	R
0x00180000	R
0x00200000	R
0x00280000	R
0x00290000	R
0x002a0000	R
0x002b0000	R
0x002c0040	R

Fig. 13. Test generation for validating XOR mapping. (a) Test template. (b) Final test example.

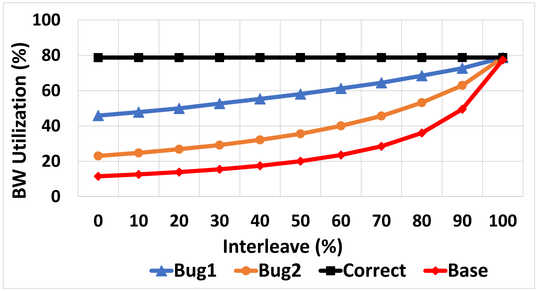


Fig. 14. XOR address mapping.

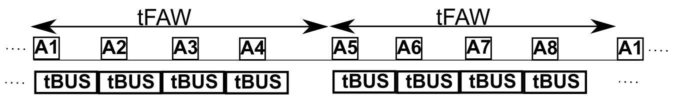


Fig. 15. Command sequence of $Suite_{XOR}$ on XOR mapping.

Bug Scenario: To illustrate potential design errors, we inject two bugs in the XOR mapping. In the first one (Bug1 in Fig. 14), 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 Fig. 14, both Bug1 and Bug2 do not achieve the expected utilization of Lemma 4; hence, they are detectable.

2) Address Masking: MCs map logical addresses to physical addresses by using a bit masking operation. Fig. 16 illustrates the masking operation to extract the row, column and bank segment of an address. Rank and channel segments are extracted with similar logic. We use *MCXplore* to generate tests to validate the address mapping operation for a variety of address mappings with different number of rows, ranks and channels, and with various row sizes. Modern MCs (such as the Intel MC hub [39]) rely on configuration registers to select one of possible address mappings. For clarity, Fig. 17 shows one simple example comprising of a single-rank single-channel DRAM subsystem with three possible address mapping schemes, $Scheme_1$: (*rw-cl-bnk*), $Scheme_2$: (*bnk-cl-rw*), and $Scheme_3$:: (*bnk-rw-cl*).

Test Plan To generate three tests: $Test_1$, $Test_2$, and $Test_3$. Each test is a stream of read accesses targeting the same bank and different rows. Tests differ only in the address mapping, where $Test_1$, $Test_2$, and $Test_3$ are designed corresponding to $Scheme_1$, $Scheme_2$, and $Scheme_3$, respectively.

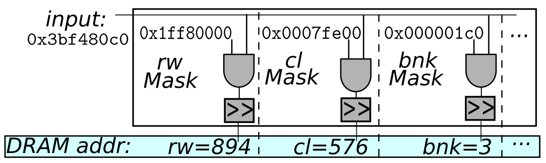


Fig. 16. Address masking operation.

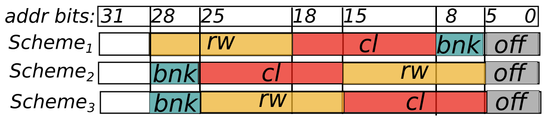


Fig. 17. Address masking schemes.

Specification: Specification 7 encodes the test plan as an LTL property, where the test comprises of ten requests.

LTL Specification 7: Address Masking.

```
G! ((num_requests = 10) & (t_hit=0) &
    (t_bank_interleave=0))
```

Diagnosis: We execute each test on each address scheme and depict the results in Fig. 18. Lemma 5 calculates the expected utilization upon executing $Test_i$ on $Scheme_i$. Each test should result in a utilization of $(tBUS/tRC) \approx 12\%$ when running on its corresponding scheme and larger utilization otherwise.

Lemma 5: Executing $Test_i$ on $Scheme_i$ for $i \in \{1, 2, 3\}$, the BW utilization of the DRAM under test can be calculated as $(tBUS/tRC)$.

Proof: Executing $Test_i$ on $Scheme_i$, the DRAM under test repeats the behavior shown in Fig. 19 every request. Focusing on one request, the data bus is busy for *tBUS* cycles. Moreover, the total DRAM latency required to transmit the data of one request is $tRC = tRP + tRAS$ cycles. ■

Bug Scenario: The masking operation may result in a different mapping than the intended one by the designer. This can be due to a fault in the masking logic, the configuration registers or even a human error when the designer unintentionally sets the wrong address scheme. For example, let the intended mapping to be $Scheme_1$, while the masking mistakenly results in a mapping of $Scheme_3$. Under this scenario, $Test_3$ is the one that will result in the expected utilization and not $Test_1$; hence, we can discover that there is inconsistency between the masking operation mapping and the intended mapping. Since the generated tests target different rows in the same bank, they generate row conflicts and result in a minimum utilization $(tBUS/tRC)$. Accordingly, if a bug results in running $Test_i$ on $Scheme_j$, where $i \neq j$, the resulting utilization will be higher than the expected value as Fig. 18 illustrates.

3) Rank Hopping: Some modern MCs use rank hopping to force consecutive requests to access different ranks [40], [41]. Hence, they will not suffer from the read-write switching required between accesses of different types accessing the same rank. Instead, requests targeting different ranks suffer from a lesser delay *tRTRS*.

Test Plan: $Suite_{HOP}$ formally describes the test plan. The target is to generate a test suite, where each test has a different read-write switching ratio (*sw*) and all requests are targeting the same bank and row.

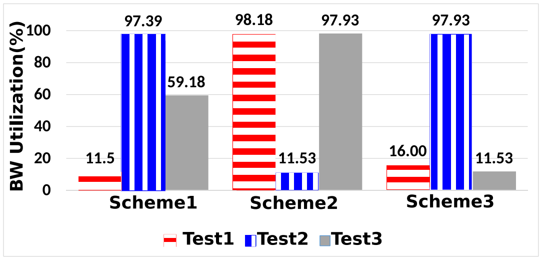


Fig. 18. Address masking results.

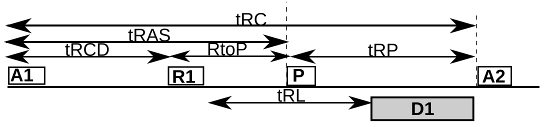


Fig. 19. Command sequence from executing $Test_i$ on $Scheme_i$, $i \in \{1, 2, 3\}$.

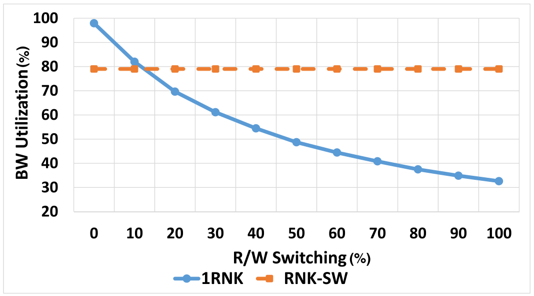


Fig. 20. Rank hopping.

$Suite_{HOP}$: Test suite for rank hopping.

$Suite_{HOP} = \{Test_{sw} : \forall sw \in [0, 100]\}$
$Test_{sw} = [Req_1, Req_2, ..., Req_n]$, where $Req_k = \langle Addr_k, ty\rangle, k \in [1, n]$. $((rw_l = rw_{l-1})$ and $(bnk_l = bnk_{l-1})$ $\forall l \in [2, n])$, and $ty_l = ty_{l-1}$ iff $l \in [2, \frac{n \times sw}{100}]$

Specifications: Each test template has its corresponding specification. The LTL in Specification 8 encodes a test plan with $sw = 40\%$, $(t\_sw = 40)$ out of $(num\_requests = 100)$.

LTL Specification 8: Rank hopping.

```
G!(num_requests=100 &t_hit=99 &t_bank_interleave=99 &
   t_sw=40)
```

Diagnosis: We test both the rank hopping mapping with a dual-rank DRAM, and a single-rank DRAM. We delineate results in Fig. 20. The correct behavior of rank hopping is to achieve a fixed utilization regardless the switching ratio while the utilization of the single-rank DRAM degrades with the increase of the switching ratio due to the *RtoW* and *WtoR_B* constraints. For tests with low switching ratio, we expect the single-rank system to have better utilization than the rank hopping because of the overhead that *tRTRS* constraint adds. Lemma 6 determines the threshold point.

Lemma 6: The rank hopping mapping outperforms the single-rank base mapping if and only if the switching ratio, *sw*, satisfies the following condition:

$$sw > \frac{2tRTRS}{tRL + tWTR + 2tBUS + tRTRS - 2tCCD}.$$

Proof: For the single-rank base mapping, the DRAM utilization for a test of requests with same type targeting same row and bank, $Test_{sw=0}$, can be approximated to $Uti_{no\_sw} = (tBUS/tCCD)$ (proof is in Section V-A1). On the other hand, the DRAM utilization for $Test_{sw=100}$ can be approximated to $Uti_{sw} = (2tBUS/tRL + tWTR + 2tBUS + tRTRS)$. The utilization of $Test_{sw}$ executing on a single rank is calculated in (2). Uti_{1rnk} is the weighted harmonic mean of Uti_{no_sw} and Uti_{sw} with weights $(1 - sw)$ and sw, respectively

$$Uti_{1rnk} = \frac{1}{\frac{1-sw}{Uti_{no\_sw}} + \frac{sw}{Uti_{sw}}}. \quad (2)$$

For the rank hopping mapping, the DRAM utilization can be approximated to $Uti_{hop} = (tBUS/tBUS + tRTRS)$. Hence, the rank hopping outperforms base mapping if: $Uti_{hop} > [1/(1 - sw/Uti_{no\_sw}) + (sw/Uti_{sw})]$. This can occur only if $sw > [Uti_{sw} \times (Uti_{no\_sw} - Uti_{hop})/Uti_{hop} \times (Uti_{no\_sw} - Uti_{sw})]$. By substituting Uti_{no_sw}, Uti_{sw}, and Uti_{hop} and conducting mathematical simplifications, the condition will be $sw > (2tRTRS/\ tRL + tWTR + 2tBUS + tRTRS - 2tCCD)$. ■

Fig. 20 illustrates that for switching ratios that are approximately larger than or equal to 12.5%, the rank hopping mapping outperforms the single-rank base mapping which coincides with the conclusion of Lemma 6.

B. Page Management Policies

Recall from Section II that the page policy controls the liveness of the row in the row buffer. We validate three page policies, which are commonly used in current architectures: close-page, open-page, and adaptive-page policies. Since DRAMsim2 supports only close- and open-page policies, we extend it to support the adaptive-page policy. We implement an adaptive-page policy that models the page policy implemented by Intel [42] and executes the following procedure. If the number of row hits in a decision window is larger than 50%, the MC executes a open-page policy; otherwise, it executes a close-page policy. We choose the decision window to be 20 requests; thus, the MC executes open-page if there are at least 10 hits in the last 20 requests.

Test Plan: To generate a test suite, where each test is a stream of read accesses targeting the same bank with a different locality ratio. $Suite_{PP}$ formally defines this test plan. We define the locality percentage as: loc = (number of row hits × 100/total number of requests). Unlike all previous tests, where all requests of the test are issued back-to-back at cycle 0, $Test_{loc}$ issues a request every $2 \cdot tRC$ cycles. The intuition is that it is necessary that each two successive requests be separated by a period larger than tRC to differentiate between the behavior of the close-page and that of open-page when it has a row-conflict.

$Suite_{PP}$: Test suite for page policy.

$Suite_{PP} = \{Test_{loc} | \forall loc \in [0, 100]\}$
$Test_{loc} = [Req_1, Req_2, ..., Req_n]$, where $Req_k = \langle Addr_k, R, t_k \rangle, k \in [1, n]$
$t_k = 2 \cdot (k - 1) \cdot tRC$
and $(bnk_l = bnk_m)\ \forall l\ \forall m \in [1, n]$, and $(rw_l = rw_m)$ iff $l, m \in [1, \frac{n \times loc}{100}]$.

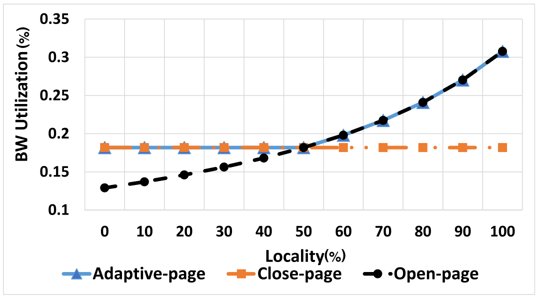


Fig. 21. Evaluation of page policies.

Specifications: Each loc value has its corresponding specification. The LTL in Specification 9 encodes a test plan with $loc = 40\%$, $(t\_hit = 40)$ out of $(num\_requests = 100)$.

LTL Specification 9: Page policy.

```
G!(t_requests=100 &t_hit=40 &t_bank_interleave=0 &
    t_reads=100)
```

Diagnosis: We execute $Suite_{PP}$ on DRAMSim2 for each page policy and depict the results in Fig. 21.

1) *Close-page* is expected to have the same DRAM utilization for all tests. Lemma 7 dictates this utilization. We observe that the obtained results of close-page in Fig. 21 coincide with the expected utilization from Lemma 7.
2) For *open-page*, the utilization depends on the loc percentage. Increasing loc, less precharging is required and the DRAM utilization increases accordingly. Lemma 7 defines the relation between DRAM utilization and loc. Fig. 21 shows that open-page policy outperforms close-page policy for $loc > 50\%$. This value can be directly derived from the utilization values that Lemmas 7 and 8 calculate.
3) Deploying *adaptive-page*, the MC executes the close-page policy for low loc values, while it switches to open-page for high loc values. Fig. 21 shows that the adaptive-page policy has the same utilization as the close-page for $loc \leq 50\%$ and the same utilization as the open-page otherwise.

Lemma 7: Executing $Test_{loc}$ on close-page policy results in the same utilization for all feasible loc ratios. This utilization can be calculated as $(tBUS/tRCD + tRL + tBUS)$.

Proof: As Fig. 22(a) delineates, each request consumes a total of $tRCD + tRL + tBUS$ DRAM cycles to transfer on the data bus for only $tBUS$ cycles. The DRAM remains idle for the remaining of the $2 \cdot tRC$ period and our utilization metric in (1) only considers the DRAM active cycles. ■

Lemma 8: Executing any test in $Tests_{PP}$ on open-page policy, the utilization can be calculated as $[tBUS/(1 - loc)(tRP + tRCD + tRL + tBUS + loc(tRL + tBUS))]$.

Proof: For $loc = 0$, each request incurs a row conflict with a command pattern as shown in Fig. 22(b). Therefore, the BW utilization can be calculated as $Uti_{conf} = (tBUS/tRP + tRCD + tRL + tBUS)$. In contrast, for $loc = 100\%$, each request incurs a row hit with the command pattern in Fig. 22(c). Accordingly, the BW utilization can be calculated as $Uti_{hit} = (tBUS/tRL + tBUS)$. In general, the BW utilization of $Test_{loc}$ with any loc value can be calculated as

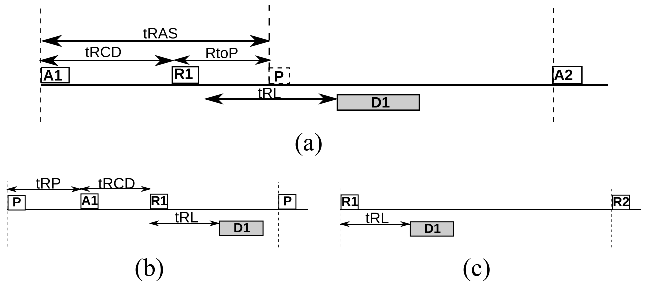


Fig. 22. Command arrangement for *Test*$_{PP}$. (a) Close-page policy. (b) Open-page policy (row conflict). (c) Open-page policy (row hit).

the weighted harmonic mean of Uti_{conf} and Uti_{hit} as follows $(1/[(1-loc)/Uti_{conf}]+[locUti_{hit}])$. This gives the utilization value, which Lemma 8 calculates. ■

C. Arbitration Schemes

We validate an MC feature that affects both the page policy and the arbitration deployed by the MC. Deploying this feature, the MC keeps the row in the row buffer for a designated number of row hits, that we call maximum row-hits threshold. 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 [20]. Usually, the threshold value is chosen such that it maximizes the performance for targeted applications. In this experiment, we assume the intended threshold to be $thr = 16$; though, the procedure is valid for any thr's value.

Test Plan: To generate a set of tests, where each test is a stream of read accesses targeting the same bank. Each test has a different number of requests targeting an open row (row hits), hit. $Suite_{thr}$ formalizes this plan, 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 Suite for threshold-based FR-FCFS arbitration.

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

Specifications: The formula in Specification 10 exemplifies the encoding of the test plan with $hit = 16$ such that the first request opens a row and requests 2 to 17 are hits on that row. Afterward, request 18 opens a different row (row conflict), and requests 19 to 34 are hits on the open row.

LTL Specification 10: Threshold-based FR-FCFS arbitration.

```
G((c_hit=16) -> !F(t_requests = 34 & t_hit=33 &
t_bank_interleave=0 & c_hit = 16))
```

Diagnosis: We execute the generated tests and compare with the expected behavior. The correct functionality is to achieve the maximum utilization (calculated by Lemma 9) when $hit = thr$. Fig. 23 shows that the MC under correct functionality (correct) achieves a maximum utilization of 73% at $hit = 16$, which confirms the conclusion of Lemma 9.

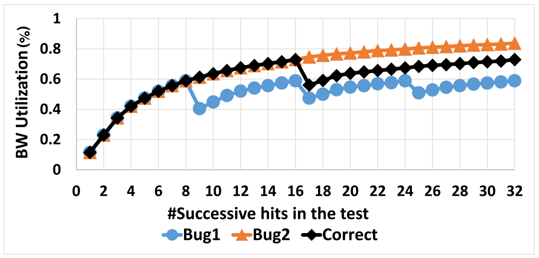


Fig. 23. Evaluation of FR-FCFS threshold.

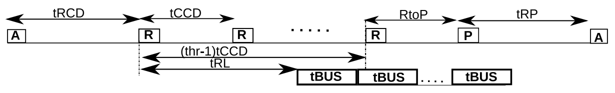


Fig. 24. Command sequence of $Suite_{thr}$ when $hit = thr - 1$.

Lemma 9: 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 $[thr \times tBUS/tRCD+(thr-1)\, tCCD+RtoP+tRP]$.

Proof: When $hit = thr$, the DRAM repeats the behavior illustrated in Fig. 24 every thr requests. During one repetition, the data bus is busy for $thr \times tBUS$, while the total access latency is $tRCD + (thr - 1)tCCD + RtoP + tRP$. ■

Bug Scenario: We embed two bugs to the logic of the row-hits threshold (Bug1 and Bug2 in Fig. 23). Bug1 reduces the threshold to 8 instead of the intended value by the designer (16), while Bug2 increases the threshold to 32. From utilization graphs in Fig. 23, we directly discover that the maximum utilization value is not the expected value calculated by Lemma 9. 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. A similar conclusion can be reached for Bug2.

VII. Extensibility of *MCXplore*

MCXplore treats extensibility as a first-class citizen. To validate a new policy, the designer only needs to identify the properties of that policy and encode them in LTL specifications. Afterward, the designer chooses the suitable model, and *MCXplore* will generate the desired test suites. Furthermore, leveraging the modularity of *MCXplore*, other models can be easily integrated; modifications involve only the interface functions that parse the model to generate the required test. We encourage researchers and validation engineers to extend and use *MCXplore* to validate and test their proposed designs.

VIII. 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*, Intel, Santa Clara, CA, USA, accessed on Aug. 31, 2015. [Online]. Available: http://download.intel.com/design/chipsets/labtour/PVPT_WhitePaper.pdf

[2] H.-M. Koo and P. Mishra, "Test generation using sat-based bounded model checking for validation of pipelined processors," in *Proc. ACM Great Lakes Symp. VLSI*, Philadelphia, PA, USA, 2006, pp. 362–365.

[3] M. Katelman, J. Meseguer, and S. Escobar, "Directed-logical testing for functional verification of microprocessors," in *Proc. ACM/IEEE Int. Conf. Formal Methods Models Co-Design (MEMOCODE)*, Anaheim, CA, USA, 2008, pp. 89–100.

[4] W.-F. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Monterrey, Mexico, 2001, pp. 301–312.

[5] M. Ghasempour, J. D. Garside, A. Jaleel, and M. Lujan, "DReAM: Dynamic re-arrangement of address mapping to improve the performance of DRAMS," in *Proc. Int. Symp. Memory Syst. (MEMSYS)*, Washington, DC, USA, 2016, pp. 362–373.

[6] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proc. ACM/IEEE Int. Symp. Microarchit. (MICRO)*, Monterey, CA, USA, 2000, pp. 32–41.

[7] JEDEC. (2015). *JEDEC DDR3 SDRAM Specifications Jesd79-3d*. Accessed on Aug. 12, 2016. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d

[8] O. Mutlu *et al.*, "Research problems and opportunities in memory systems," *Supercomput. Front. Innov.*, vol. 1, no. 3, pp. 19–55, 2014.

[9] *MCXplore*. Accessed on May 22, 2017. [Online]. Available: https://caesr.uwaterloo.ca/mcxplore/

[10] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proc. IEEE Int. Conf. Formal Eng. Methods*, Brisbane, QLD, Australia, 1998, pp. 46–54.

[11] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," in *Advances in Computers*. Amsterdam, The Netherlands: Academic Press, 2003.

[12] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan./Jun. 2011.

[13] N. Chatterjee *et al.*, "Usimm: The Utah simulated memory module," Univ. Utah, Salt Lake City, UT, USA, Tech. Rep. UUCS-12-002, 2012.

[14] M. Jeong, D. H. Yoon, and M. Erez. *Drsim: A Platform for Flexible DRAM System Research*. Accessed: May 22, 2017. [Online]. Available: http://lph.ece.utexas.edu/public/DrSim

[15] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan./Jun. 2016.

[16] M. Jung *et al.*, "Dramsys: A flexible dram subsystem design space exploration framework," *IPSJ Trans. Syst. LSI Design Methodol. (T-SLDM)*, 2015, pp. 63–74.

[17] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Seattle, WA, USA, 2015, pp. 307–316.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *ACM SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 128–138, 2000.

[19] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Orlando, FL, USA, 2006, pp. 208–222.

[20] 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 Comput. Archit. News*, vol. 40, no. 3, pp. 416–427, 2012.

[21] *DDR3 SDRAM Verilog Model*, Micron, Boise, ID, USA, 2012.

[22] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Porto Alegre, Brazil, 2011, pp. 24–35.

[23] T. Podda *et al.*, *Smart Way to Memory Controller Verification: Synopsys Memory VIP*, Synopsys, Mountain View, CA, USA, accessed on May 22, 2017. [Online]. Available: https://www.design-reuse.com/articles/38587/synopsys-memory-controller-verification.html

[24] D. Sahoo and M. Satpathy, "MSimDRAM: Formal model driven development of a DRAM simulator," in *Proc. IEEE Int. Conf. VLSI Design Int. Conf. Embedded Syst. (VLSID)*, Kolkata, India, 2016, pp. 597–598.

[25] K. Khalifa and K. Salah, "Implementation and verification of a generic universal memory controller based on UVM," in *Proc. IEEE Int. Conf. Design Technol. Integr. Syst. Nanoscale Era (DTIS)*, Naples, Italy, 2015, pp. 1–2.

[26] M. O. Kayed, M. Abdelsalam, and R. Guindi, "A novel approach for SVA generation of DDR memory protocols based on TDML," in *Proc. IEEE Int. Microprocessor Test Verification Workshop*, Austin, TX, USA, 2014, pp. 61–66.

[27] Y. Li, B. Akesson, K. Lampka, and K. Goossens, "Modeling and verification of dynamic command scheduling for real-time memory controllers," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Vienna, Austria, 2016, pp. 1–12.

[28] J. Callahan, F. Schneider, and S. Easterbrook, "Automated software testing using model-checking," in *Proc. SPIN Workshop*, 1996, pp. 1–19.

[29] V. Okun, P. E. Black, and Y. Yesha, "Testing with model checker: Insuring fault visibility," *WSEAS Trans. Syst.*, pp. 77–82, 2003.

[30] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: A survey," *Softw. Test. Verification Rel.*, vol. 19, no. 3, pp. 215–261, 2009.

[31] *VC Verification IP for DRAM Memory*, Synopsys, Mountain View, CA, USA, accessed on May 22, 2017. [Online]. Available: https://www.synopsys.com/verification/verification-ip/dram-memory.html

[32] Keysight Technologies and FuturePlus Systems. *DDR4 Memory Protocol Analysis and Compliance Verification*. Accessed on May 22, 2017. [Online]. Available: http://literature.cdn.keysight.com/litweb/pdf/59911827EN.pdf?id=2295381

[33] M. Hassan and H. Patel, "MCXplore: An automated framework for validating memory controller designs," in *Proc. IEEE Design Autom. Test Europe Conf. Exhibit. (DATE)*, Dresden, Germany, 2016, pp. 1357–1362.

[34] A. Adir *et al.*, "A unified methodology for pre-silicon verification and post-silicon validation," in *Proc. IEEE Design Autom. Test Europe Conf. Exhibit. (DATE)*, Grenoble, France, 2011, pp. 1–6.

[35] Y. Naveh *et al.*, "Constraint-based random stimuli generation for hardware verification," *AI Mag.*, vol. 28, no. 3, pp. 13–30, 2007.

[36] A. Cimatti *et al.*, "NuSMV 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*. Heidelberg, Germany: Springer, 2002.

[37] M. Ghosh and H.-H. S. Lee, "Smart Refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMS," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Chicago, IL, USA, 2007, pp. 134–145.

[38] I. X. Processor, *E5-2600 Product Family Uncore Performance Monitoring Guide*, Intel Corporat., Santa Clara, CA, USA, 2012.

[39] *Memory Controller Hub (MCH), a Datasheet*, Intel, Santa Clara, CA, USA, Feb. 2005.

[40] B. L. Jacob and D. T. Wang, "System and method for performing multi-rank command scheduling in DDR SDRAM memory systems," U.S. Patent 7 543 102, 2009.

[41] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A rank-switching, open-row DRAM controller for time-predictable systems," in *Proc. IEEE Euromicro Conf. Real-Time Syst. (ECRTS)*, Madrid, Spain, 2014, pp. 27–38.

[42] *Intel Xeon Processor X5650*, Intel, Santa Clara, CA, USA, accessed on May 22, 2017. [Online]. Available: http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI

Mohamed Hassan received the M.Sc. degree from Cairo University, Giza, Egypt, in 2012, and the Ph.D. degree from the University of Waterloo, Waterloo, ON, Canada, in 2017.

He is currently a SoC Research and Development Engineer with Intel, Toronto, ON, Canada. His current research interests include realtime embedded systems, computer architecture, hardware validation, and security.

Hiren Patel is an Associate Professor with the Electrical and Computer Engineering Department, University of Waterloo, Waterloo, ON, Canada. His current research interests include realtime embedded systems, computer architecture, and systemlevel design methodologies.