Universidad Autónoma de Madrid

Biblos-e Archivo
Repositorio Institucional UAM

Repositorio Institucional de la Universidad Autónoma de Madrid
https://repositorio.uam.es

Esta es la **versión de autor** del artículo publicado en:
This is an **author produced version** of a paper published in:

Expert Systems with Applications 132 (2019): 44 – 62

DOI: https://doi.org/10.1016/j.eswa.2019.04.070

Copyright: © 2019 Elsevier Ltd. All rights reserved.

El acceso a la versión del editor puede requerir la suscripción del recurso
Access to the published version may require subscription

An expert system for checking the correctness of memory systems using simulation and metamorphic testing

Pablo C. Cañizares[1] , Alberto Núñez[1] and Juan de Lara[2]

[1]*Dept. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain*

e-mail: pablocc@ucm.es, alberto.nunez@pdi.ucm.es

[2]*Dept. Ingeniería Informática, Universidad Autónoma de Madrid, Spain*

e-mail: juan.delara@uam.es

Abstract

During the last few years, computer performance has reached a turning point where computing power is no longer the only important concern. This way, the emphasis is shifting from an exclusive focus on the optimisation of the computing system to optimising other systems, like the memory system. Broadly speaking, testing memory systems entails two main challenges: the oracle problem and the reliable test set problem. The former consists in deciding if the outputs of a test suite are correct. The latter refers to providing an appropriate test suite for determining the correctness of the system under test.

In this paper we propose an expert system for checking the correctness of memory systems. In order to face these challenges, our proposed system combines two orthogonal techniques – simulation and metamorphic testing – enabling the automatic generation of appropriate test cases and deciding if their outputs are correct. In contrast to conventional expert systems, our system includes a factual database containing the results of previous simulations, and a simulation platform for computing the behaviour of memory systems. The knowledge of the expert is represented in the form of metamorphic relations, which are properties of the analysed system involving multiple inputs and their outputs. Thus, the main contribution of this work is two-fold: a method to automatise the testing process of memory systems, and a novel expert system design focusing on increasing the overall performance of the testing process.

To show the applicability of our system, we have performed a thorough evaluation using 500 memory configurations and 4 different memory management algorithms, which entailed the execution of more than one million of simulations. The evaluation used mutation testing, injecting faults in the memory management algorithms. The developed expert system was able to detect over 99% of the critical injected faults, hence obtaining very promising results, and outperforming other standard techniques like random testing.

Keywords: Memory Systems, Metamorphic Testing, Simulation, Mutation Testing, Expert Systems, Memory Scheduling

1. Introduction

Currently, CPU manufacturers are proposing multi-core CPUs as the answer to scaling up system performance. This trend has led to the emergence of increasingly powerful systems, provided with several CPUs consisting of multiple processing cores. As an example, the Multi-Purpose Processing Array processor integrates 256 cores in a single 28nm CMOS chip de Dinechin et al. (2014).

Systems based on multi-core architectures achieve a fair improvement level in terms of performance Gepner & Kowalik (2006). Generally, in multi-core platforms, the main memory is shared to enable communication between different processes Mahapatra & Venkatrao (1999). Hence, when the number of CPU cores increases, this memory becomes a system bottleneck and, consequently, emphasises the significance of *memory wall* Wulf & McKee (1995) and *bandwidth wall* Rogers et al. (2009) problems. The current trend focuses on designing hierarchical multi-channel architectures aimed at exploiting the parallelism in multi-core systems. These architectures provide sophisticated and complex memory systems that alleviate this issue.

However, designing an efficient memory hierarchy is a difficult task faced by system designers. From a design perspective, there is a wide spectrum of possible configurations and parameters along multiple dimensions that must be carefully analysed before providing a valid design. For instance, there are different important key factors that have a direct impact on the overall memory performance. These include the number of memory controllers, their placement and the number of channels supported by each controller, just to name a few Awasthi et al. (2010); Kim et al. (2010); Abts et al. (2009). Moreover, there is a vast number of choices related to the organisation of the memory device, like the hierarchical organisation of channels, banks, rows and columns Jacob et al. (2007).

Thus, checking the correctness of memory systems is crucial to ensure system scalability. Beyond the architectural design, testing scheduling policies orchestrated by the memory controller is a challenging problem. First, the controller needs to obey all DRAM timing constraints to provide correct functionality. Second, the controller must intelligently prioritise DRAM commands from different memory requests to optimise system performance Ipek et al. (2008). In order to completely check a memory system, in a systematic and exhaustive way, a broad range of hardware configurations and controller architectures must be analysed, which becomes a time-expensive and costly task. Moreover, analysing the power consumption of different memory chips may require an additional hardware, which significantly increases the monetary cost of the testing process.

Unfortunately, applying conventional testing techniques for checking memory architectures entails two main difficulties. First, test suites consisting of a large number of test cases are needed to accurately check the system under study, which requires a high computational cost. Additionally, since each test case must be executed in the platform under study, this process requires access to specific hardware. Second, an oracle that indicates if a given system is correct or not is, in most situations, unavailable or computationally too expensive Weyuker (1982). Moreover, the PASS/FAIL output provided by the major part of the testing techniques is not enough to locate where the fault has been produced. It is therefore desirable that the user obtains information that helps identifying the part of the system under test that is not working as expected.

In order to alleviate these issues, we propose an expert system that combines two orthogonal techniques to check the correctness of memory systems: simulation and metamorphic testing (in short, MT). On the one hand, simulation techniques are especially useful when the expected architectures are not available or expensive. Hence, these techniques provide a cost-effective

method to simulate the behaviour of the target architecture. On the other hand, MT Chen et al. (1998) is a technique developed for testing systems where there is no oracle, or it is too expensive to compute Weyuker (1982). MT is based on metamorphic relations (in short, MRs), which describe properties of the system under study. The essential idea is that instead of checking the output o_1 produced when testing with one input x_1, we test with a second (follow-up) input x_2, observing the obtained output o_2, and then check that o_1 and o_2 are related as specified in the MR. Thus, in MT there are two relations: the relation between the original test input x_1 and the follow-up input x_2, and the expected relation between the two outputs.

Our proposed expert system consists of an inference engine, a knowledge base, a simulation platform, a graphical user interface and a factual database. The knowledge is introduced in the system by an expert in the form of MRs, which model the behaviour of the different parts of the memory system. The proposed system is scalable in the sense that the knowledge base can be updated by the expert, that is, the number of MRs can be increased, which improves the completeness of the system to check new memory models. Since each MR focuses on a specific part of the memory, the expert system is able not only to detect if a memory system is correct or not, but to provide precise information about the cause of the error and the part of the system where it is located. Additionally, the factual database stores the results of previous simulations, which increases the overall system performance by accessing those results that have been previously calculated.

In order to demonstrate both the applicability and suitability of the expert system, we have performed an experimental study using 500 different memory configurations and 4 memory management algorithms. In this study, we used 50 different workloads inspired by the PARSEC benchmarks Bienia et al. (2008). We designed 10 different MRs to analyse the correctness of each memory system configuration. In addition, 5 different mutants were generated for each scheduling algorithm to evaluate the effectiveness of our expert system. In the testing process we execute 25,000 different test cases over the original system and the generated mutants. Overall, in this process, we executed more than a million of simulations. The results obtained are promising, since the expert system was able to detect the vast majority of the injected faults. In general, 90% of the injected faults representing critical errors in the system have been detected, while the system provides acceptable results to detect those faults focusing on general aspects of the system, obtaining a 64% of average effectiveness. However, it is important to remark that when the expert combines several MRs in the knowledge base, the system is able to detect over 99% of the injected faults. Finally, a second set of experiments shows the benefits of our method with respect to standard techniques like random testing. The benefits are both in terms of effectiveness of the testing process (our method discovers more faults, more efficiently), and effort (the tester needs to provide a complete oracle, which is not required in our case).

The rest of the paper is organised as follows. Section 2 presents a literature review. Section 3 shows a brief overview of memory systems and introduces the main concepts of MT. In Section 4 we describe our model to represent different memory system configurations. Section 5 presents our catalogue of MRs. In section 6, our proposed expert system is described in detail. Next, in Section 7, we present a thorough experimental evaluation. Finally, Section 8 finishes with the conclusions and future work.

2. Literature review

During the last years, several approaches targeted to check the correctness of memory systems have been proposed Dreibelbis et al. (1998); Huang et al. (1999); Karpovsky et al. (1995);

Miyano et al. (1999); Pundir & Sharma (2017); Yang et al. (2015). Those proposals focus on creating fault models and testing algorithms to analyse different types of memory systems. Among them, we can highlight several significant techniques, such as design-for-testability Dreibelbis et al. (1998), built-in self-test Huang et al. (1999); Yang et al. (2015), simulation-based algorithms Wu et al. (2000) and mathematical approaches using finite-state machines de Goor & Smit (1994a,b). Although they are considered cost-effective to identify some of the most common errors in memory systems, none of them focus on memory management policies.

In the field of memory controllers, some novel techniques aimed to design and check new algorithms to enhance the scheduling process have been proposed. In most cases, these proposals are validated using manual and random testing techniques Hassan et al. (2015); Rixner et al. (2000), as well as with benchmarks Lin et al. (2001); Ghasempour et al. (2016). Random testing is a widely used technique for checking the correctness of computer systems. The main advantage of random testing is its simplicity, which allows to automatically generate test cases and execute them with a reasonable effort. However, this technique has some weaknesses. First, large test suites are required to reach a good level of coverage. Second, high computational resources are required to execute large test suites. Third, due to the stochastic nature of the generated test cases, there is no guarantee that these are appropriate for accurately testing the system. Consequently, it is possible that critical parts of the system remain untested. On the contrary, manual testing is an arduous and error prone task that requires a considerable effort from the tester. The main advantage of this technique is the certainty that specific and critical parts of the system are tested. A common weakness of both techniques is the need for an oracle that checks if the outputs generated by the test cases are correct. In many cases, the (human) tester acts as the oracle.

In order to alleviate these issues, several proposals based on model checking Clarke et al. (2001) have been proposed. Sahoo and Satpathy proposed MSimDRAM Sahoo & Satpathy (2016), a formal model-driven framework to model and check DRAM controllers. The authors modelled the DRAM memory controller using the SAL language and finite state machines. In addition, in order to check the requirements, the authors encoded the correct behaviour using linear temporal logic (LTL). Then, they used bounded model checking, a technique for checking the satisfiability of a property. Khalifa and Salah presented a generic universal memory controller Khalifa & Salah (2015). This approach is based on a system-level architecture that has been verified using the universal verification methodology. The verification environment consists of two monitors, a reference transaction level model (TLM), and a driver that generates test cases, which are applied to the reference and the design under test to find possible errors. Kayed et al. presented a novel approach based on the JEDEC standards Standard (2012). This way, to verify the validity of the generated commands, the timing constraints are defined using a Timing Diagram Mark up Language and transformed into system Verilog assertions Kayed et al. (2014). Li et al. proposed the modelling and verification of dynamic command scheduling for real-time memory controllers Li et al. (2016). In this work, the memory controller is modelled using timed automata and is analysed using model checking through the UPAAL tool. Hassan and Patel proposed an approach to automate the validation process of DRAM memory controller designs, known as MCXplore Hassan & Patel (2017). This proposal provides a methodology oriented to generate test cases, using the properties defined in each policy, to maximise the coverage. Moreover, this methodology can be used to seamlessly validate the policies of memory controllers.

Although there are numerous advantages related to the use of model checking for analysing the correctness of systems, like the high coverage it achieves, the previously described ap-

proaches entail different issues, which are alleviated by our proposed system. First, these solutions are focused on verifying memory controllers with an ad-hoc model design and, therefore, these require specific requirements for each design, like a reference TLM model Khalifa & Salah (2015) and a specific register-transfer level implementation Kayed et al. (2014). Hence, in order to successfully achieve a new version of the scheduler, several temporal logic constraints must be re-adapted and re-written for each memory controller. Second, in general, the translation of the system under test to a checkable-model using formal languages is a complex and difficult task. In some cases, the real system is too complex to be represented with enough fidelity using a given formalism. Third, these approaches are not scalable and require powerful resources and a high execution time. Finally, in some of these systems the user obtains a YES/NO answer indicating whether a system is correct or not and, therefore, lacking information to locate the anomaly. In summary, these facts hamper the generalisation of the model checking proposals to analyse memory management policies in real systems.

In the last 5 years, MT has been applied in different application domains, including the validation of complex systems Segura et al. (2016). For example, Jiang et al. proposed several MRs to ensure the correctness of CPU schedulers Jiang et al. (2013). As a result, two faults were detected in one of the simulators under study. Ding et al. presented an iterative approach focused on the development and refinement of MRs for testing scientific applications Ding et al. (2016). Núñez and Hierons proposed a methodology based on MT for fault detection in cloud systems Núñez & Hierons (2015). Cañizares et al. proposed preliminary ideas for designing energy aware systems Cañizares et al. (2015). These works show that MT can be applied in complex systems where an oracle is not available.

During the last decade, simulation tools have gained popularity to model and analyse memory systems. Rosenfeld et al. presented DRAMSim2, whose main strength is its accuracy for simulating DDR2/DDR3 models Rosenfeld et al. (2011). Also, DRAMSim2 provides different models to represent the energy consumption of the simulated memories. The main weakness of DRAMSim2 is the lack of mechanisms to deploy different memory management policies. The Utah SImulated Memory Module (in short, USIMM Chatterjee et al. (2012)) is a trace-based memory system simulator focused on DDRx memories. USIMM provides mechanisms for modelling the different components of the memory system, such as the system architecture, DRAM timing and latency parameters, scheduling policies and power consumption. Moreover, USIMM includes some of the most used PARSEC benchmarks Bienia et al. (2008). Jeong et al. proposed DrSim, a simulation platform for modelling DRAM systems, which provides a widespread spectrum of memory architectures and topologies Jeong et al. (2012). Similarly, Kim et al. Kim et al. (2016) presented Ramulator, a fast and cycle-accurate DRAM simulator that supports an extensive spectrum of DRAM standards, such as DDR3/4, LPDDR3/4, GDDR5, WIO1/2 and HBM. The main advantage of this simulator is its performance, which appoints Ramulator as the fastest memory simulator. However, several weaknesses like the high abstraction level of the memory components and the lack of both power consumption models and memory management policies, make Ramulator not appropriate for our proposed system. Although these simulators support modelling and simulation of memory systems, the testing process must be manually performed.

To the best of our knowledge, expert systems have not been applied to check the correctness of memory systems. However, they have been successfully applied – as an effective approach for analysing complex systems – to a wide variety of domains including, among others, acoustic diagnosis Hussain et al. (2015), power systems Liberado et al. (2015), geographic information systems C.M. Herrero-Jiménez (2012), fault diagnosis of computer systems Bennett & Hollander (1981) and productivity of industrial environments J. Bautista-Valhondo & R. Alfaro-Pozo

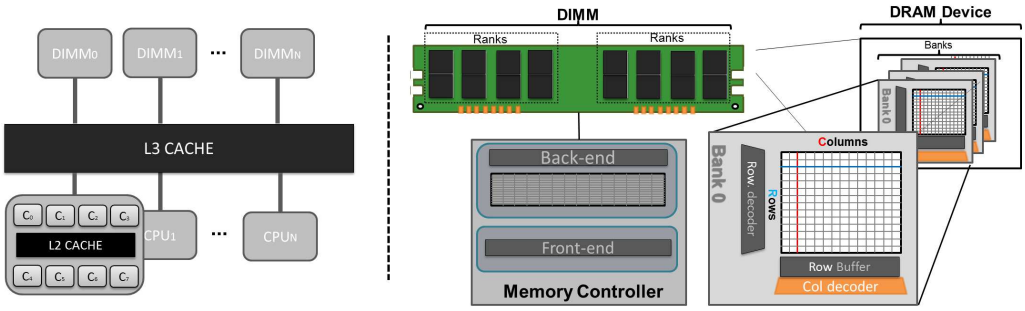


Figure 1: Architecture of the memory system

(2018). Hence, we think that expert systems are perfectly suitable to achieve our goals.

3. Background

In this section we provide introductory concepts about the memory system and MT.

3.1. The memory system

Since the last decades, memory systems are based on Dynamic Random Access Memory technologies (in short, DRAM) Hardee et al. (1991). The continuous evolution of the computational systems has encouraged the sophistication of this technology by increasing its throughput and capacity. Currently, DDR4 is the prevailing off-chip memory technology. The initial JEDEC DDR4 DRAM specification was released in September 2012 Council (2017), where memory speeds described in this standard are expected to reach 3,200 Mbps, while its predecessor DDR3 reached 1,600 Mbps Mukundan et al. (2013).

The system organisation and main components of DRAM are illustrated in Figure 1. The left of the figure depicts a typical high-performance processing architecture. In this environment, several processors are connected to the memory system through a memory controller, which allows parallel access using multiple channels.

Current DDRx memories are structured as Dual In-line Memory Modules (in short, DIMM), which consist of several devices. The DIMMs are interconnected through a data bus to the memory controller. The right part of Figure 1 shows the DRAM channel organisation, which consists of a set of channels connected to a collection of DIMMs. Each DIMM consists of a small number of *ranks*, which contain several DRAM devices (also called chips).

A rank is a collection of DRAM devices that operate in parallel. Thus, each *rank* is itself partitioned into a set of *banks* that are independently controlled and have their own row buffers (also called pages) of data Sudan et al. (2010). The DRAM controller manages the memory requests using several scheduling policies, while obeying timing and hardware constraints of the DRAM chip to improve performance.

Requests from the CPU arrive to the memory controller, which translates them into a collection of orchestrated commands for DRAM access. Once the data request arrives to the controller, the memory address is extracted and then the channel is selected. At this point, it is important to avoid information leakage in the memory controller Gundu et al. (2014); Shafiee et al. (2015).

Next, the DIMM and a rank inside it are calculated. Thus, DRAM devices within a rank synchronously work together to return as many bits of data as the width of the channel. In general, accesses to a DRAM device require first selecting a bank and then a row. For any read request, a row of data is read into the row-buffer associated with the bank.

3.2. Metamorphic Testing

Traditional testing techniques require checking the conformance between the input(s) and the output(s) of the system under study. Schematically, let $\mathcal{S}$ be a system, I the input domain and TS a test selection strategy. Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\} \subseteq I$ be the set of tests generated by using TS. When these tests are sequentially applied to the system $\mathcal{S}$ we obtain a sequence of outputs $\mathcal{S}(t_1), \mathcal{S}(t_2), ..., \mathcal{S}(t_n)$. Given an oracle f, an error is found in $\mathcal{S}$ if there exists $t_i \in \mathcal{T}$ such that $\mathcal{S}(t_i) \neq f(t_i)$.

However, a complete oracle f, able to exactly characterise the expected output of a test, is challenging in many domains, including ours. In order to alleviate this problem, we propose using MT techniques Chen et al. (1998); Ding et al. (2016); Liu et al. (2014). The main difference between traditional testing techniques and MT lies in the comparison of the obtained outputs. This way, while traditional techniques compare the output of each individual test case with the one obtained from the oracle, MT checks the relation between multiple test inputs and their outputs.

MT uses expected properties of the target system relating multiple test inputs with the corresponding outputs obtained from the system under test. These properties are formulated as metamorphic relations. A metamorphic relation (in short, MR) is a property of the analysed system that involves multiple inputs and their outputs. We represent a MR as a tuple (MR_i, MR_o), where MR_i refers to the relation between the source test case and the follow-up test case, and MR_o refers to the relation that must be fulfilled by the outputs obtained from the source test case and the follow-up test case.

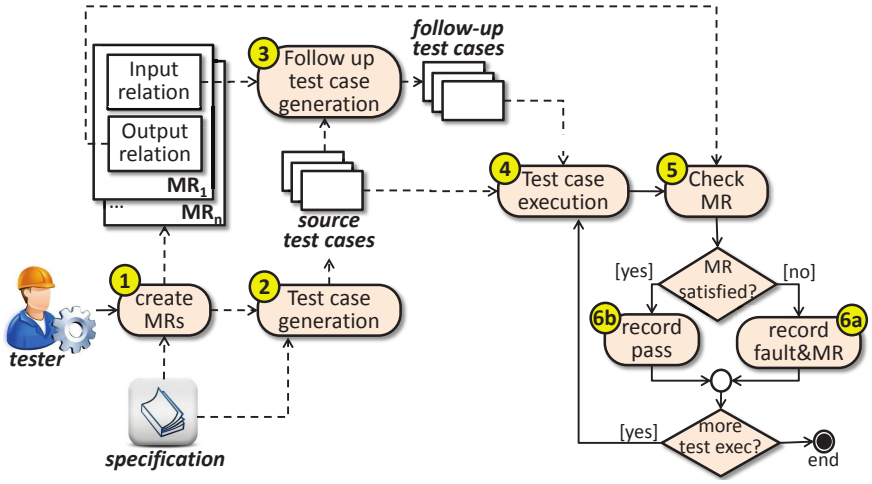


Figure 2: Scheme of the MT process

Figure 2 illustrates the MT process. Initially (activity with label ①), the tester must build a repository of MRs, which act as an oracle to check whether the outputs returned by the system

under test are the expected ones. These MRs must be designed according to the specification of the system under test. In this work, the specification of memory systems is used to create MRs. Please note that the tester must be able to interpret the specification of the system under test, if it is available, to properly build suitable MRs. This task is specially challenging when the system under study is complex.

Next, for each *MR* in the repository of MRs, the tester must build a test suite consisting of a collection of source test cases. These test cases must be generated considering the specification of the system under test (label ② in the Figure). Any traditional testing technique, like random testing Ciupa et al. (2011), can be used to create each test suite. Similarly, in the next step (label ③), for each previously generated source test suite in ②, a new follow-up test suite, containing the same number of test cases, is built. Thus, follow-up test cases are generated by using both the source test cases and the input relation of *MR*, that is, MR_i.

Next, each test case generated in the previous steps is executed against the system under study (label ④). When the execution of all the test cases finishes, the MRs are used to check the obtained outputs (label ⑤). In order to accomplish this task, these MRs are chosen one by one from the repository of MRs created in ①. Hence, for each *MR*, the source and follow-up test cases are used to check whether their outputs satisfy the relation given by MR_o. If the relation is not satisfied, an error has been found, and the corresponding fault and the violated MR are stored to analyse the issue (see 6a). On the contrary, if the *MR* is fulfilled, the statistics are updated increasing the number of test cases that satisfy this relation (see 6b). Once all the test cases are checked against the *MR*, the next MR is chosen. This process is repeated until all the MRs are processed.

4. Modelling the memory system and the workloads

Our approach needs a model of the memory system, and for this purpose we use the structure given by Definition 1. This structure is inspired by the architecture of DDRx systems, where off-chip memories associated with each memory channel are hierarchically organised in terms of ranks, banks, rows and columns Jacob et al. (2007). Also, the memory page can be modelled by configuring the row buffer size parameter (in short, *rbs*). In order to provide a high level of flexibility, this model supports the configuration of a variable number of latencies. Thus, a wide spectrum of memory systems, using different levels of complexity, can be modelled. It is important to remark that our proposed expert system is not focused on a specific simulation platform. On the contrary, the user must select the most appropriate parameters used in the target simulator to configure the required memory system from the model.

Definition 1. A memory system m is a tuple $(dimm, chan, rank, chip, bank, row, col, freq, rbs, lat)$, where:

- $dimm \in \mathbb{N}$ is the number of memory modules,
- $chan \in \mathbb{N}$ is the number of channels,
- $rank \in \mathbb{N}$ denotes the set of DRAM chips that can be simultaneously accessed,
- $chip \in \mathbb{N}$ is the number of DRAM chips of each *rank*,
- $bank \in \mathbb{N}$ refers to the total number of banks in each $chip$,

- $row \in \mathbb{N}$ represents the total number of rows per *bank*,
- $col \in \mathbb{N}$ represents the total number of columns per *bank*,
- $freq \in \mathbb{R}$ denotes the frequency of m measured in MHz,
- $rbs \in \mathbb{N}$ denotes the row buffer size measured in bytes, and
- lat is a set $\{l_i | i \geq 0\}$ that refers to the different latencies used in m.

□

We denote by m_{size} the total size of the memory system m, measured in bytes, which can be calculated using the following formula:

$$m_{size} = m_{dimm} * m_{rank} * m_{chip} * m_{bank} * m_{row} * m_{col} \tag{1}$$

The input tests are described by workloads, which are sequences of operations that are executed over a memory system. Their structure is given by Definition 2.

Definition 2. A workload ω is a sequence $\{r, w\}^*$, where:

- r represents a read operation, denoted as a tuple $(nops, addr1, addr2)$ where $nops$ is the number of non-memory instructions carried out before the operation, $addr1$ and $addr2$ are the addresses related with the read operation.
- w is a write operation, denoted as a tuple $(nops, addr)$ where $nops$ is the number of non-memory instructions before the operation and $addr$ is the address where the write is performed.

□

In the following, we denote the empty sequence by ϵ, and use $numR(\omega)$ and $numW(\omega)$ to refer to the number of read and write operations in the workload ω. These operations can be calculated using the following functions:

$$numR(\omega) = \begin{cases} 0 & \text{if } s = \epsilon \\ 1 + numR(\omega') & \text{if } s = r \cdot s' \end{cases}$$

$$numW(\omega) = \begin{cases} 0 & \text{if } s = \epsilon \\ 1 + numW(\omega') & \text{if } s = w \cdot s' \end{cases}$$

We say that a workload ω is included in ω', written $\omega \leq \omega'$, if $\exists x, y \in \{r, w\}^*$ s.t. $\omega' = x \cdot \omega \cdot y$. We write $\omega = \omega'$ when both workloads are equal, that is, ω and ω' have the exact same elements in the same order.

In order to test a memory system, a suitable collection of test cases needs to be generated. A test case is a pair (m, ω^n), where m is a memory system, ω is a workload and n is the number of workload instances to be executed. These instances are equal, that is, each one has the exact same elements. For the sake of clarity, we use the notation ω (omitting the super-index) to represent the execution of 1 workload instance. We assume a suitable simulator able to run the workloads, and produce information about time, power, and number of performed read and write operations.

Definition 3. Let m be a memory system and ω be a workload. The result of simulating the execution of n workload instances of ω over the memory system m is denoted by $S(m, \omega^n)$. In those cases where $n > 1$, we assume that a dedicated CPU is used to execute each workload instance.

The output obtained from simulating the workload ω over the memory system m is represented with the following notation:

- $S_T(m, \omega^n) \in \mathbb{R}_+$ denotes the time required to execute n instances of ω over m.
- $S_P(m, \omega^n) \in \mathbb{R}_+$ denotes the power required to execute n instances of ω over m.
- $S_W(m, \omega^n) \in \mathbb{N}_0$ denotes the number of performed write operations to execute n instances of ω over m.
- $S_R(m, \omega^n) \in \mathbb{N}_0$ denotes the number of performed read operations to execute n instances of ω over m.

□

5. A catalogue of MRs for testing memory systems

In this work we have designed a suitable collection of MRs[1]. These relations represent properties of the system under test – inferred by experts – that are stored in the knowledge base and used by the inference engine to check the correctness of memory systems. Next, we formally define the pattern of our relations.

Definition 4. A *metamorphic* relation MR for a memory system m and a workload ω is the set of 4-tuples

$$MR(m, \omega) = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} MR_i((m, \omega), (m', \omega')) \\ \Downarrow \\ MR_o(S(m, \omega), S(m', \omega')) \end{array} \right\}$$

where MR_i is a relation over the source test case and a follow-up test case, and MR_o is a relation over the results obtained from the execution of these test cases. □

In this work we propose a catalogue of MRs focusing on different aspects of memory systems, such as performance, energy consumption and functionality. The design of these MRs is inspired by common errors found in memory management algorithms. In order to identify these errors, different sources have been carefully investigated. We started our efforts by analysing bug reports and "*whats new*" logs from different repositories containing memory management algorithms. In particular, we analysed repositories of well-known simulators, including Ramulator Kim (2017), DRAMSim Rosenfeld (2017), NVMain Poremba (2017) and USIMM Kumar (2017). From this analysis we gathered several errors committed by real programmers, like wrong managing of ranks, wrong use of channel indexes, wrong timing in write operations, deadlock scenarios with large memory sizes and several issues with the memory controller state, among others.

[1]The collection of MRs is available at `http://antares.sip.ucm.es/cana/MRlist.pdf`

Next, we studied different papers found in the current literature. Since memory scheduling is the most important function of the memory controller, the research community has invested a significant effort to analyse and test a wide-spectrum of techniques to optimise the overall performance of memory systems Modgil et al. (2015); Natarajan et al. (2004). We focused our efforts in investigating those aspects of the analysed techniques that may produce a bug in the system like, just to name a few, delayed write scheduling, request re-ordering features and in-order request processing.

Finally, we studied a specialised site on micro-controller architectures. Specifically, we investigated the topic focusing on software-based memory testing Barr (2017), where the author remarks the importance of detecting issues in the scheduling algorithms to avoid catastrophic failures, like bypassing a memory channel or bypassing a rank.

In the following, we describe the catalogue of the 10 proposed MRs. These focus on checking for errors gathered from the previous study.

$$MR_1 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} \omega = \omega' \wedge m_{chan} > m'_{chan} \\ \wedge \\ m_{lat} = m'_{lat} \\ \Downarrow \\ S_T(m, \omega) \leq S_T(m', \omega') \end{array} \right\}$$

MR$_1$: Given two memory models m and m' and two workloads ω and ω', if the workloads ω and ω' is equal, and the number of channels of m is greater than the number of channels of m', and the latencies of both memories are equal, then the time required to execute ω over m should be less or equal than the one required to execute ω' over m'.

$$MR_2 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} m = m' \\ \wedge \\ numW(\omega) > numW(\omega') \\ \wedge \\ numR(\omega) > numW(\omega') \\ \Downarrow \\ S_T(m, \omega) > S_T(m', \omega') \end{array} \right\}$$

MR$_2$: Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal, and the number of write operations in ω is greater than the number of write operations in ω', and the number of read operations in ω is greater than the ones of ω', then the time required to execute ω over m should be greater than the time required to execute ω' over m'.

$$MR_3 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} \omega = \omega' \wedge m_{size} < m'_{size} \\ \Downarrow \\ S_W(m, \omega) = S_W(m', \omega') \\ \wedge \\ S_R(m, \omega) = S_R(m', \omega') \end{array} \right\}$$

MR$_3$: Given two memory models m and m' and two workloads ω and ω', if ω and ω' are equal and the size of memory m is smaller than the size of memory m', then the number of performed write and read operations during the execution of ω over m should be equal to the number of performed write and read operations during the execution of ω' over m', respectively.

$$MR_4 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} \omega = \omega' \\ \wedge \\ m_{lat} \geq m'_{lat} \wedge m_{chan} = m'_{chan} \\ \Downarrow \\ S_T(m, \omega) \geq S_T(m', \omega') \end{array} \right\}$$

$\boldsymbol{MR}_4$: Given two memory models m and m' and two workloads ω and ω', if the workloads ω and ω' are equal and the latency of memory m is greater or equal than the latency of memory m', and the number of channels of both memories are equal, then the time required to execute ω over m should be greater than or equal to the time required to execute ω' over m'.

$$MR_5 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} \omega = \omega' \\ \wedge \\ m_{lat} \geq m'_{lat} \wedge m_{chan} = m'_{chan} \\ \Downarrow \\ S_W(m, \omega) = S_W(m', \omega') \\ \wedge \\ S_R(m, \omega) = S_R(m', \omega') \end{array} \right\}$$

$\boldsymbol{MR}_5$: Given two memory models m and m' and two workloads ω and ω', if ω and ω' are equal and the latency of memory m is greater than or equal than the latency of memory m', and the number of channels of both memories are equal, then the number of performed write and read operations during the execution of ω over m should be equal to the number of performed write and read operations during the execution of ω' over m', respectively.

$$MR_6 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} m = m' \wedge \omega' \leq \omega \\ \Downarrow \\ S_P(m, \omega) \geq S_P(m', \omega') \end{array} \right\}$$

$\boldsymbol{MR}_6$: Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal and all the elements of ω' are contained in ω, then the power required to execute ω over m should be greater than or equal to the power required to execute ω' over m'.

$$MR_7 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega') \\ S(m, \omega), \\ S(m', \omega') \end{array} \right) \middle| \begin{array}{c} \omega = \omega' \wedge m_{size} < m'_{size} \\ \wedge \\ m_{lat} = m'_{lat} \wedge m_{chan} = m'_{chan} \\ \Downarrow \\ S_P(m, \omega) < S_P(m', \omega') \end{array} \right\}$$

$\boldsymbol{MR}_7$: Given two memory models m and m' and two workloads ω and ω', if the workloads ω and ω' are equal and the size of memory m is less than the size of memory m', and the latencies of both memories are equal, and both memories have the same number of channels, then the power required to execute ω over m should be less than the power required to execute ω' over m'.

$$MR_8 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega'^n) \\ S(m, \omega), \\ S(m', \omega'^n) \end{array} \right) \middle| \begin{array}{c} m = m' \wedge \omega = \omega' \wedge n > 1 \\ \Downarrow \\ \lceil \frac{n}{m_{chan}} \rceil \cdot S_T(m, \omega) \\ > \\ S_T(m', \omega'^n) \end{array} \right\}$$

MR$_8$: Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal, and the workloads ω and ω' are equal, then the time required to execute n instances of ω' over m' should be less than $\lceil \frac{n}{m_{chan}} \rceil$ times the time required to execute ω over m.

$$MR_9 = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega'^n) \\ S(m, \omega), \\ S(m', \omega'^n) \end{array} \right) \middle| \begin{array}{c} m = m' \wedge \omega = \omega' \wedge n > 1 \\ \Downarrow \\ n \cdot S_W(m, \omega) = S_W(m', \omega'^n) \\ \wedge \\ n \cdot S_R(m, \omega) = S_R(m', \omega'^n) \end{array} \right\}$$

MR$_9$: Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal and the workloads ω and ω' are equal, then the number of performed write and read operations during the execution of n instances of ω' over m' should be equal to n times the number of performed write and read operations during the execution of ω over m, respectively.

$$MR_{10} = \left\{ \left(\begin{array}{l} (m, \omega), \\ (m', \omega'^n) \\ S(m, \omega), \\ S(m', \omega'^n) \end{array} \right) \middle| \begin{array}{c} m = m' \wedge \omega = \omega' \wedge n > 1 \\ \Downarrow \\ S_P(m, \omega) < S_P(m', \omega'^n) \end{array} \right\}$$

MR$_{10}$: Given two memory models m and m' and two workloads ω and ω', if the memory systems m and m' are equal and the workloads ω and ω' are equal, then the power consumption associated to execute ω over m should be less than the power required to execute n instances of ω' over m', it being $n > 1$.

6. Proposed expert system

An expert system is a computational system that emulates the decision-making of human expertise using domain specific knowledge. The main difference between an expert system and a conventional one lies in the method used to solve complex problems. That is, while expert systems apply reasoning based on rules, conventional systems are based on procedural code.

In this section we present our proposed expert system. Its basic architecture is shown in Section 6.1 and the testing procedure is described in Section 6.2.

6.1. Architecture of the proposed expert system

In this work we propose an expert system for checking the correctness of memory systems using MT and simulation techniques. In contrast with the conventional architecture of expert systems, we have included two additional modules: a factual database and a simulation platform. Thus, our proposed system consists of 5 main modules (see Figure 3).

The *knowledge base* (in short, KB) is a module that is built using the knowledge of the expert. In essence, KB consists of rules and facts. In this case, the rules are introduced into the KB by the expert, in the form of MRs.

In some situations, different MRs may have the same MR_i (the relation with the source test case and the follow-up test case), while the MR_o (the relation that must be fulfilled by the obtained outputs) differs. In particular, this is the case of MR_4 and MR_5, where the former relation focuses on performance by comparing the time required to execute a given workload, while the latter focuses on functionality by comparing the number of read and writes. In these cases, there

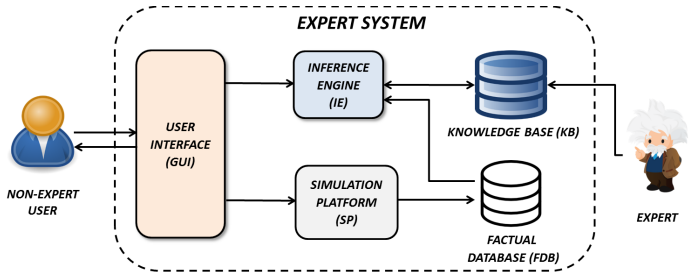


Figure 3: Architecture of the proposed expert system

are two possible solutions. First, using two different MRs, like MR_4 and MR_5. This solution is used when different outputs, obtained from the same input, are required to be separately analysed. Second, to construct a new MR by combining several existing relations Liu et al. (2012). For instance, let us consider a memory scheduler that must process a given workload. The algorithm of this scheduler may, or may not, fulfil the corresponding specification. Also, this algorithm may, or may not, be efficient enough to be considered acceptable. If we use our proposed expert system to check this algorithm, we will be able to decide if there is an unexpected behaviour of the algorithm under test by analysing those tests that do not fulfil MR_4 and MR_5. Hence, if there are tests that do not fulfil MR_4, there is an error in the implementation of the algorithm. On the contrary, if there are tests that do not fulfil MR_5, we can assume that the obtained performance of the scheduling algorithm is not acceptable. However, if we use a combined MR, it is more difficult to locate the unexpected behaviour of the system under test.

In some cases using all the available MRs may be too expensive in terms of computational and time costs, and a subset or combination of them must be selected. This is so because the follow-up test cases must be created ad-hoc for each MR. Therefore, the higher the number of MRs, the higher the number of follow-up test cases that need to be generated. Consequently, the expert has to take the decision of introducing into the KB separate MRs or to combine them.

The *user interface* module is a friendly and easy-to-use application – written in Java – that provides a graphical user interface (in short, GUI). Using this GUI, non-expert users can perform different tasks like modelling new memory systems, editing the configuration of a current memory model and testing memory models. Figure 4 shows the editor for modelling memory systems. Basically, this editor contains each parameter of the memory model and its corresponding value. The left part of the panel shows a repository of memory models, where users can easily save, edit and remove memory models in the application.

Figure 5 shows a panel that allows users to select the MRs that will be used in the testing process. These rules are obtained from the KB and displayed in the GUI. Thus, if the expert updates the rules in the KB, these are also updated in the GUI.

The *simulation platform* (in short, SP) is in charge of two main tasks. First, once the non-expert user has defined a memory model and selected the required MRs, the SP uses this information to automatically generate a set of follow-up test cases. Second, for each generated follow-up test case, the *factual database* (in short, FDB) is accessed to request information of the test. If the test is stored in the FDB, then the required information is obtained from the SP. In other case, the SP executes the simulation of the test case to produce the results and to extract the required information to be stored in the FDB.

The facts and rules – also called MRs – are analysed by the *inference engine* (in short, IE)

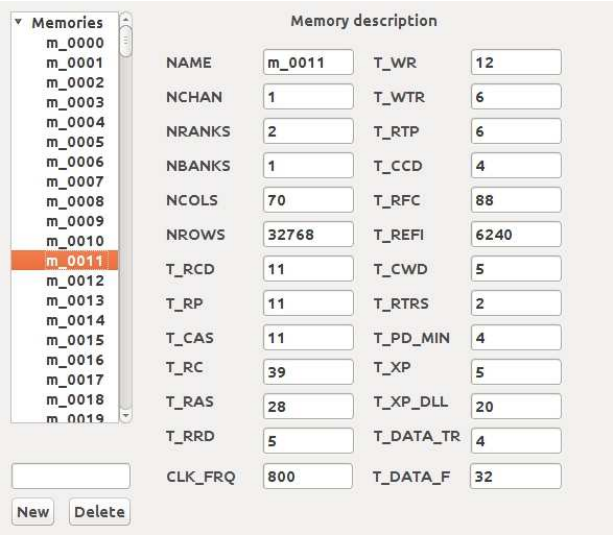


Figure 4: GUI: Editor for modelling memory configurations.

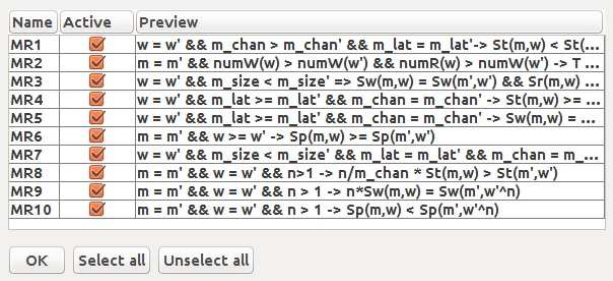


Figure 5: GUI: Panel to select the MRs used in the testing process.

and, for each test case, the IE checks if the involved MRs in the testing process are fulfilled by matching the obtained outputs.

6.2. Testing procedure

This section presents a detailed description of the required steps to test a memory system using our approach (see Figure 6). For the sake of clarity and completeness, we also describe some relevant internal steps performed by the main modules of the expert system.

In the first place, a memory model needs to be defined (label 1). Next, the non-expert user must select a memory scheduling policy (label 2), and as in any testing process, a set of input test cases need to be provided (label 3). These test cases are inspired by the PARSEC suite Bienia et al. (2008). In the following step (label 4) the user selects one, or several, MRs, and follow-up test cases are automatically generated by the SP (label 5).

In order to illustrate the concepts described in this section, we present an example that shows the generation of a follow-up test case using MR_1. Listing 1 represents a source test case, where m is a memory system, l is a tuple representing the latencies of m and ω is a (simplified) workload.

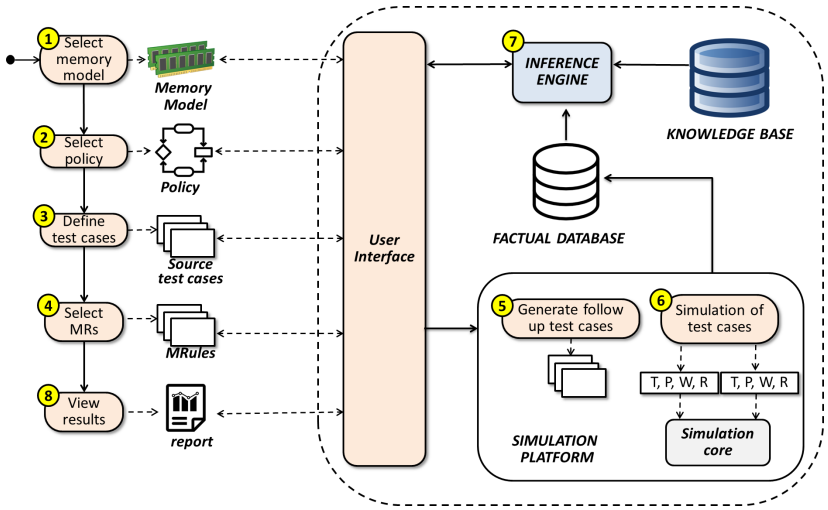


Figure 6: Scheme of the testing procedure

Listing 2 shows the generated follow-up test case, which satisfies the relation of MR_{1_i}. This test case has been generated by using a memory containing less channels than the memory used in the source test case. The workload and latencies are the same for both test cases.

$$
\begin{aligned}
l &= (9,9,9,39,28,5,32,12,6,6,\\
&\quad 4,128,6240,5,2,4,5,20,4)\\
m &= (1,\mathbf{2},7,32768,128.0,1)\\
\omega &= \{r,r,r,w,r,w,r\}\\
T_{source} &= (m,\omega)
\end{aligned}
$$

Listing 1: Source test case

$$
\begin{aligned}
l' &= (9,9,9,39,28,5,32,12,6,6,\\
&\quad 4,128,6240,5,2,4,5,20,4)\\
m' &= (1,\mathbf{1},7,32768,128.0,l') \blacktriangleleft\\
\omega' &= \{r,r,r,w,r,w,r\}\\
T_{follow-up} &= (m',\omega')
\end{aligned}
$$

Listing 2: Follow-up test case matching MR_1

In step 6, the SP checks if the required information for the test exists in the FDB. If this is the case, then the simulation of the test case is not executed, and this information is obtained from the FDB. On the contrary, the memory management policy is simulated on the memory model using both the input test cases and the follow-up test cases. The simulation provides outputs, typically informing about the consumed power, time, and number of read/write operations.

Next, the IE uses both the facts and rules from the KB and FDB to check those MRs that are fulfilled by the test cases. If they do not match, it means an error has been found. If they match, the confidence on the correctness of the memory system increases.

Finally, in the last step (label 8), the IE generates a report that is displayed in the GUI. Figure 7 shows the results of testing a memory system containing faults. In this case, the expert system detects faults in the read queue, write queue and delays in the operations, which are detected by MR_2, MR_7 and MR_8 (see Explanation area in Figure 7). Similarly, Figure 8 shows the results of testing a correct memory system.

It is important to remark that our proposed expert system locates faults in *isolated* memory systems using simulation. Thus, a wide spectrum of memory configurations can be automatically and efficiently tested. If a fault is found during the testing process, the expert must fix the memory system module of the corresponding operating system. Unfortunately, not all the existing

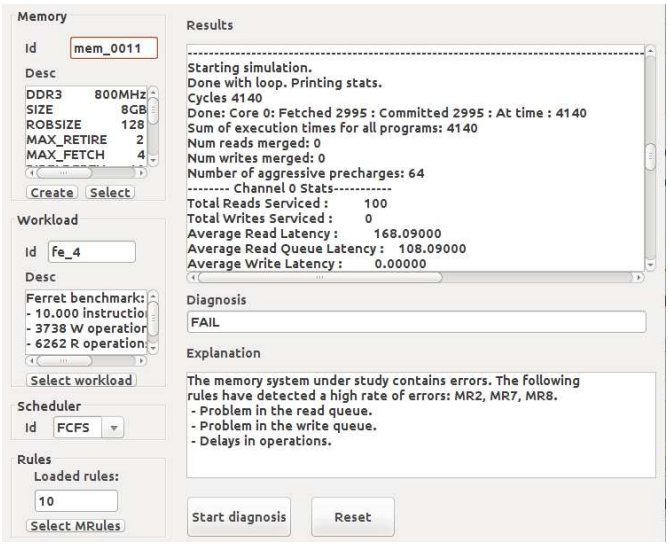


Figure 7: GUI: Results of a faulty memory system.

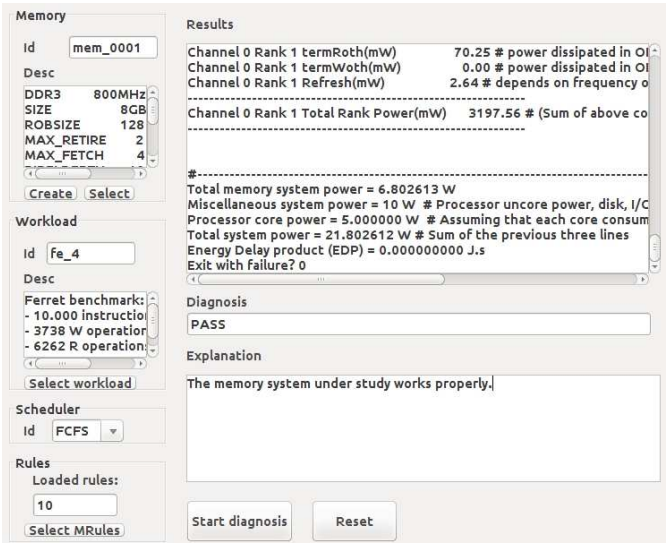


Figure 8: GUI: Results of a correct memory system.

operating systems allow the modification of the memory system internals, like the memory allocating policy and, therefore, the applicability of our proposed system depends on the availability of the target operating system to apply changes in its memory system. However, open-source operating systems, like Linux, allow not only to modify the actual memory system configuration, but to include new and customised algorithms. In any case, using an appropriate and error-free memory system accurately exploits the overall system performance achieving, in the major part of the cases, a good trade-off between performance and hardware complexity Subramanian et al.

(2016); Modgil et al. (2015).

7. Empirical study

Once we have developed our expert system, there are three main research questions (RQs) to be answered, namely:

RQ1: *Are the designed MRs suitable to be used as rules in the knowledge base?*

We ask the first research question to check if the designed KB – using MRs – properly represents the underlying behaviour of the memory system. To answer **RQ1**, we have performed an experiment using the USIMM simulator Chatterjee et al. (2012) (which we initially assumed correct), creating 500 memory models and 50 different workloads based on the PARSEC suite Bienia et al. (2008), and using 4 memory management algorithms[2]. The results of these experiments are detailed in Section 7.2.

RQ2: *How effective is the proposed expert system to catch errors in faulty memory systems?*

Next, we ask a further question to know the effectiveness of the expert system for detecting errors in faulty memory systems. In this case, we answer **RQ2** by testing different faulty versions of the system under test. Hence, minor syntactical faults were artificially injected in the memory management system, which is considered the main core of the memory system. These faulty versions are known as *mutants* in the sense of *mutation testing* Hierons et al. (2010). The results of this experiment are described in Section 7.3.

RQ3: *How suitable is the proposed expert system to test memory systems compared with standard methods?*

Finally, we are interested in investigating the suitability of our expert system for testing memories, compared to other methods. In order to ask this question, we have analysed the current standard methods for testing memory systems. We focus the search on general methods that are suitable for analysing a wide spectrum of memory configuration. Hence, methods for testing a specific memory system are not considered in this study. The results of this comparison are shown in Section 7.4.

The rest of this section is organised as follows. The experimental setting is described in Section 7.1. Sections 7.2 and 7.3 detail the two experiments performed. A comparison between our approach and a standard method for testing memory systems is shown in Section 7.4. The results obtained in the experiments are discussed in Section 7.5, where we also answer the research questions. Finally, we analyse threats to validity in Section 7.6.

7.1. Experimental setting

The main goal of the proposed expert system is to check the correctness of memory systems using realistic memory models with a high level of detail. Hence, even though our system is general and independent of a specific simulator, we consider USIMM as the most suitable option for the purposes of our study. First, this simulator provides a good compromise between the

[2]The experiments results can be found at `http://antares.sip.ucm.es/tools/expertSystems/experiments.7z`

high level of detail in the hardware models and the inherent flexibility to model a wide range of memory systems. Second, USIMM supports simulation using customised memory management policies. In fact, USIMM has been used in the Memory Scheduling Championship Chatterjee et al. (2012). Third, this simulator provides an accurate power consumption model and generates a detailed collection of statistics as output.

For both experiments 500 different memory models have been generated. Also, 50 different workloads have been created, which are inspired by PARSEC benchmarks, such as *blackscholes, facesim, ferret, fluidanimate, freqmine, streamcluster and swaptions*. In order to generate traces that are representative of the selected benchmarks, the SimPoint platform has been used Sherwood et al. (2003). During the trace generation process, SimPoint uses basic block vectors to recognise execution intervals that can be used to reflect the behaviour of the benchmark.

Each memory model generated in this study has been tested with 4 different memory management algorithms. Two of these algorithms are based on well-known scheduling policies: first come, first serve (in short, FCFS) and an approach based on the close page policy (in short, CPP). The other 2 management algorithms won the Memory Scheduling Championship: high performance memory access (in short, HPMA) and request density aware fair memory (in short, RDAF) Balasubramonian (2012).

Table 1 shows the list of parameters used for modelling different memory systems in the USIMM simulator Chatterjee et al. (2012), where the first column refers to the name of the parameter in the simulator and the second column presents a description of the parameter. The first six parameters are related with architectural constraints, such as the number of channels, ranks and banks of the memory system. The rest of these parameters refers to latency constraints, such as row to column interval delay, column access and row cycle interval delay, among others.

7.2. Assessing the suitability of the knowledge base (RQ1)

The main objective of this first experiment is to check the suitability of the knowledge base, which uses MRs to represent the behaviour of memory systems. Initially, we assume that the scheduling algorithms and the simulator are correct.

In this experiment, the testing process consists in executing 25,000 different test cases, generated from combining 500 memory models and 50 benchmarks, over the original system. This process is carried out for the 4 memory management algorithms. Also, in those cases where the workload is executed in parallel, an additional simulation is performed. Table 2 depicts the results of this experiment, which shows the percentage of test cases that fulfil each MR using the 4 memory scheduling algorithms. These results show that the major part of the MRs is satisfied by all the test cases. In contrast, only MR_6 and MR_{10} are fulfilled by none of the test cases.

For those cases where a given MR is satisfied for all the test cases, we assume this MR correct. However, there are 2 possible scenarios for those cases where none of the test cases satisfy the MR: the MR is not correct or, on the contrary, the simulator has a limitation to execute some test cases.

In order to reject the assumption that the MR is not correct, we have carefully designed a few test cases that should fulfil the involved MRs. For instance, to check MR_6, we use $\omega = \omega' \cdot \omega'$, that is, the workload executed over m is the result of concatenating the workload ω' to itself. Since both memories m and m' are equal, the power required to execute ω over m must be greater than the power required to execute ω' over m'. Similarly, we have manually generated some tests to check the correctness of MR_{10}. However, we obtain different results than expected and, therefore, we conclude that the simulator has a limitation to represent certain scenarios.

Parameter	Description
NUM_CHANNELS	Number of channels
NUM_RANKS	Number of ranks per channel
NUM_BANKS	Number of banks per chip
NUM_ROWS	Number of rows
NUM_COLUMNS	Number of columns
DRAM_CLK_FREQ	Frequency of the memory (MHz)
T_RCD	Row to Column interval delay
T_RP	Row pre-charge delay
T_CAS	Column access
T_RC	Row cycle interval delay
T_RAS	Row access strobe
T_RRD	Interval row activation delay
T_FAW	Four bank activation window
T_WR	Write recovery time
T_WTR	Write to read delay time
T_RTP	Read to pre-charge
T_CCD	Column to Column delay
T_RFC	Refresh cycle time
T_REFI	Refresh interval period
T_CWD	Column write delay
T_RTRS	Rank to Rank switching time
T_PD_MIN	Minimum power down duration
T_XP	Time to exit fast power down
T_XP_DLL	Time to exit slow power down
T_DATA_TR	CPU to memory transfer time

Table 1: Configuration parameters in USIMM Chatterjee et al. (2012)

Consequently, since MR_6 and MR_{10} are not able to evaluate the underlying behaviour of the memory system using USIMM, these MRs have been removed from the KB and not used in the following experiment.

Id	MR_1	MR_2	MR_3	MR_4	MR_5	MR_6	MR_7	MR_8	MR_9	MR_{10}
Cpp	100	100	100	100	100	0	100	100	100	0
$Fcfs$	100	100	100	100	100	0	100	100	100	0
$Hpma$	100	100	100	100	100	0	100	100	100	0
$Rdaf$	100	100	100	100	100	0	100	100	100	0

Table 2: Percentage of test cases that satisfy each MR

7.3. Assessing the effectiveness of the expert system (RQ2)

In this section we evaluate the effectiveness of our proposed expert system for finding errors in memory management systems. In order to accomplish this analysis, we have used mutation testing to generate 5 different mutants from each memory management algorithm. These mutants reproduce typical errors committed by programmers while designing memory management policies. For this, different faults have been seeded in the main parts of the memory controller,like

Id	Original Statement	Faulty statement	Description
M_1^{Cpp}	$if(wr\_ptr \rightarrow comm\_issuable)$	$if(!wr\_ptr \rightarrow comm\_issuable)$	Delaying a write operation
M_2^{Cpp}	$if(wr\_q\_len[ch] > HI)$	$if(wr\_q\_len[ch] < HI)$	Modifying the write queue
M_3^{Cpp}	$rd\_q\_len[ch] - -$	$rd\_q\_len[ch]$	Modifying the read queue
M_4^{Cpp}	$if(!drain\_wr[ch])$	$if(drain\_wr[ch])$	Swapping a read for a write
M_5^{Cpp}	$wr\_q\_head[ch]$	$wr\_q\_head[++ch]$	Swapping operation channel
M_1^{Fcfs}	$if(drain\_wr[ch] \wedge wr\_q\_len[ch]$	$if(drain\_wr[ch] \| \| wr\_q\_len[ch])$	Forcing a write operation
M_2^{Fcfs}	$if(!rd\_q\_len[ch])$	$if(rd\_q\_len[ch])$	Modifying the read queue
M_3^{Fcfs}	$if(wr\_q\_len[ch] > HI)$	$if(wr\_q\_len[ch] < HI)$	Modifying the write queue
M_4^{Fcfs}	$if(wr\_ptr \rightarrow comm\_issuable)$	$if(!wr\_ptr \rightarrow comm\_issuable)$	Delaying a write operation
M_5^{Fcfs}	$for(ch = 0; ch < NC; ch++)$	$for(ch = 1; ch < NC; ch++)$	Bypassing a channel
M_1^{Hpma}	$if(issue\_request(rdat\_ptr))$	$if(!issue\_request(rdat\_ptr))$	Delaying a read operation
M_2^{Hpma}	$if(rdat\_ptr \neq NULL)$	$if(rdat\_ptr == NULL)$	Delaying a read operation
M_3^{Hpma}	$if(drain\_wr \| \| issue\_wact)$	$if(drain\_wr \wedge issue\_wact)$	Modifying the serving policy
M_4^{Hpma}	$if(wdat\_ptr \neq NULL)$	$if(wdat\_ptr == NULL)$	Delaying a write operation
M_5^{Hpma}	$switch(sboard[ch].state)$	$switch(sboard[ch].state++)$	Changing the controller state
M_1^{Rdaf}	$if(wr\_ptr \rightarrow comm\_issuable)$	$if(!wr\_ptr \rightarrow comm\_issuable)$	Delaying a write operation
M_2^{Rdaf}	$if(r\_ptr \rightarrow comm\_issuable)$	$if(!rd\_ptr \rightarrow comm\_issuable)$	Delaying a read operation
M_3^{Rdaf}	$if(wr\_q\_len[ch] > HI)$	$if(wr\_q\_len[ch] < HI)$	Modifying the write queue
M_4^{Rdaf}	$is\_TFW(ch, rank, cycle))$	$is\_TFW(ch, ++rank, cycle)$	Bypassing a rank
M_5^{Rdaf}	$state[ch][rank][bank].next$	$state[ch][rank][bank].next++$	Changing the controller state

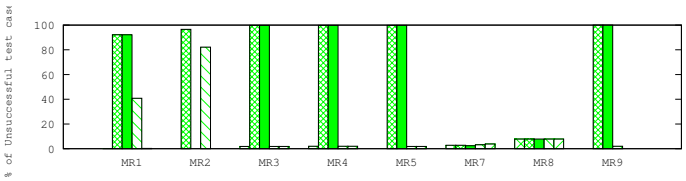
Table 3: Description of the generated mutants

the scheduler and the read/write queue management (delaying operations, modifying the queue, etc). The mutations are summarised in Table 3. The algorithms used in this study are independent to each other and, therefore, their source code is different in all cases. Hence, it has not been possible to exactly create the same mutants for each algorithm and thus a collection of errors has been specifically adapted for each planner.

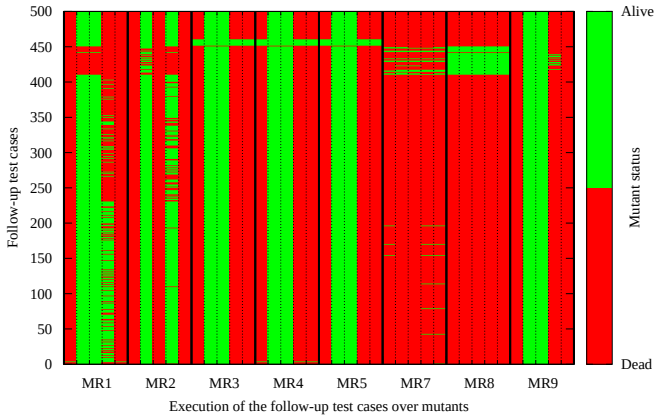
The testing process of this experiment uses the same test cases that were generated in Section 7.2. However, in this case, these tests are executed over the original system and the generated mutants. Overall, we required the execution of more than one million of simulations to accomplish this experiment.

Figures 9, 10, 11 and 12 show the effectiveness of each MR to detect faults in each memory management algorithm. For the sake of clarity, the results of each experiment are depicted in two charts. The chart on the top provides the results in a compact form, showing the percentage of unsuccessful test cases (i.e., those that do not discover an error). The chart on the bottom shows a detailed view of each test case execution, which are numbered from 0 to 499. If the execution of a test case over a mutant satisfies the MR, the mutant is kept alive, which is represented in green. On the contrary, if the execution of a test case over a mutant does not fulfil the MR, a fault is detected and the mutant becomes killed, which is shown in red. In both charts, the x-axis shows the MRs involved in the testing process, where each MR is divided in 5 columns representing the generated mutants.

Table 4 shows the effectiveness of each MR for detecting faults in the tested memory system, that is, the percentage of test cases that do not fulfil the MR. To represent these results, we use the following notation: M_i^{sys} denotes the generated mutant i from the system sys; M_{Avg}^{sys} is the average effectiveness of each MR for checking all the mutants of the system sys (5 mutants are

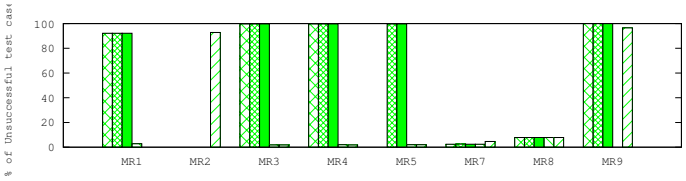


(a) Percentage of unsuccessful test cases (tests not discovering errors)

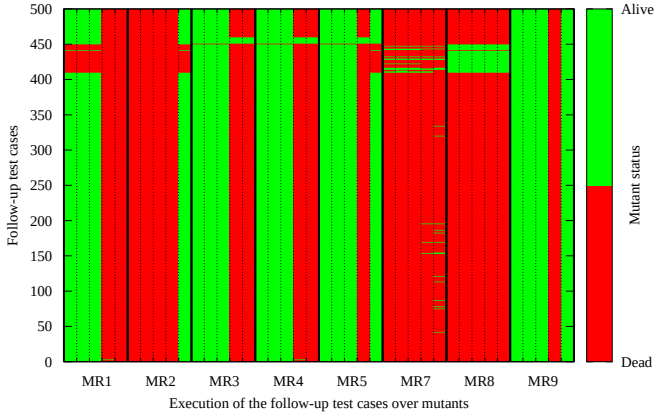


(b) Detailed results of the testing process

Figure 9: Effectiveness of the proposed MRs for checking Close Page Policy

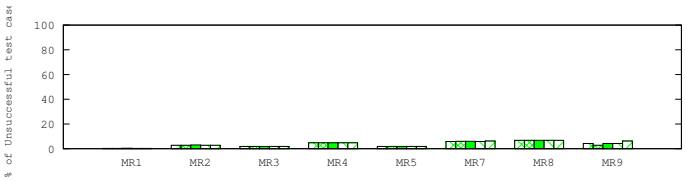


(a) Percentage of unsuccessful test cases (tests not discovering errors)

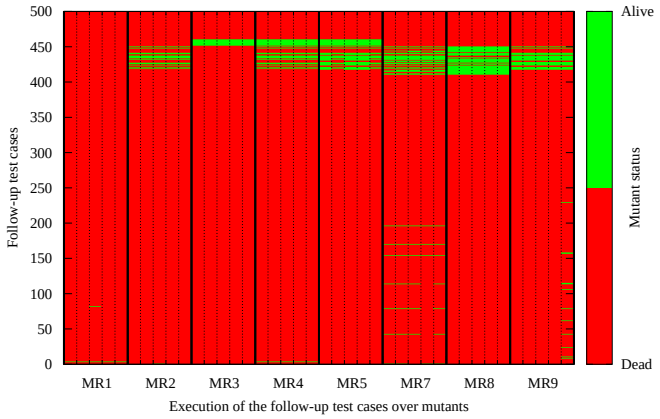


(b) Detailed results of the testing process

Figure 10: Effectiveness of the proposed MRs for checking First Come, First Serve

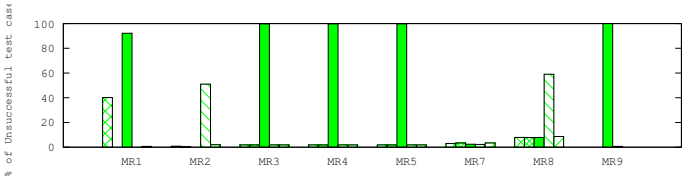


(a) Percentage of unsuccessful test cases (tests not discovering errors)

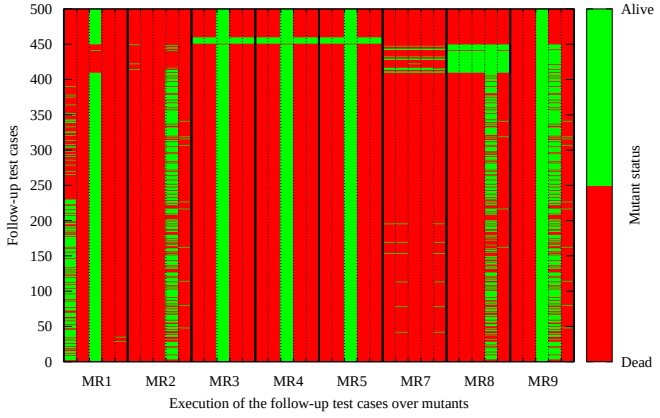


(b) Detailed results of the testing process

Figure 11: Effectiveness of the proposed MRs for checking High Performance Memory Access



(a) Percentage of unsuccessful test cases (tests not discovering errors)



(b) Detailed results of the testing process

Figure 12: Effectiveness of the proposed MRs for checking Request Density Aware Fair Memory

Id	MR_1	MR_2	MR_3	MR_4	MR_5	MR_7	MR_8	MR_9	C_{13}	C_{45}	C_{47}
M_1^{Cpp}	99.8	100.0	98.2	98.0	100.0	97.2	92.2	100.0	100.0	98.2	100.0
M_2^{Cpp}	7.8	3.4	0.2	0.2	0.2	97.2	92.2	0.0	9.8	0.2	100.0
M_3^{Cpp}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_4^{Cpp}	59.2	17.8	98.2	98.0	98.2	96.8	92.2	98.0	59.39	98.2	100.0
M_5^{Cpp}	99.8	100.0	98.2	98.0	98.2	96.2	92.2	100.0	100.0	98.2	100.0
M_{Avg}^{Cpp}	54.8	64.2	59.0	58.8	59.0	97.0	92.2	59.6	55.8	59.0	100.0
M_1^{Fcfs}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_2^{Fcfs}	7.8	100.0	0.2	0.2	0.2	97.39	92.2	0.0	9.8	0.2	100.0
M_3^{Fcfs}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_4^{Fcfs}	99.8	100.0	98.2	98.0	98.2	97.6	92.2	100.0	100.0	98.2	100.0
M_5^{Fcfs}	100.0	7.8	98.2	98.2	98.2	95.4	92.2	3.4	100.0	98.2	100.0
M_{Avg}^{Fcfs}	44.64	81.56	39.4	39.35	39.4	97.12	92.2	20.68	45.87	39.4	100.0
M_1^{Hpma}	99.8	97.2	98.2	95.20	98.2	94.2	93.2	95.8	100.0	95.4	100.0
M_2^{Hpma}	99.8	97.2	98.2	95.20	98.2	94.0	93.2	97.2	100.0	95.4	100.0
M_3^{Hpma}	99.6	97.0	98.2	95.20	98.2	94.0	93.2	95.8	99.8	95.4	100.0
M_4^{Hpma}	99.8	97.2	98.2	95.20	98.2	94.8	93.2	95.8	100.0	95.4	100.0
M_5^{Hpma}	99.8	97.2	98.2	95.20	98.2	93.8	93.2	93.8	100.0	95.4	100.0
M_{Avg}^{Hpma}	99.75	97.15	98.2	95.20	98.2	94.15	93.2	95.67	99.96	95.4	100.0
M_1^{Rdaf}	59.8	99.2	98.2	98.2	98.2	97.0	92.2	100.0	100.0	98.2	99.8
M_2^{Rdaf}	100.0	99.8	98.2	98.2	98.2	96.6	92.2	100.0	100.0	98.2	100.0
M_3^{Rdaf}	7.8	100.0	0.2	0.2	0.2	97.6	92.2	0.0	9.8	0.2	100.0
M_4^{Rdaf}	100.0	39.0	98.2	98.2	98.2	97.79	41.0	99.4	100.0	98.2	100.0
M_5^{Rdaf}	99.4	98.0	98.2	98.2	98.2	96.6	91.39	100.0	100.0	98.2	100.0
M_{Avg}^{Rdaf}	73.4	87.2	78.6	78.6	78.6	97.12	81.79	79.88	81.96	78.6	99.96
Total_{avg}	68.14	82.52	68.8	67.98	68.8	96.34	89.84	63.95	70.90	68.1	99.99

Table 4: Effectiveness (in %) of each MR for detecting faults in memory scheduling systems

involved); $Total_{avg}$ is the average effectiveness of each MR for finding faults in all the generated mutants (20 mutants are involved). The first column of this table refers to the involved mutant(s) for calculating the effectiveness value. The next 8 columns represent the effectiveness of each MR to detect faults.

In general, the proposed expert system detects all the generated mutants, which represent faulty memory systems. However, each MR provides a different effectiveness, which depends on the memory management system under test. Let us remark that a low percentage of unsuccessful test cases in the MRs (top chart in Figures 9, 10, 11 and 12) represents a high effectiveness to detect faults.

Figure 9 shows the results for finding faults in the CPP algorithm. In this case, the most effective relations are MR_7 and MR_8, which have detected faults in the major part of the follow-up test cases. This is depicted in Figure 9.a, where these MRs show a low percentage of unsuccessful test cases, achieving an effectiveness of 97% and 92.2%, respectively. The rest of the MRs provide a lower effectiveness, which ranges from 54.8% to 64.2%.This fact is reflected in the columns that contain both green and red lines (see Figure 9.b). It is important to note that MR_9 is not able to detect mutants 2 and 3.

Figure 10 shows the results obtained for checking the FCFS algorithm. While MR_2, MR_7 and MR_8 achieve an effectiveness of 81.56%, 97.12% and 92.2%, respectively, the rest of the

MRs provides a significantly lower effectiveness, ranging from 20.68% to 44.64%. Similar to the previous analysed algorithm, MR_9 does not detect mutants 1, 2 and 3.

Figure 11 shows the results for checking the HPMA algorithm. In this case, only a reduced number of test cases executed over the mutants have satisfied the MRs, which are represented with green stripes. Consequently, these MRs achieve, in average, an effectiveness of 96.44% for detecting all the mutants in this system.

Figure 12 depicts the results for checking the RDAF algorithm. In this case, the most effective relations are MR_2, MR_7 and MR_8 achieving an effectiveness of 87.2%, 97.12% and 81.79%, respectively. However, in general, all the MRs provide acceptable results, achieving an effectiveness that ranges from 73.4% to 97.12%. In this algorithm, mutant 3 seems hard to kill, it is not detected by MR_9 and it is barely detected by MR_1, MR_3, MR_4 and MR_5.

It is important to mention that the MRs that provide the lowest average effectiveness values for detecting faults, have difficulties to kill mutants that have been created by modifying statements based on the read/write queue. In order to alleviate this drop in the average effectiveness, we have generated 3 new MRs by composing different MRs from Section 5. This way, C_{13} refers to the composition of relations MR_1 and MR_3, C_{45} refers to the compositions of MR_4 and MR_5, and C_{47} refers to the composition of MR_4 and MR_7. The idea is to complement each MR for detecting faults in a greater number of test cases than using a single MR. The testing process has been repeated by using the new generated MRs (see the last three columns of Table 4). The results obtained for C_{13} and C_{47} have achieved a better effectiveness than each single MR, reaching an average effectiveness of 70.9% and 99.99%, respectively. In this case, these new MRs provide an accurate model to represent the behaviour of the system under test and, therefore, the results obtained are promising. However, C_{26} achieves a lower effectiveness than each single MR, that is, MR_2 and MR_6. Thus, we observe that, in this case, it is not practical to merge these MRs because the conditions inside the relations (MR_i, MR_o) are not compatible for combination to accurately model the system under test.

7.4. Comparison with current standard methods for testing memory systems (RQ3)

This section presents a study that compares our proposed expert system with a current standard method for testing memory systems. To the best of our knowledge, expert systems have not been applied to test memories in the past. Thus, since the main objective of this work is to automatically test memory systems using a wide range of configurations, we think the most appropriate method to be used in this study is random testing, which despite its simplicity, is widely used in computer systems and applications Hassan et al. (2015); Rixner et al. (2000); Arcuri et al. (2012); Ciupa et al. (2011).

We are particularly interested in investigating the effectiveness of random testing to detect faults in memory systems. Hence, we have carried out an experiment where 1000 test cases are randomly generated for testing the four different memory scheduling algorithms analysed in previous sections (CPP, FCFS, HPMA and RDAF). Again, for each algorithm, the same five faults were artificially injected. Similarly, other 1000 test cases have been generated, using our proposed system, to detect the same faults. The idea is to compare the effectiveness to detect the injected faults of our system against the one reached by random testing.

Table 5 shows the results of this study, where each column - namely $Fault_1$ to $Fault_5$ - represents the different injected faults in the memory schedulers. Each row represents the system under test (in short, SUT) and consists of two different values, where *valid* shows the percentage of test cases that are successfully executed and *detect* is the percentage of *valid* test cases that

detects the fault. For instance, in order to test the CPP scheduler, random testing generates 31 valid test cases – from the 1000 test cases generated in total – where only 58.06% of these test cases (18 in total) are able to detect $Fault_1$. In this experiment, our system (in short, ES) executes the testing process using C_{47}, which is the best combination obtained in the previous study (see Section 7.3). Random testing (in short, Rnd) does not apply constraints to generate the test cases.

SUT		$Fault_1$		$Fault_2$		$Fault_3$		$Fault_4$		$Fault_5$	
		Rnd	ES	Rnd	ES	Rnd	ES	Rnd	ES	Rnd	ES
CPP	Valid	3.10	99.20	2.90	99.60	3.00	99.20	3.10	99.01	3.20	99.41
	Detect	58.06	100	89.65	100	90	100	48.38	100	41.93	100
FCFS	Valid	3.10	99.10	2.90	99.80	2.90	99.65	3.00	99.90	3.10	99.10
	Detect	90.32	100	100	100	89.65	100	60.00	100	100	100
HPMA	Valid	3.00	99.6	2.90	99.5	3.10	99.40	3.10	99.5	3.10	99.6
	Detect	60.00	100	27.58	100	58.06	100	60.00	100	19.35	100
RDAF	Valid	3.00	99.6	2.90	99.5	3.10	99.4	3.10	99.5	3.10	99.6
	Detect	20.00	99.8	37.93	100	90.32	100	67.74	100	27.58	100

Table 5: Effectiveness (in %) of random testing vs. our proposed system using composed MRs

In general, our proposed system provides better results than random testing. These results show that, when random testing is applied to generate the test cases, there is a considerable number of them that are not valid. In some cases, a test case contains a wrong hardware configuration of a memory system that cannot be simulated like, just to name a few, wrong number of channels, wrong size and non-compatible frequencies and, therefore, only valid test cases can be executed to test the memory. Additionally, we observe that, in the major part of the cases, our system clearly outperforms random testing. There are few cases where random testing provides the same results – in percentage of test cases – to detect the fault. However, it is important to remark that our proposed system generates quality test cases, in the sense that almost all the generated cases are valid, for testing memory systems, which increases the overall performance of the testing process. Also, using combined MRs clearly provides the best results for testing memory systems achieving, in the worst case scenario, 99.8% of effectiveness to detect faults.

Next, we compare the effort required to carry out the testing process. Using random testing requires relatively few effort to generate and execute the test cases. Basically, the tester only has to identify the input parameters that must be randomly generated to create the test suite. However, random testing requires the manual creation of an oracle, or manually checking the provided outputs, which is a very time consuming task.

Similar to random testing, our system only requires effort for preparing the test cases if the MRs do not exist yet. In that case, the method to generate the test suite must be adapted for the involved MRs. However, in contrast to random testing, our expert system uses the constraints defined in the MRs to automatically generate the test cases and, therefore, the major part of the test cases are valid memory configurations. Our method solves the oracle problem, because once the test cases are executed, our system is able to automatically check the provided outputs using the defined MRs. Hence, checking whether a test is successful is automatic in our case.

In conclusion, random testing provides a lower effectiveness to test the memories than our proposed system and requires a considerable effort to check the provided results. Hence, we think that our proposed system is a valuable contribution for the research community, not only for automatically generating quality test cases, but to alleviate the oracle problem, eliminating the effort of the tester to check the provided output.

7.5. Discussion of the results

In this section, we answer the research questions using the results obtained from the previous experiments.

In order to answer **RQ1:** "*Are the designed MRs suitable to be used as rules in the knowledge base?*", we use the results described in Section 7.2. These results were obtained by using the USIMM simulator and 4 different memory management algorithms, which we assumed correct. Table 2 summarises these results, which clearly show 2 different behaviours for each MR, that is, all the tests satisfy the MR or, by the contrary, none of the test cases satisfy the MR. It is interesting the fact that exactly the same results are obtained for the 4 memory management algorithms. However, as we described in Section 7.2, these results are mainly produced by the shortcomings of the USIMM simulator Chatterjee et al. (2012) and, consequently, we decided to remove these MRs from the KB. Since the 100% of the test cases satisfied the rest of the MRs, we assume that these are suitable to be used as rules in the knowledge base.

Next, we answer the research question **RQ2:** "*How effective is the proposed expert system to catch errors in faulty memory systems?*", by using the results obtained in Section 7.3, which are summarised in Table 4. In general, those rules representing critical aspects of the system provide promising results. Relations MR_2, MR_7 and MR_8 achieve by far the best results, reaching an average effectiveness of 82.52%, 96.34% and 89.84%, respectively. Those MRs focusing on general aspects of the system provide acceptable results. In this case, relations MR_1, MR_3, MR_4 and MR_5 exceed 63% of average effectiveness in all these cases. However, since MR_9 is not able to detect several mutants in 3 of the previously analysed algorithms, we can state that this a useless MR to detect those kind of faults. Also, we can observe a correlation between the complexity of the analysed scheduling algorithms and the effectiveness of each MR. The two first algorithms, FCFS and CPP, are less complex than HPMA and RDAF. However, the results show that the analysed MRs provide the best effectiveness in those algorithms that have a high complexity. Since HPMA and RDAF contain sensible parts, a small change in those algorithms generates an unexpected behaviour that is easily detected by the MRs. On the contrary, basic algorithms contain less parts of code that are sensible to produce unexpected behaviour and, consequently, the seeded errors are more difficult to detect. Additionally, we have identified some MRs that detect similar errors. For instance, MR_1, MR_3, MR_4 and MR_5 provide promising results for detecting errors focusing on read/write delays. However, since these MRs focus on general aspects of the system, these do not provide specific information to locate the error. Hence, we conclude that the proposed expert system is effective to detect anomalies in the memory management system, providing better results in complex algorithms.

To answer **RQ3:** "*How suitable is the proposed expert system to test memory systems compared with standard methods?*", we have carried out an experiment for comparing our proposed expert system with random testing to test faulty memory systems. We observe that random testing requires few effort to generate and execute test cases. However, it suffers from the oracle problem, because the tester has to manually check the provided outputs, which is a tedious and error prone task. This is alleviated by our approach, which allows to automatically execute the testing process. The results obtained using our approach are better than the ones obtained by random testing. First, we show that our approach is more efficient to create quality test cases. Second, we obtain the best effectiveness to detect faults when the system combines different MRs to execute the testing process, which clearly outperforms the effectiveness provided by random testing. Hence, the answer to this question is that our proposed system is fairly suitable to test memory systems because, in contrast to other standard techniques, the testing process is automatically executed obtaining better effectiveness.

After a careful analysis, we can conclude that the results provided by the experiments carried out in the empirical study are promising. Our system is not only able to detect faults in the memory system, but also to show those bugs that are more complicated to detect, which is calculated using the effectiveness of each MR. It is important to note that an accurate design of the MRs is key for detecting errors. In this work MR_2, MR_7 and MR_8 are able to detect the major part of the faults. We have also investigated the impact of combining different MRs for detecting faults, showing that the right combination of MRs increases the system effectiveness. However, using wrong combinations provides worse results than using MRs individually. Finally, we found that the expert has a significant impact on the system performance, that is, providing an accurate design of MRs and properly combining MRs is directly reflected in the overall system effectiveness.

The main strengths of our proposed frameworks focuses on the two problems previously described: the oracle problem and the reliable test set problem. On the one hand, we show that the proposed expert system is able to automatically test a wide range of memory configurations using large test suites. Thus, the tester is able to execute the testing process without manually checking the result of each test. On the other hand, the novel design of the proposed expert system, which includes in its core a data-base, a simulator and a collection of MRs, allows automatically generating appropriate test suites. We show that the generated tests are able to identify bugs in different memory scheduling algorithms. However, our proposed system also has some limitations. We think that the most relevant weakness of our proposed system lies in the design of the MRs, which must be manually designed by the tester. These MRs are integrated in the core of the expert system and are used both to generate the test cases and to automatically check the output provided by the tests execution. Hence, providing accurate and appropriate MRs is key for the proper functioning of the system. A challenge to the community is the automatic discovery of MRs from the observation of execution traces using e.g., machine learning techniques Kanewala et al. (2016). A second limitation is that the workload used to test the memory system must be representative. That is, using small and non-realistic workloads might not be suitable for accurately testing the memory system. In this work we alleviate this issue using a wide spectrum of workloads inspired by the PARSEC benchmarks Bienia et al. (2008).

7.6. Threats to validity

In this section, we discuss the threats to validity of our empirical study.

7.6.1. Internal threats

Internal validity is concerned with whether our findings (based on the obtained results from the empirical study) truly represent a cause-and-effect relationship. Thus, the internal validity of our study lies in the implementation of our experiments.

The design of the KB is based on the experience of two experts. We are aware that the ability of the expert system to detect errors highly depends on the selection of metamorphic properties and, therefore, the results may have varied if different MRs were used to build the KB. However, the use of domain-specific properties, like the ones used to design our proposed catalogue of MRs, should reveal a high percentage of failures Xie et al. (2009).

We have implemented the MRs in Java and used USIMM to simulate a wide spectrum of scenarios to obtain the results. These results are used to check if the MRs are fulfilled, or not. We have conducted code inspection and run different tests by hand to assure the correctness of

these implementations. Moreover, the source code has been checked by different individuals. Our evaluation of the MRs is based on the randomly generated inputs, that is, the source test cases. Similarly, the follow-up test cases have been generated by using random values and the corresponding constrains to assure the relation between the source test case and the generated one is fulfilled.

The chosen mutation operators could be another threat to internal validity. Different operators and hand-seeded faults may produce different mutants. However, we carefully designed the mutation operators by investigating, from different sources, common faults produced by programmers, including works in the current literature, repositories of different simulators, "whats new" logs and mailing lists.

Other issues might arise due to the simulator used. This might have errors that can affect our findings. The USIMM simulator, which represents the behaviour of different scenarios of memory systems to execute the tests, has been widely used by the research community. Moreover, this simulator has been in the Memory Scheduling Championship Chatterjee et al. (2012). We mitigate this threat with the experiment described in Section 7.2, where a broad range of test cases, involving 500 different memory models, were executed and checked over our proposed MRs.

7.6.2. External threats

External validity is concerned with the extent to which the results of a study can be generalised.

We have used 500 different memory configurations and 50 different workloads, inspired by the PARSEC benchmarks. Although we believe that these models represent a broad range of memory configurations, there is no guarantee that the obtained results and the achieved improvements of effectiveness of the MRs are the same for other scenarios.

We have chosen four memory management algorithms to conduct our empirical study. The purpose of choosing these algorithms is to analyse planners with different degrees of complexity. While two of these algorithms are relatively simple, FCFS and CPP, the complexity of the other two algorithms, HPMA and RDAF, is significantly higher. However, more experiments would be needed to fully warrant the generalisation of our proposal.

7.6.3. Construct threats

Construct validity is concerned with whether the used measures are representative or not.

We measured the quality of the expert system based on its fault-detection effectiveness, which is also widely used in the community.

Defects in the simulator or in our proposed expert system could be a threat to construct validity. We controlled this threat by executing a wide spectrum of test cases, using four different memory management algorithms, over the USIMM simulator. After this experiment, we removed 2 MRs from the KB because we detected a limitation in the simulator, which does not properly represent the properties reflected in the discarded MRs. Hence, we check that the MRs were properly designed and that our implementation worked correctly.

8. Conclusions

In this paper we have presented an expert system for detecting faults in memory systems using simulation and metamorphic testing. For this, we have built the KB using MRs, which address

critical aspects of memory systems, such as functional, performance and energy consumption. Additionally, we use the methodology of metamorphic testing to automatically generate test cases, which are simulated in a memory simulator.

To measure the effectiveness of our proposed expert system, we performed an experimental study based on mutation testing. In general, the proposed expert system achieves promising results, detecting the major part of the injected faults. The MRs focused on specific aspects of the system achieve better results than those focused on general aspects. In addition, the composition of MRs outperforms the results of each single MR. A comparison with random testing shows that our approach requires less effort to validate the test results (the MRs solve the oracle problem) while achieving higher effectiveness.

We can conclude that, during the testing process, the role of the expert is of vital importance. First, the expert is in charge of designing the MRs. Second, her decisions to choose those MRs to be combined – and included into the KB – have a direct impact of the final obtained results.

Since the proposed system presents a novel design, which integrates simulation and metamorphic testing in its core, different lines for future work have been opened. First, we plan to develop a new language for designing constraints in memory systems. Thus, new MRs could be easily created by the expert and integrated into the expert system, which will increase the spectrum of tested memory configurations. Using this language, the expert system will be able to automatically check the MRs designed by the expert. The expert might provide wrong MRs that do not accurately represent the behaviour of the memory system and, therefore, an automatic checking would provide a significant value to the system effectiveness. Another line of research is to use machine learning methods to analyse the FDB of performed simulations, and infer possible MRs, in the style of Kanewala et al. (2016).

Additionally, in order to provide a solid contribution to the scientific community, we also plan to create a public repository of MRs where researchers could share and use these relations into their own systems. Hence, using a proper DSL, we will allow not only to design new MRs for testing memory systems, but to share these MRs in the public repository. Finally, due to the widely adoption of cloud services by the research community, the proposed service could be deployed in the cloud as a service, which allows researchers accessing to the proposed system through the Internet and without installing additional software into their computers.

Acknowledgments

This work was supported by the Spanish MINECO/FEDER project DArDOS under Grant TIN2015-65845-C3-1-R, and the *Comunidad de Madrid* project FORTE under Grant S2018/TCS-4314. The first author is also supported by the Universidad Complutense de Madrid - Santander Universidades grant (CT17/17-CT18/17).

References

Abts, D., Jerger, N. D. E., Kim, J., Gibson, D., & Lipasti, M. H. (2009). Achieving Predictable Performance Through Better Memory Controller Placement in Many-core CMPs. In *Proceedings of the 36th International Symposium on Computer Architecture* (pp. 451–461).

Arcuri, A., Iqbal, M. Z., & Briand, L. (2012). Random testing: Theoretical results and practical implications. *IEEE Trans. Software Eng.*, *38*, 258–277.

Awasthi, M., Nellans, D. W., Sudan, K., Balasubramonian, R., & Davis, A. (2010). Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques* (pp. 319–330). ACM.

Balasubramonian, R. (2012). Memory scheduling championship results. `http://www.cs.utah.edu/~rajeev/jwac12/ results_table.html`. Accessed July, 2017.

Barr, M. (2017). EmSA: Products, Consulting & Training for Embedded Systems - Software Based Memory Testing. `http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/software-based-memory-testing.html`. [Online; Accessed on Dec 15, 2017].

Bennett, J., & Hollander, C. (1981). DART: An expert system for computer fault diagnosis. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence-Volume 2* (pp. 843–845). Morgan Kaufmann Publishers Inc.

Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel architectures and compilation techniques* (pp. 72–81). ACM.

Cañizares, P. C., Núñez, A., Núñez, M., & Pardo, J. J. (2015). A methodology for designing energy-aware systems for computational science. In *Proceedings of the 15th International Conference on Computational Science* (pp. 2804–2808). Elsevier volume 51.

Chatterjee, N., Balasubramonian, R., Shevgoor, M., Pugsley, S. H., Udipi, A. N., Shafiee, A., Sudan, K., Awasthi, M., & Chishti, Z. (2012). USIMM: the Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship.

Chen, T. Y., Cheung, S. C., & Yiu, S. M. (1998). *Metamorphic testing: a new approach for generating next test cases*. Technical Report HKUST-CS98-01.

Ciupa, I., Pretschner, A., Oriol, M., Leitner, A., & Meyer, B. (2011). On the number and nature of faults found by random testing. *Software testing, verification and reliability*, *21*, 3–28.

Clarke, E. M., Grumberg, O., & Peled, D. A. (2001). *Model checking*. MIT Press.

C.M. Herrero-Jiménez (2012). An expert system for the identification of environmental impact based on a geographic information system. *Expert Systems with Applications*, *39*, 6672–6682.

Council, J. E. D. E. (2017). JEDEC DDR4 SDRAM Standard, 2012. `https://www.jedec.org/standards-documents/docs/jesd79-4a`. Accessed: 2017-07-30.

de Dinechin, B. D., van Amstel, D., Poulhies, M., & Lager, G. (2014). Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition* (pp. 1–6).

Ding, J., Zhang, D., & Hu, X.-H. (2016). An application of metamorphic testing for testing scientific software. In *Proceedings of the 1st International Workshop on Metamorphic Testing* (pp. 37–43). ACM.

Dreibelbis, J., Barth, J., Kalter, H., & Kho, R. (1998). Processor-based built-in self-test for embedded DRAM. *IEEE Journal of Solid-State Circuits*, *33*, 1731–1740.

Gepner, P., & Kowalik, M. F. (2006). Multi-core processors: New way to achieve high system performance. In *Proceedings of the 5th International Conference on Parallel Computing in Electrical Engineering* (pp. 9–13). IEEE.

Ghasempour, M., Jaleel, A., Garside, J. D., & Luján, M. (2016). DReAM: Dynamic re-arrangement of address mapping to improve the performance of drams. In *Proceedings of the 2nd International Symposium on Memory Systems* (pp. 362–373). ACM.

de Goor, A. J. V., & Smit, B. (1994a). Automating the verification of memory tests. In *Proceedings of the 12th IEEE VLSI Test Symposium* (pp. 312–318). IEEE.

de Goor, A. J. V., & Smit, B. (1994b). Generating march tests automatically. In *Proceedings of the International Test Conference* (pp. 870–878). IEEE.

Gundu, A., Sreekumar, G., Shafiee, A., Pugsley, S. H., Jain, H., Balasubramonian, R., & Tiwari, M. (2014). Memory bandwidth reservation in the cloud to avoid information leakage in the memory controller. In *Proceedings of the 3rd Workshop on Hardware and Architectural Support for Security and Privacy* (pp. 11:1–11:5).

Hardee, K. C., Chapman, D. B., & Pineda, J. (1991). Dynamic random access memory. US Patent.

Hassan, M., & Patel, H. (2017). MCXplore: Automating the Validation Process of DRAM Memory Controller Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, .

Hassan, M., Patel, H., & Pellizzoni, R. (2015). A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium* (pp. 307–316). IEEE.

Hierons, R. M., Merayo, M. G., & Núñez, M. (2010). Mutation testing. In *Encyclopedia of Software Engineering* (pp. 594–602). Wiley-Interscience.

Huang, C.-T., Huang, J.-R., Wu, C.-F., Wu, C.-W., & Chang, T.-Y. (1999). A programmable BIST core for embedded DRAM. *IEEE Design & Test of Computers*, *16*, 59–70.

Hussain, A., S.J. Lee, M.S. Choi, & Brikci, F. (2015). An expert system for acoustic diagnosis of power circuit breakers and on-load tap changers. *Expert Systems with Applications*, *42*, 9426–9433.

Ipek, E., Mutlu, O., Martínez, J. F., & Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. In *2008 International Symposium on Computer Architecture* (pp. 39–50).

J. Bautista-Valhondo, & R. Alfaro-Pozo (2018). An expert system to minimize operational costs in mixed-model sequencing problems with activity factor. *Expert Systems with Applications*, *104*, 185–201.

Jacob, B., Ng, S., & Wang, D. (2007). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc.

Jeong, M. K., Yoon, D. H., & Erez, M. (2012). DrSim: A platform for flexible DRAM system research.

Jiang, M., Chen, T. Y., Kuo, F.-C., & Ding, Z. (2013). Testing central processing unit scheduling algorithms using metamorphic testing. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science* (pp. 530–536). IEEE.

Kanewala, U., Bieman, J. M., & Ben-Hur, A. (2016). Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Softw. Test., Verif. Reliab.*, *26*, 245–269.

Karpovsky, M. G., van de Goor, A. J., & Yarmolik, V. N. (1995). Pseudo-exhaustive word-oriented DRAM testing. In *Proceedings of the 1995 European conference on Design and Test* (p. 126). IEEE Computer Society.

Kayed, M. O., Abdelsalam, M., & Guindi, R. (2014). A novel approach for SVA generation of DDR memory protocols based on TDML. In *Proceedings of the 15th International Microprocessor Test and Verification Workshop* (pp. 61–66). IEEE.

Khalifa, K., & Salah, K. (2015). Implementation and verification of a generic universal memory controller based on UVM. In *Proceedings of the 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era* (pp. 1–2). IEEE.

Kim, Y. (2017). Ramulator: A Fast and Extensible DRAM Simulator. `https://github.com/CMU-SAFARI/ramulator`. [Online; Latest commit on Dec 11, 2016. Accessed on Dec 15, 2017].

Kim, Y., Han, D., Mutlu, O., & Harchol-Balter, M. (2010). ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Conference on High-Performance Computer Architecture* (pp. 1–12).

Kim, Y., Yang, W., & Mutlu, O. (2016). Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, *15*, 45–49.

Kumar, P. (2017). USIMM: the Utah SImulated Memory Module. `https://github.com/pranith/usimm`. [Online; Latest commit on Oct 22, 2014. Accessed on Dec 15, 2017].

Li, Y., Akesson, B., Lampka, K., & Goossens, K. (2016). Modeling and verification of dynamic command scheduling for real-time memory controllers. In *Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium* (pp. 1–12). IEEE.

Liberado, E., ao, F. M., oes, M. S., W.A. de Souza, & Pomilio, J. (2015). Novel expert system for defining power quality compensators. *Expert Systems with Applications*, *42*, 3562–3570.

Lin, W.-F., Reinhardt, S. K., & Burger, D. (2001). Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture* (pp. 301–312). IEEE.

Liu, H., Kuo, F.-C., Towey, D., & Chen, T. Y. (2014). How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, *40*, 4–22.

Liu, H., Liu, X., & Chen, T. Y. (2012). A new method for constructing metamorphic relations. In *Proceedings of the 12th International Conference on Quality Software* (pp. 59–68).

Mahapatra, N. R., & Venkatrao, B. (1999). The processor-memory bottleneck: problems and solutions. *Crossroads*, *5*, 2.

Miyano, S., Sato, K., & Numata, K. (1999). Universal test interface for embedded-DRAM testing. *IEEE design & test of computers*, *16*, 53–58.

Modgil, A., Nitin, & KumarSehgal, V. (2015). Understanding and Analyzing the Impact of Memory Controller's Scheduling Policies on DRAM's Energy and Performance. *Procedia Computer Science*, *70*, 399–406. Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems.

Mukundan, J., Hunter, H., hyoun Kim, K., Stuecheli, J., & Martínez, J. F. (2013). Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems. *SIGARCH Comput. Archit. News*, *41*, 48–59.

Natarajan, C., Christenson, B., & Briggs, F. (2004). A study of performance impact of memory controller features in multi-processor server environment. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture* WMPI '04 (pp. 80–87).

Núñez, A., & Hierons, R. M. (2015). A methodology for validating cloud models using metamorphic testing. *Annals of telecommunications*, *70*, 127–135.

Poremba, M. (2017). NVMain - An Architectural Level Main Memory Simulator for Emerging Non-Volatile Memories. `https://bitbucket.org/mrp5060/nvmain`. [Online; Latest commit on Oct 18, 2017. Accessed on Dec 15, 2017].

Pundir, A. K., & Sharma, O. (2017). CHECKERMARC: A Modified Novel Memory-Testing Approach for Bit-Oriented SRAM. *International Journal of Applied Engineering Research*, *12*, 3023–3028.

Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., & Owens, J. D. (2000). Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)* (pp. 128–138). ACM.

Rogers, B. M., Krishna, A., Bell, G. B., Vu, K., Jiang, X., & Solihin, Y. (2009). Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *SIGARCH Comput. Archit. News*, *37*, 371–382. doi:`10.1145/1555815.1555801`.

Rosenfeld, P. (2017). DRAMSim2: A cycle accurate DRAM simulator. `https://github.com/umd-memsys/`

DRAMSim2. [Online; Latest commit on Nov 8, 2014. Accessed on Dec 15, 2017].

Rosenfeld, P., Cooper-Balis, E., & Jacob, B. (2011). DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, *10*, 16–19.

Sahoo, D., & Satpathy, M. (2016). MSimDRAM: Formal model driven development of a dram simulator. In *Proceedings of the 29th International Conference on Embedded Systems and VLSI Design* (pp. 597–598). IEEE.

Segura, S., Fraser, G., Sanchez, A. B., & Ruiz-Cortés, A. (2016). A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, *42*, 805–824.

Shafiee, A., Gundu, A., Shevgoor, M., Balasubramonian, R., & Tiwari, M. (2015). Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture* (pp. 89–101).

Sherwood, T., Perelman, E., Hamerly, G., Sair, S., & Calder, B. (2003). Discovering and exploiting program phases. *IEEE micro*, *23*, 84–93.

Standard, J. (2012). DDR3 SDRAM Standard. *JESD79-3, JEDEC*, .

Subramanian, L., Lee, D., Seshadri, V., Rastogi, H., & Mutlu, O. (2016). Bliss: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems*, *27*, 3071–3087.

Sudan, K., Chatterjee, N., Nellans, D., Awasthi, M., Balasubramonian, R., & Davis, A. (2010). Micro-pages: Increasing DRAM Efficiency with Locality-aware Data Placement. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* ASPLOS XV (pp. 219–230). ACM.

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, *25*, 465–470.

Wu, C.-F., Huang, C.-T., Cheng, K.-L., & Wu, C.-W. (2000). Simulation-based test algorithm generation for random access memories. In *Proceedings of the 18th IEEE VLSI Test Symposium* (pp. 291–296). IEEE.

Wulf, W. A., & McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, *23*, 20–24. doi:10.1145/216585.216588.

Xie, X., Ho, J., Murphy, C., Kaiser, G., Xu, B., & Chen, T. Y. (2009). Application of metamorphic testing to supervised classifiers. In *Ninth International Conference on Quality Software* (pp. 135–144).

Yang, C.-C., Li, J.-F., Yu, Y.-C., Wu, K.-T., Lo, C.-Y., Chen, C.-H., Lai, J.-S., Kwai, D.-M., & Chou, Y.-F. (2015). A hybrid built-in self-test scheme for DRAMs. In *Proceedings of the 2015 International Symposium on VLSI Design, Automation and Test* (pp. 1–4). IEEE.