McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation

Marco Elver
University of Edinburgh
marco.elver@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

ABSTRACT

The memory consistency model (MCM), which formally specifies the behaviour of the memory system, is used by programmers to reason about parallel programs. It is imperative that hardware adheres to the promised MCM. For this reason, hardware designs must be verified against the specified MCM. One common way to do this is via executing tests, where specific threads of instruction sequences are generated and their executions are checked for adherence to the MCM. It would be extremely beneficial to execute such tests under simulation, i.e. when the functional design implementation of the hardware is being prototyped. Most prior verification methodologies, however, target post-silicon environments, which when applied under simulation would be too slow.

We propose McVerSi, a test generation framework for fast MCM verification of a full-system design implementation under simulation. Our primary contribution is a Genetic Programming (GP) based approach to MCM test generation, which relies on a novel crossover function that prioritizes memory operations contributing to non-determinism, thereby increasing the probability of uncovering MCM bugs. To guide tests towards exercising as much logic as possible, the simulator's reported coverage is used as the fitness function. Furthermore, we increase test throughput by making the test workload simulation-aware. We evaluate our proposed framework using the Gem5 cycle accurate simulator in full-system mode with Ruby. We discover 2 new bugs due to the faulty interaction of the pipeline and the cache coherence protocol. Crucially, these bugs would not have been discovered through individual verification of the pipeline or the coherence protocol. We study 11 bugs in total. Our GP-based test generation approach finds all bugs consistently, therefore providing much higher guarantees compared to alternative approaches (pseudo-random test generation and litmus tests).

1. INTRODUCTION

Shared-memory multiprocessors are now ubiquitous, with programmers being exposed to an increasingly heterogeneous landscape of multiprocessor systems. In order to write correct parallel programs, the programmer must reason in terms of the provided memory consistency model (MCM) [1]. The MCM is a *specification*, providing guarantees about how the underlying memory system should behave.

The relationship between weaker MCMs and processor implementations can be seen as a "chicken-and-egg" problem. While many existing weak MCMs are the product of desired microarchitectural optimizations (MCM formalized after implementation), it is equally desirable that new optimizations do not violate a specified MCM of an *architecture*. For example, write-buffers, in the absence of any other visible optimizations, give rise to Total Store Order (TSO), as in e.g. x86 [2]. On the other hand, recent work proposes designing *consistency directed* cache coherence protocols, e.g. TSO-CC [3] has been designed specifically with TSO in mind.

Problems arise, however, when the programmer believes they are working with a particular model, but the hardware exhibits behaviour weaker than the promised MCM: either the model is incorrect, or the hardware contains bugs – both scenarios are undesirable. Recent work has uncovered problems with deployed CPUs [4], and GPUs [5] using litmus testing. While a proof of MCM correctness of the *functional design implementation* (e.g. cycle accurate model) would provide the highest possible guarantees, unfortunately the complexity to achieve such a feat is usually not cost-effective [6]. Nonetheless, there exists a wealth of literature on memory system and MCM verification, which all help to raise the designer's confidence in the implementation.

Various methodologies can be applied at different stages of the design development. In the *pre-silicon* design phase, approaches based on formal methods are usually applied to *abstract models* of *components* of a design. For example, model checking of coherence protocols has been studied extensively [7, 8, 9, 10, 11, 12, 13, 14]; consistency properties verified are commonly based on derived properties, such as the Single-Writer–Multiple-Reader (SWMR) [15] invariant. Another example is the recently developed PipeCheck [16] tool, which is a domain-specific MCM model checker for pipeline abstractions. Applying any of these techniques to as many components as possible is essential, to avoid an implementation based on faulty component specifications.

In the final design implementation, however, the composition and interaction of the components must remain safe with respect to the MCM. With individual component verification, this is often overlooked. For example, one of the bugs discovered in our evaluation (MESI,LQ+IS,Inv[1]) would not

[1]The coherence protocol fails to forward invalidation to the Load Queue (LQ), leading to reordered reads (§5.3).

have been discovered through individual verification of either pipeline or coherence protocol.

Consequently, *full-system MCM verification* is required. This has arguably been achieved in the *post-silicon* environment using a testing based approach [17, 18, 19, 20, 21, 22]. Tests consist of threads of instruction sequences, which are executed in the full-system and their results checked for adherence to the MCM. There are various approaches to test generation, ranging from random [20] to user-directed [17]. Post-silicon approaches can afford to execute large tests, as instruction throughput is much higher than in simulation. *Yet, while all these approaches could be made to work in a pre-silicon environment, i.e. full-system simulation, they would be too slow as the above approaches do not optimize test generation for simulation.*

Problem Statement: Simulation of a design is available much earlier in the development cycle (pre-silicon). As such it is much cheaper if as many bugs as possible are found early. Furthermore, the added observability in simulation makes debugging more straightforward. For example, the advantages of simulation for memory system verification using user-guided random tests have been described and exploited by Wood et al. [23].

Unfortunately, throughput (in terms of instructions executed in wall-clock time) of an accurate simulated system is orders of magnitude lower than a real chip. The challenge is, *how do we automatically generate efficient MCM tests for simulation, such that the wall-clock time to explore rare corner cases and find bugs is reduced?*

Approach and Contributions: Any verification approach strives to ensure an implementation's adherence to its high-level specification. Test based methods trade off a non-exhaustive (reduced states and transitions covered) result for a more detailed implementation. Therefore, the goal of any test based method should be to *cover* as many states and transitions as possible, in order to provide the highest possible guarantees about the system in the absence of a proof [6, 24]. To achieve this, we develop an approach to automatically improve test suitability for exposing MCM violations, and guide tests towards unexplored states and transitions.

Our focus lies on automated *simulation-based verification of a full-system design implementation*: we propose McVerSi, a test generation framework for *fast, coverage directed MCM verification in simulation*. Using a Genetic Programming (GP) [25] based approach, we show how to generate tests for a full-system simulation that achieve (§3): ① greater coverage of the system, and ② improved test quality specifically for MCM verification.

To achieve ①, we leverage the additional observability available in simulation (white-box), and use coverage as the GP *fitness function*. The designer of a system can select any number of suitable coverage metrics; in our implementation we use the covered logic implementing the coherence protocol as the coverage metric (structural coverage), as the coherence protocol is crucial in enforcing the desired MCM, and is the source of some of the most elusive bugs. To achieve ②, we design a domain-specific GP *crossover function*. Our crossover is selective on memory operations contributing to high non-determinism of a test; highly deterministic tests are uninteresting from a MCM verification point of view, as few execution witnesses are invalid, and the probability of observing an invalid witness due to a bug is low.

The tests generated are compiled on-the-fly to the target ISA, which are then executed in the full-system. For a GP-based approach to quickly converge towards better tests, high test throughput is essential. Therefore, instead of large tests typically used with random tests (e.g. TSOtool [20] shows results for $\geq$12k operations), we are interested in very short tests (e.g. in our evaluation we use 1k operations). Thus, the time from one test to the next must be minimized (overhead for checking, test generation, synchronization). To accelerate test execution, we introduce several extensions that any simulator to be used for verification must provide (§4). In particular, a host interface for the simulation-aware guest control program for: configuring the test generator, emitting code on-the-fly, checking, and host-assisted barriers.

We evaluate (§5) McVerSi using the Gem5 [26] cycle accurate simulator in x86-64 full-system mode with Ruby. We found 2 new bugs in the Ruby MESI implementation of Gem5[2]. In total we study 11 bugs. In comparison with a pseudo-random test generator (utilizing the test generation independent (§4) parts of our framework), and diy [17] generated litmus tests for TSO, our GP-based approach finds all bugs consistently within practical time bounds, thereby providing much higher bug finding guarantees.

2. BACKGROUND

This section first provides an overview about memory consistency models (MCMs) with a focus on checking execution traces for violations (§2.1). This is followed by an overview of evolutionary algorithms and their use in microprocessor verification (§2.2).

2.1 Memory Consistency Models

Programming shared memory multiprocessor systems correctly requires a precise definition of the semantics of such a system. In particular, the programmer must be aware of the memory access ordering guarantees the hardware provides. The memory consistency model (MCM) *formally specifies* the ordering guarantees with which the programmer can reason about parallel programs [1]. In this paper, we are concerned with *system-centric consistency models* [27], i.e. hardware MCMs. Over the years, various formalizations of MCMs have emerged. Both *axiomatic* – e.g. Sequential Consistency (SC) [28], Total Store Order (TSO) [4, 29], Release Consistency (RC) [30], POWER [17, 4, 31] and ARM [4] – as well as *operational* – e.g. x86-TSO [2] and POWER [32]) – models can be used to describe MCMs formally.

Axiomatic style models define an MCM in terms of *constraints* over *candidate executions*, effectively limiting the set of valid executions; stricter MCMs reduce the set of valid executions (strictest being SC, with nothing reordered), whereas weaker models are more permissive. An execution in an axiomatic model is defined abstractly in terms of each thread's *program order* (po) and *conflict orders*; conflict orders in turn consist of *read-from* (rf) (ordering write→read pairs, the write providing the value for the read), and *coherence order* (co) (ordering write→write pairs, the first write serialized

[2]Fixes for the bugs have been sent to the Gem5 maintainers.

before the second) relating memory operations with the same address. Each such order is a relation over *events*. Events are memory operations (reads and writes) associated with concrete instructions. Each memory instruction is typically associated with a unique event, with the exception of e.g. read-modify-write instructions which map to two events. The *architecture* defines the specific constraints that must be satisfied by an execution, as well as derived relations that tell the programmer which orders are guaranteed to be enforced; in particular the *preserved program order* (ppo) is a subset of po which captures ordering guarantees of the hardware. Each constraint typically expresses that a derived relation is acyclic or irreflexive. For a complete description of the terminology we use in this work, we refer the reader to [4].

At the core of an axiomatic model checker (of an execution) is a graph-search algorithm, which is used to construct all required derived relations and then assert all constraints over these are satisfied. The complexity of checking a candidate execution against particular axiomatic models has been the primary concern of many post-silicon verification works [20, 21, 22]. Unlike post-silicon, however, a pre-silicon environment can afford a straight-forward, complete and polynomial-time decision procedure as all conflict orders are visible [33].

On the other hand, operational models are defined in terms of an abstract machine, which given a read, then specifies the set of possible values a read may observe. Their complexity advantages for simulation-based verification have been realized in past works [34, 35], where such models are also referred to as *relaxed scoreboards*. Each transition is monitored by a checker, effectively ensuring that the simulated system only performs transitions which are legal according to the model.

In this work we describe MCMs in terms of axiomatic models (§4.1), as simulation together with short tests (§4) affords an efficient and relatively simple checker. Note, however, our main contribution concerns MCM test generation.

2.2 Evolutionary Algorithms

Evolutionary algorithms are heuristic search algorithms, a machine learning approach inspired by the principles of natural selection. Genetic Algorithms (GAs) [36] are one such approach, with a broad range of optimizations problems where GAs have proven to be practical solutions [37]. The goal is to iteratively evolve an initially random population of *chromosomes* (or genomes) towards increasingly optimal solutions. In GAs, each chromosome is usually represented as a fixed-size string, which encodes values of the parameters to be searched. Each solution has a *fitness* value, determined via a domain-specific fitness-function. Based on the fitness values, *selection* then chooses several solutions to be used to generate new offspring. *Crossover* and *mutation* are the operators used to create offspring from the selected parents, with crossover choosing parts of each chromosome to be recombined into one or more children, and mutation selecting few individual genes to be modified.

Genetic Programming (GP) [38, 25] is an adaption of GAs, that instead of searching for strings of parameters, search for actual *executable programs* which yield *executable solutions* to the search problem. The general approach is like in GAs, but the representation and crossover of chromosomes is specialized to yield valid programs in the language and domain being targeted.

Machine learning approaches have been successfully applied to generate successively better tests to increase coverage across a wide range of microprocessor verification scenarios [24]. For example, GAs have been used to search for biases for pseudo-random test generators [39]. Using GP, *μGP* [40] has been proposed to generate test programs directed by various coverage metrics. In this work we use a GP test generation approach, similar to *μGP*, but with a focus on multi-threaded test generation for the purpose of MCM verification.

3. TEST GENERATION

This section describes the proposed automated test generation approach, whose goal is to reveal as many MCM bugs as fast as possible. Section §3.1 provides an overview; §3.2 discusses in more detail the mapping of coverage to fitness; and finally §3.3 describes test representation, crossover and mutation operators.

3.1 Overview

Given pseudo-randomly generated tests (instruction sequences), with some constraints given by the user (distribution of operations, memory address range, and stride), *how can the test generator improve tests without further user input?* Coverage, which refers to the fraction of system state explored, is a widely used metric to assess test quality [6, 24, 39, 40], giving an indication of how close the verification task is to completion. Over time, the test generator's goal should therefore be to direct tests towards rarely covered state transitions based on coverage feedback. In the absence of further information about the implementation, apart from coverage reports, the only input we will give the simulated system are instruction sequences. Finding a precise solution to cover rare state transitions given this degree of control is a complex problem, and approximate solutions based on *evolutionary algorithms* (see §2.2) have been used successfully.

McVerSi uses Genetic Programming (GP) [25, 38] based test generation. Tests (chromosomes) are *represented* as directed acyclic graphs (DAGs) of operations [40]. Each node (gene) represents a high-level operation of a thread; each operation in turn, maps to an executable representation in the target ISA. A *test-run* corresponds to executing the test for several iterations; after a test-run completes, the fitness of the run is evaluated and associated with the test.

As the goal of the test generator is to generate tests covering as many states and transitions of the system as possible, the *fitness function* is defined in terms of coverage. For the purpose of MCM verification, all crucial bits of logic affecting enforcement of the MCM should be captured by coverage. This, by and large, means the processor pipeline, coherence protocol and on-chip interconnect. In our implementation, we restrict coverage to structural (code) coverage of the coherence protocol, as the most challenging bugs we study are related to the coherence protocol; having said this, our framework is not tied to this choice and it is indeed possible to augment coverage with functional coverage metrics (e.g. store buffer becoming full). Our coverage computation dynamically adapts such that frequent state transitions are

init: x = 0, y = 0	
Thread 1	**Thread 2**
$x \leftarrow 1$	$r1 \leftarrow y$
$y \leftarrow 1$	$r2 \leftarrow x$

Figure 1: Message passing example.

excluded from coverage, so that the focus shifts towards rare protocol transitions. In other words, the GP verification goals change over time.

However, in order to be able to detect MCM bugs in the first place, we require tests which are more likely to expose MCM violations; we will refer to this as *MCM test suitability*. This means, we seek tests where a large fraction of possible candidate executions are invalid under the specified MCM, to increase the probability of observing an invalid candidate execution due to a possible bug. To illustrate, Figure 1 shows the common message passing litmus test. Assuming e.g. a TSO model, the outcome $r1 = 1 \land r2 = 0$ is forbidden. If, however, we were to remove either write of x or y, all candidate executions become valid – such a test would not be very useful from a MCM verification perspective. While there are several ways to increase the probability of generating suitable MCM tests (e.g. constrain the usable address range), this would preclude us from generating tests which could expose bugs requiring large address ranges (e.g. due to cache evictions).

In order to be able to converge towards more suitable tests, first we must be able to tell how suitable a given test is. Given that our tests far exceed the size of litmus tests, it would be too costly to enumerate all possible candidate executions of a test in order to determine the set of valid and invalid executions. Instead, we observe that tests with large number of candidate executions are highly non-deterministic/racy. Therefore, to generate more suitable MCM tests, we should favour such tests. The key metric we introduce is the *average non-determinism of a test* ($\mathsf{ND_T}$), which informally is a measure of average number of races observed per event (memory operation) across all iterations of the test-run. More precisely, it is the average number of events that are *conflict ordered before* any given event in the test[3]. A value of 1 means the test-run is not observed to be non-deterministic/racy, i.e. all events are only ordered after the initial events (e.g. "init" in Figure 1). A $\mathsf{ND_T}$ value larger than 1 implies that races have been observed.

Definition 1. Let i be the iteration in a test-run. The simulator records the conflict order relations rf_i and co_i (defined in §2.1) for each iteration. Then we define the *union of all iteration's observed conflict orders* to be

$$\mathsf{rfco_{RUN}} = \bigcup_i (\mathsf{rf}_i \cup \mathsf{co}_i)$$

Definition 2. Let n be the *total events (memory operations) executed* in a test. We define the average non-determinism in a test as the cardinality of $\mathsf{rfco_{RUN}}$ divided by n

$$\mathsf{ND_T} = \frac{|\mathsf{rfco_{RUN}}|}{n}$$

[3]A prerequisite for this metric to be meaningful, is that a test-run has more than one iteration.

We initially assessed including $\mathsf{ND_T}$ in the fitness function, and using standard GP crossover operators [25, 38]. This, however, did not result in significantly more suitable tests over time. This is because, the non-deterministic result of a test is sensitive to the specific sequences and interaction of instructions: breaking them up without considering the key ingredients to the non-deterministic result caused little progress towards more suitable MCM tests. In other words, merely combining random instructions from two racy tests cannot guarantee a new more racy test – instead tests must be recombined in a way, such that the resulting test is likely to be more racy than its parents.

Instead, to generate more suitable MCM tests, we design a *selective crossover*, which gives preference to memory operations involved in races, i.e. those with observed non-deterministic results across several iterations. More specifically, our goal is to preserve sequences of memory operations on addresses (a key ingredient of MCM tests) which contribute highly towards non-determinism of the test outcomes. For this we measure the *non-determinism of each event (memory operation)* (ND_e), and give preference to those events whose non-determinism is higher than the test's $\mathsf{ND_T}$.

Definition 3. We define the non-determinism of an individual event e_k as the cardinality of the set of events which are ordered before e_k (via conflict order) across a test-run

$$\mathsf{ND}_e = |\{e \mid \forall e \;:\; (e, e_k) \in \mathsf{rfco_{RUN}}\}|$$

Further details of GP parameters, *selection* method and operations used, which are independent of our proposed scheme, are discussed in the evaluation (§5).

3.2 Coverage and Fitness

Coverage gives an indication of how close the verification task is to completion, and ideally lets us judge if all interesting scenarios that we think can lead to bugs have been covered. Therefore, we use coverage as the GP fitness function. Note, the verification scenarios and therefore coverage goals are highly system dependent, and the following is one of many possible options.

In modern multiprocessors, the cache coherence protocol is a key component in the implementation of the MCM [15]. The goal of the coherence protocol is to make caches transparent, such that data inconsistencies (and MCM violations) are not due to accesses to stale cached data. Because the hardest to find bugs we study originate in the coherence protocol, we use structural coverage over the protocol's possible state transitions as the *fitness function*. For our study, we do not distinguishing between identical controllers, and instead consider the sum of their transitions.

Each test fitness is assigned a coverage value independent of any prior run tests, i.e. only what has been covered by a particular test-run. Because the simulation is running continuously, loading new tests on-the-fly, any state changes of previous tests that affect following tests must be reset (e.g. flush caches – see §4) to produce consistent results.

Next, we do not consider all protocol transitions, and instead frequent transitions are excluded from coverage, i.e. we compute an *adaptive coverage*. The goal of this is, since the simulation is running continuously, and thus recording all transitions since simulation start, we can direct the test population towards unexplored and rarer transitions. Effectively, this helps avoid getting a population stuck in a local maximum, where little progress is made towards unexplored states.

Upon initialization we consider those transitions, whose transition counts is less than a low initial cut-off value. If the adaptive coverage falls below a certain threshold for too many test evaluations, the cut-off is doubled (exponential increase). Then, if we consider a total of t transitions, and if in a test run n of these were covered, the fitness of that test would be n/t. Each test's fitness is evaluated only once.

3.3 Test Representation, Crossover and Mutation

Representation: Each test (chromosome) is represented as a DAG of a constant number of nodes (genes), which naturally represents control flow [40] and each *disjoint sub-graph representing one thread.* A sequence of nodes corresponds to the program order of one thread. Each node is a high-level operation (op) of a thread which maps to executable code of the target ISA. Furthermore an op is responsible for the mapping to one or more events in the MCM (only for memory operations) as per the defined instruction semantics.

Nodes are stored internally as a flat list of tuples: each tuple represents $\langle \mathsf{pid}, \mathsf{op} \rangle$, where pid is the processor/thread ID and op is an operation. The order of nodes within this list gives rise to the code sequence of instructions, but not necessarily program order (po), e.g. due to branches. The class and properties of an operation specifies how nodes are connected *upon* code generation, such that copying individual nodes of one thread to another (via crossover), forms another valid thread. The final DAG representation is restrictive enough to allow efficient generation of the static ordering relations of the target MCM, as well as an efficient crossover as described in the following.

Crossover: As outlined in §3.1, we determine that a standard crossover is unsuitable for the problem of generating more suitable MCM tests. We design a *selective crossover*, which selects and merges thread sub-graphs based on operations which highly contribute towards non-determinism of a test. The key metric we introduce to assess a test's degree of non-determinism is the *average non-determinism of a test* ($\mathsf{ND_T}$). After evaluation of a test-run, we obtain its $\mathsf{ND_T}$ (Definition 2) and each event's ND_e (Definition 3). From this, we obtain the set of events' addresses fitaddrs, where an event's *ND_e is larger than the rounded ND_T of the test.* The proposed *selective crossover* then always selects those nodes where the address of a memory operation is a member of the set of addresses fitaddrs.

To be able to place a bound on the simulated execution time of a test, we enforce the number of nodes of a test to be constant. Note, however, that the number of nodes per thread is not necessarily constant. Additionally, in MCM tests we would like to *preserve some of the relative scheduling properties* of test operations; e.g. an operation which is placed at the end of a thread should not be moved to the beginning of a thread in a new test after crossover. During recombination of two tests, the flat list representation of the DAG nodes simplifies enforcing both the above properties efficiently.

Mutation: Following crossover, *mutation* takes place if necessary, which mutates nodes by randomizing thread and operation, but preserving the relative position in the test. As not all operations are necessarily selected from either parent test, missing nodes are generated pseudo-randomly: this step already contributes to mutation, effectively enabling more directed mutation, such that in the early stages of test generation useful sequences of operations are retained.

Summary: Algorithm 1 shows our proposed crossover and

Algorithm 1: Crossover and mutation.

```
Let $P_{MUT}$ be the mutation probability;
Let $P_{USEL}$ be the unconditional mem. op. selection probability;
Let $P_{BFA}$ be the bias with which a new operation has an address from
    the set of fitaddrs;
Let fitaddrs(test) return the set of addresses of events where $\mathsf{ND}_e$
    is larger than the rounded $\mathsf{ND_T}$ of test;
Let is_memop(op) return true if op is a memory operation, false
    otherwise; where true, op has a valid attribute addr denoting the
    memory address accessed;
Let random_bool(p) generate a Bernoulli variate with probability p;

Function fitaddr_fraction(test) begin /* Returns
fraction of memory operations which are guaranteed
to be selected.                                        */
    $\mathsf{mem\_ops} \leftarrow [\mathsf{op} | \langle \mathsf{pid}, \mathsf{op} \rangle \in \mathsf{test} \wedge \mathtt{is\_memop(op)}]$;
    return $\frac{\mathtt{len}([\mathsf{op} | \mathsf{op} \in \mathsf{mem\_ops} \wedge \mathsf{op.addr} \in \mathtt{fitaddrs}(\mathsf{test})])}{\mathtt{len}(\mathsf{mem\_ops})}$;

Function crossover_mutate(test1, test2) begin
    $a_1 \leftarrow$ fitaddr_fraction(test1);
    $a_2 \leftarrow$ fitaddr_fraction(test2);
    $P_{SELECT1} \leftarrow a_1 + P_{USEL} - (a_1 \cdot P_{USEL})$;
    $P_{SELECT2} \leftarrow a_2 + P_{USEL} - (a_2 \cdot P_{USEL})$;
    child $\leftarrow$ test1;
    mutations $\leftarrow 0$;
    for i $\leftarrow 0$ to len(child) do
        $\langle \mathsf{pid}, \mathsf{op} \rangle \leftarrow$ test1[i];
        if is_memop(op) then
            select1 $\leftarrow$ random_bool($P_{USEL}$)
                    $\vee$ op.addr $\in$ fitaddrs(test1);
        else
            select1 $\leftarrow$ random_bool($P_{SELECT1}$);
        $\langle \mathsf{pid}, \mathsf{op} \rangle \leftarrow$ test2[i];
        if is_memop(op) then
            select2 $\leftarrow$ random_bool($P_{USEL}$)
                    $\vee$ op.addr $\in$ fitaddrs(test2);
        else
            select2 $\leftarrow$ random_bool($P_{SELECT2}$);
        if $\neg$select1 $\wedge$ select2 then
            child[i] $\leftarrow$ test2[i];
        else if $\neg$select1 $\wedge$ $\neg$select2 then
            mutations $\leftarrow$ mutations + 1;
            if random_bool($P_{BFA}$) then
                child[i] $\leftarrow$ Make random $\langle \mathsf{pid}, \mathsf{op} \rangle$, with addresses
                constrained to
                fitaddrs(test1) $\cup$ fitaddrs(test2);
            else
                child[i] $\leftarrow$ Make random $\langle \mathsf{pid}, \mathsf{op} \rangle$;
        else
            /* Retain node child[i].                           */
    if mutations/len(child) $< P_{MUT}$ then
        Mutate child with probability $P_{MUT}$;
    return child
```

Table 1: Guest-host interface.

Function	*Description*
barrier_wait_coarse()	*Host-assistance optional.* Barrier which does not mandate threads to be precisely synchronized.
barrier_wait_precise()	*Host-assistance suggested.* Barrier which mandates that threads are precisely synchronized via host-assistance or otherwise, such that upon return threads are in lock-step.
make_test_thread(*code*)	*Direct host-interface.* Host writes *code* for current test of thread.
mark_test_mem_range(*a*, *b*)	*Direct host-interface.* At guest workload initialization, use to set test generator address-range from start address *a* to end *b*.
reset_test_mem()	*Host-assistance suggested.* Resets (write initial values) locations used by test; flushes cache lines and other structures affecting following test executions.
verify_reset_all()	*Direct host-interface.* Verifies last test execution. Clear entire candidate execution object (static and conflict orders). *Evaluates test-run and sets up next test.*
verify_reset_conflict()	*Direct host-interface.* Verifies last test execution. Clears only conflict orders of candidate execution object.

mutation. Figure 2 illustrates test representation and their crossover.

4. ACCELERATING TEST EXECUTION & CHECKING

To allow a GP approach to progress towards more optimized tests as fast as possible, we must increase test throughput, i.e. minimize the wall-clock time for each test-run. As the tests are run in a full-system, a minimal guest workload is responsible for setup and running each test. We minimize the wall-clock time to execute code that is part of test setup and control, but does not contribute towards actual test execution. We propose several extensions, that any simulator to be used for verification should implement.

Table 1 shows the proposed interface between the simulation-aware guest workload and the host system[4]. Algorithm 2 shows the kernel of the guest workload, and is self-explanatory. While it is possible to implement many of the functions as part of the guest program (*optional* and *suggested* in Table 1), host-assistance transfers the implementation onto the simulation host system, thereby *speeding them up significantly*. In particular, we found that the host assisted barrier is a mandatory pre-requisite to execute very short tests, as the perturbation and thread offset induced by a guest barrier implementation was too large. With the host assisted barrier, thread start offset is minimized, and using very short tests becomes possible.

4.1 Checker

A pre-silicon environment, in this case simulation, provides certain advantages over post-silicon; most notably, we can afford to observe all necessary conflict orders to implement a polynomial-time decision procedure to verify if a recorded candidate execution object is valid or invalid with respect to the target MCM [33]. At the core of the checker is a regular depth-first search (DFS).

[4]Here we refer to *guest* as the system being simulated, and *host* the simulation software.

Algorithm 2: Guest workload: per thread kernel. The *control thread* is a thread selected at program startup to drive the generate-verify-reset cycle.

```
Input: test_iterations denoting the execution count of a test per
       test-run.
/* Every thread has its own independent memory
   region, which is used by the host to copy the
   respective code for the thread.                    */
code ← Allocate executable memory, host-writable;
while true do
    barrier_wait_coarse();
    make_test_thread(code);
    for i ← 0 to test_iterations do
        barrier_wait_precise();
        execute code;
        barrier_wait_coarse();
        if i + 1 < test_iterations ∧ is control thread then
            verify_reset_conflict();
            reset_test_mem();
    if is control thread then
        verify_reset_all();
        reset_test_mem();
```

Our implementation bases the formalization of MCMs on the framework proposed by Alglave et al. [4]. The precise and correct formalization of more complex MCMs (e.g. ARM or Power, or proposed GPU [5] models) should not be attempted in an ad-hoc manner, and by implementing the aforementioned framework we can afford a more direct mapping of these published MCMs to our checker. Note, however, that the style (axiomatic vs. operational) nor the particular formalization is a dependency for our proposed test generation scheme.

All static orders required to compute the preserved program order (ppo) are gathered before first execution of a test. The DAG representation (§3.3) of a test makes this straightforward. Furthermore, before test execution, each write event is assigned a unique ID – the value to be written by the associated instruction – to be able to map observed values to a producing write. This implies that the size of each instruction can support the maximum desired writes; there is no limit for read count. Initially, all memory is zero, and upon reading the initial value, the initial write event is created on first use.

All dynamic orders (conflict orders rf and co) are observed during execution of a test (without affecting functional execution). Constructing rf requires extracting the value an instruction reads; inserting into co requires extracting the value an instruction overwrites. In order to map *committed* instructions to an operation of a test, which then maps to an event in the MCM, we use the respective instruction pointers (IPs) to create a unique mapping. In case where an instruction can give rise to several reads and/or writes, we use the microcode counter to uniquely map to an event.

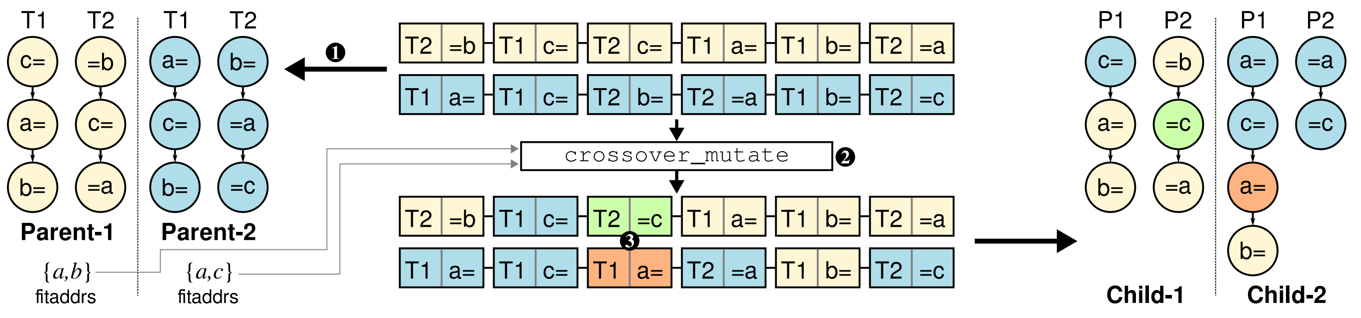

Figure 2: Crossover and mutation example. Initially two tests with two threads each, ❶ which are then evaluated and the set of fitaddrs determined to be $\{a,b\}$ for Parent-1 and $\{a,c\}$ for Parent-2. ❷ Given these two parents, crossover can produce several children, of which two are shown. ❸ Unselected addresses in the same slot for both parents result in mutation in this slot, and further mutation is no longer necessary.

5. EVALUATION METHODOLOGY

This section discusses the evaluation methodology used in obtaining the results (§6). The goal of the evaluation is to show the performance of the McVerSi framework with regard to its bug finding capability and the wall-clock time required to find a bug.

5.1 Simulation Environment

We evaluate our approach using the cycle accurate Gem5 simulator [26] with Ruby and GARNET [41] in *full-system* mode with the x86-64 ISA. It is worth noting that Gem5 is frequently used for pre-silicon design evaluation, with several industrial users. The processor model used for each core is a simple out-of-order processor. Table 2 shows the key-parameters of the system. All cache coherence protocol implementations are modeled in a functionally accurate manner (not just timing), to ensure that stale data (e.g. due to a protocol bug) affects functional execution.

To demonstrate that the bug finding ability of a particular test generator is consistent (statistically significant), we run each generator/bug pair 10 times with a time limit – this should provide high confidence in a test generator in case all runs find a bug found. To demonstrate that the convergence time of the GP-based approach is within practical bounds, each simulation run is limited to 24 hours of host time[5]. For non-GP test generators, we shall note that this effectively translates to measuring the frequency of a bug found within 24×10 hours (10 days), as these test generators do not continuously update their internal state to progress towards better tests.

Each simulation run (out of 10) uses a different random seed for both simulation and test generation yielding different executions per sample. Furthermore, simulation startup overheads are negligible, as the simulation loads the guest workload (§4), and then runs it continuously until a bug is found or the time limit is reached. Upon reset after a test execution (one iteration of a test-run), non-test related simulation state is not reset; therefore, the following executions of the same test in the same simulation are all perturbed differently.

5.2 Test Generation & Checking

[5]The host platform is server-grade, with Intel Xeon® E5620 CPUs; we measure 30k simulated instructions per second.

Table 2: System parameters.

Core-count & frequency	8 (out-of-order) @ 2GHz
LSQ entries ROB entries	32 40
L1 I+D -cache (private) L1 hit latency	32KB+32KB, 64B lines, 4-way 3 cycles
L2 cache (NUCA, shared) L2 hit latency	128KB×8 tiles, 64B lines, 4-way 30 to 80 cycles
Memory Memory hit latency	512MB 120 to 230 cycles
On-chip network	2D Mesh, 2 rows, 16B flits
Kernel	Linux 2.6.32.61

This section outlines the test generation and checking approaches we evaluate and compare.

5.2.1 McVerSi

To demonstrate the effectiveness of our proposed test generation approach, we compare against the following test generation variants. It is worth noting that each of the following still makes use of the simulation-specific optimizations (§4) of the McVerSi framework.

First, to show the effectiveness of pseudo-randomly generated tests, which most previous works (§7) rely on, we include the **McVerSi-RAND** configuration.

Next, we evaluate a naïve GP-based approach, **McVerSi-Std.XO**, which demonstrates the need for our domain-specific crossover. McVerSi-Std.XO does not make use of the selective crossover and instead, for all threads, connects subgraphs, by removing a random vertex, of a thread from two parents; a standard single-point crossover over the flat list can be exploited to efficiently realize this. The fitness function is modified to include the additional objective for improving test suitability (equal weighting for coverage and normalized $\mathrm{ND_T}$).

Finally, the configuration **McVerSi-ALL** includes all proposed test generation (§3) and the simulation-specific optimizations (§4). Both McVerSi-ALL and McVerSi-Std.XO implement a *steady-state GA* with *tournament-selection* and the *delete oldest replacement* strategy [42]. It is worth noting that steady-state GAs have been shown to outperform generational GAs in dynamic or non-stationary environments [42].

To assess the effect of our proposed crossover function alone – i.e. to answer the question: *are highly non-deterministic*

Table 3: Test generation parameters.

Test size	1k operations (total across threads)
Iterations	10 test executions per test-run
Test memory (stride)	1KB (16B), 8KB (16B)
Operations:bias% (comment)	• Read:50% (read into reg.) • ReadAddrDp:5% (read into reg. with address dependency) • Write:42% (write from reg.) • ReadModifyWrite:1% (RMW, on x86 also implies fences) • CacheFlush:1% (cache flush, e.g. `clflush` on x86) • Delay:1% (constant delay using NOPs)
McVerSi-ALL, McVerSi-Std.XO	
Population size	100
Tournament size	2
Mutation probability (P_{MUT})	0.005
Crossover probability	1.0
McVerSi-ALL	
P_{USEL}	0.2
P_{BFA}	0.05

tests alone sufficient? – we did evaluate a configuration with a constant fitness function. While better performing than either McVerSi-RAND or McVerSi-Std.XO, we still found its performance to be notably inferior to McVerSi-ALL, and so do not include it in the final results in §6. In a similar vein, we do not include configurations without our simulator-specific optimizations in our results, as without these, the simulation runs were impracticably slow (around a couple of orders of magnitude slower).

Test Generation Parameters: Key parameters for all configurations are shown in Table 3. Note that these parameters were determined to give good results in a limited design space exploration. The *test size* of 1k operations is sufficient to find all studied bugs; in fact, we determine that larger test sizes cause performance to degrade, as the evolution of tests simply takes longer. We must ensure that tests are large enough to be able to detect most bugs in the first place, but not too large to limit the search performance. With this test size and depending on several factors (conflicting accesses, L1/L2 cache hits/misses, etc.), we note that the checker (§4.1) generally uses between 30% and 40% of the total wall-clock time.

The *test memory size* denotes the *usable* address range. The *stride* merely affects the base address (base addresses are generated in multiples of stride). In order to ensure cache capacity evictions take place, the test memory is *partitioned* in contiguous blocks of 512B, where the respective starting addresses of partitions are separated by a range of 1MB; e.g. in the case of 8KB, 16 such 512B partitions exist. As we are running full-system simulations, the allocation is not fully under our control, and the virtual memory manager (VMM) of the OS has final control over placement in physical memory. In our experiments, however, we observe our chosen test memory partitioning to have the desired effect.

The selected *operations and their bias*, while independent of a particular ISA, should be guided by the target MCM. In our case, to cover all enforced orderings of x86-TSO, the presented operations are sufficient. For more relaxed MCMs, the set of operations that need to be generated could be more extensive.

5.2.2 diy-*litmus*

The diy tool suite [17] automates litmus test generation, using knowledge of the MCM to generate a number of short tests which may trigger interesting behaviour. Litmus tests are self-checking, i.e. they include the code for performing checking.

We generate all litmus tests for x86-TSO – we use all 38 tests available. We modified the run-script to exit the simulation on a detected MCM violation. As the simulation is time-limited (24 hours), and realistically it is not possible to pre-determine which of the litmus test will detect an error, we choose conservative parameters to limit the runtime of an individual litmus test, but re-execute all tests (in an outer loop) after the last of the tests has been executed. Thus the litmus tests may run until the simulation is terminated by the time-limit. For simulation we choose the following parameters: -st 4 (stride), -r 3 (runs), -s 8000 (size of test, iterations).

5.3 Selected Bugs

The following outlines the 11 studied bugs, 2 of which have not been discovered in Gem5 prior to this work. All bugs marked with a "*" denote real bugs in Gem5; others refer to artificially injected bugs. The prefix of the name we give a bug denotes which protocol is affected, as well as if either Load Queue (LQ) or Store Queue (SQ) contribute to the bug manifestation. We study two cache coherence protocols, one being the Ruby MESI implementation in Gem5, and the other the recently proposed TSO-CC [3] protocol. TSO-CC provides an interesting case-study, as it implements a lazy consistency-directed coherence protocol for TSO. TSO-CC explicitly violates SWMR, a key invariant of traditional coherence protocols such as MESI, which makes it arguably more difficult to verify adherence to memory consistency using formal verification approaches such as model checking.

MESI,LQ+IS,Inv*: This bug causes read→read reordering (same or different addresses) that is prohibited by TSO. It is caused by the coherence protocol failing to forward an invalidation to the LQ after sinking an incoming Inv (invalidate) request in the IS (invalid-to-shared) transient state. The correct behaviour would be to forward the invalidate along with the data once the data response message is received in the IS_I (invalid-to-shared, sunk invalidate) transient state. This is a real bug in Gem5, that had not been discovered previously. It is worth noting that this bug could not have been found via individual verification of either the coherence protocol (SWMR is not violated) or the LQ. The fix required correcting both components, and the Gem5 developers have been notified.

Note that this bug, as well as all following bugs with prefix MESI,LQ are variants of the "Peekaboo" problem [15] – in these cases, arising due to speculative execution. Indeed, Gem5's implementation of the LQ provides correct behaviour on a forwarded invalidation: if there exist any unperformed older reads and an invalidation is received, all newer reads are retried. However, if the coherence protocol never forwards an invalidation as is the case here, then newer reads may observe stale values.

MESI,LQ+SM,Inv*: This bug also causes read→read reordering (same or different addresses). It is caused by the coherence protocol failing to forward an invalidation to the LSQ in the SM (shared-to-modified) transient state upon receiving an Inv request. This bug has not been discovered previously. The fix only required correcting the coherence protocol, and a patch has been sent upstream to Gem5.

MESI,LQ+E,Inv: This bug results in read→read reordering (same or different addresses). It is caused by the coherence protocol failing to forward an invalidation to the LQ in the E state upon receiving an Inv.

MESI,LQ+M,Inv: Similar to MESI,LQ+E,Inv, but fails to forward an invalidation to the LQ in the M state.

MESI,LQ+S,Replacement: This bug is caused by the coherence protocol failing to forward an invalidation to the LQ upon replacement in the S state. It results in read→read reordering (same or different addresses).

MESI+PUTX-Race*: This bug is caused by a protocol race condition and subsequent invalid transition. It is described in detail by Komuravelli et al. [12], who previously discovered it via model checking with Murφ. This bug does not manifest as a MCM bug directly, but instead is caught by Ruby as an invalid transition. If such a protocol had passed to a post-silicon stage, the effect the bug can have is not very clear: the result may be unexpected behaviour (including an MCM bug) or something arguably more critical (e.g. system lockup). This bug has since been fixed in Gem5 (in January 2011).

MESI+Replace-Race: This bug is another protocol race; however, it is more subtle in nature. It manifests due to a L1 replacement in M and simultaneous L2 replacement of a previously clean block in MT (potentially modified, in local L1), where the L2 does not expect modified data, thereby failing to write back the modified block to memory.

TSO-CC+no-epoch-ids: To reset timestamps, TSO-CC requires epoch-ids to avoid races between timestamp-reset messages and read/write requests. Eliminating epoch-ids causes TSO violations (read→read reordering).

TSO-CC+compare: This bug is subtler than the previous one. In the presence of timestamp-groups, [3] states "where the requested line's timestamp is larger or equal than the last-seen timestamp from the writer of that line self-invalidate all Shared lines" – we change the comparison to just *larger than*. This bug causes read→read reordering.

LQ+no-TSO*: This bug causes read→read reordering to different addresses. The bug is caused by the LQ not squashing subsequent reads after an incoming forwarded invalidation from the coherence protocol. We previously discovered this bug via litmus testing, and sent a fix upstream in March 2014. This bug has also been independently discovered by PipeCheck [16].

SQ+no-FIFO: This bug causes write→write reordering by not writing back in FIFO order, but instead out-of-order from the SQ.

6. EXPERIMENTAL RESULTS

This section discusses the results we obtain for each individual test generation approach. First and foremost, we are interested in bug coverage, which addresses the bug-finding guarantees that each approach provides. This is followed by analysis of structural coverage, which addresses how thoroughly each approach explores the coherence protocol state transitions.

6.1 Bug Coverage

As seen in Table 4, the only configuration consistently finding all bugs in under 24 hours is our GP-based approach McVerSi-ALL (8KB). In comparison, McVerSi-RAND (best case with 1KB) only finds 8/11 of bugs and litmus tests only 2/11 bugs consistently within 24 hours. Furthermore, we can see that even when the competing approaches successfully find all bugs consistently within 24 hours, our GP-based approach almost always finds them sooner. This confirms our hypothesis that, although litmus testing and pseudo-random testing are effective post-silicon verification methodologies, without substantial optimizations, they are unsuitable for practical simulation-based verification.

What guarantees are provided with increasing runtime? Other than our McVerSi-ALL (8KB), no other configuration is able to find all bugs within 1 day. But what happens when the competing non-GP approaches (pseudo-random and litmus tests) are run for more than 1 day? Recall that we run each generator/bug pair 10 times (samples) up to 24 hours[6]. Since the non-GP approaches are stateless (they do not continuously update their internal state to progress towards better tests), we note that running each bug 10 times for 24 hours is tantamount to measuring bug coverage when running for up to 10 days. Table 5 summarizes the results under this assumption. It is worth noting that neither litmus nor pseudo-random tests are able to find all bugs within effectively 10 days of running time. Although McVerSi-RAND (8KB) can guarantee finding additional bugs after running for more than 1 day, 2 out of 11 bugs (18%) are still not found. In these cases (NF in Table 4), the implication is that the test generator would either need more than 10 days, or is incapable of generating tests required to expose the particular bug.

How does usable address range affect test quality? With just 1KB of test memory, all test generation schemes achieve similar results. Because of the constrained address space, tests consist of a large number of conflicting accesses even if generated randomly. However, note that both GP approaches improve the average time to find all bugs over the pseudo-random test generator even with just 1KB of test memory; in particular, McVerSi-ALL reduces the average time by 27% in comparison with McVerSi-RAND. It is important to note, however, that none of the approaches using 1KB of test memory are able to find the following bugs: MESI,LQ+S,Replacement, MESI+PUTX-Race, and MESI+Replace-Race. Clearly, we require a larger test memory size to find these. With 8KB of test memory, McVerSi-ALL is able to find all bugs in all simulation runs, including the 3 bugs above.

How effective is our selective crossover? We note that McVerSi-Std.XO is unable to find certain bugs (NF), in cases where the bugs only manifest due to racy accesses. Those configurations not making use of the proposed selective crossover simply do not converge towards suitable MCM tests with high non-determinism/races; i.e. their set of candidate executions

[6]For practical reasons, we are restricted to 24 hours per run.

Table 4: Bug coverage: *bug found count out of 10 samples (arith. mean hours to find the bug across 10 samples)*; NF = "Not Found within 24 hours"; **bold** highlights configurations which consistently find the bug within 24 hours.

Bug	*McVerSi-ALL (1KB)*	*McVerSi-ALL (8KB)*	*McVerSi-Std.XO (1KB)*	*McVerSi-Std.XO (8KB)*	*McVerSi-RAND (1KB)*	*McVerSi-RAND (8KB)*	*diy-litmus*
MESI,LQ+IS,Inv	**10 (0.01)**	**10 (0.49)**	**10 (0.01)**	**10 (0.73)**	**10 (0.01)**	**10 (0.89)**	NF
MESI,LQ+SM,Inv	**10 (0.33)**	**10 (5.20)**	**10 (0.27)**	1 (5.01)	**10 (0.48)**	NF	NF
MESI,LQ+E,Inv	**10 (2.97)**	**10 (0.09)**	**10 (3.22)**	**10 (0.16)**	**10 (4.34)**	**10 (0.10)**	NF
MESI,LQ+M,Inv	**10 (1.42)**	**10 (1.37)**	**10 (2.40)**	7 (3.80)	**10 (1.93)**	**10 (11.05)**	NF
MESI,LQ+S,Replacement	NF	**10 (2.69)**	NF	4 (15.05)	NF	6 (10.10)	NF
MESI+PUTX-Race	NF	**10 (4.64)**	NF	5 (8.83)	NF	3 (9.63)	NF
MESI+Replace-Race	NF	**10 (0.12)**	NF	**10 (0.12)**	NF	**10 (0.19)**	5 (0.53)
TSO-CC+no-epoch-ids	**10 (0.90)**	**10 (7.40)**	**10 (0.50)**	NF	**10 (0.96)**	NF	6 (5.93)
TSO-CC+compare	**10 (0.01)**	**10 (2.28)**	**10 (0.01)**	NF	**10 (0.01)**	1 (22.31)	**10 (0.92)**
LQ+no-TSO	**10 (0.00)**	**10 (0.03)**	**10 (0.00)**	**10 (0.02)**	**10 (0.00)**	**10 (0.08)**	**10 (5.35)**
SQ+no-FIFO	**10 (0.01)**	**10 (0.24)**	**10 (0.01)**	**10 (0.83)**	**10 (0.01)**	**10 (0.40)**	9 (4.77)
All	80 (0.71)	**110 (2.23)**	80 (0.80)	67 (2.31)	80 (0.97)	70 (3.41)	40 (3.60)

Table 5: Bugs found, when running up to the equivalent of 10 days time.

Bugs found within	1 day	5 days	10 days
McVerSi-ALL (8KB)	**100%**	N/A	N/A
McVerSi-RAND (1KB)	73%	73%	73%
McVerSi-RAND (8KB)	55%	73%	82%
diy-litmus	18%	45%	45%

is too small to have a high probability of encountering the sequence of events in the system required to expose faulty logic. In order to find bugs which only manifest due to replacements, a large address range is required but also *suitable* MCM tests, i.e. highly racy tests. The bugs which are *only* found by McVerSi-ALL (8KB) require tests with an average $\mathsf{ND_T}$ of at least 2.0, and often greater than 3.0. The 1KB configurations' initial set of tests automatically achieve an average $\mathsf{ND_T}$ exceeding 2.0, whereas the 8KB configurations start out with an $\mathsf{ND_T}$ of around 1.1. At 8KB, only McVerSi-ALL is able to generate tests with an $\mathsf{ND_T}$ of 2.0 or above.

6.2 Structural Coverage

What is the impact of using coverage as fitness? Table 6 shows the maximum total achieved coverage (higher is better). Recall that, the fitness function we use does not make use of the total coverage, and instead focuses on rare transitions to avoid getting stuck in a local maximum. We note that the implementations of MESI and TSO-CC contain transitions which are extremely unlikely to occur (e.g. replacements in transient states from invalid – the LRU replacement policy in use is very unlikely to select such blocks), which we did not exclude from the coverage calculation, and therefore we do not reach 100%.

From Table 6 we can see that McVerSi-ALL (8KB) achieves highest coverage for both MESI and TSO-CC. Using coverage as fitness achieves its goal, leading to the improved performance (bug coverage discussed above) of McVerSi-ALL compared to McVerSi-RAND. More importantly, while the selective crossover continually increasing $\mathsf{ND_T}$ could have a negatively correlated effect on coverage, the GP-based approach ensures balance by simply proceeding to no longer select individuals with too high $\mathsf{ND_T}$.

7. RELATED WORK

Formal verification: While formal verification provides the highest possible guarantees, i.e. a proof of correctness, the model being verified against its specification is typically a component abstraction of what is present in the functional design implementation; with the coherence protocol being the main artifact being subjected to formal verification [14]. For most model checking approaches [7, 9, 10, 12, 13], the consistency properties intended to capture MCM correctness are derived properties, such as the SWMR [15] invariant; these are inadequate for protocols explicitly violating such properties (e.g. lazy self-invalidation based protocols using "tear-off" blocks [43]). More powerful formal methods approaches for coherence protocol verification use operational models [8, 44, 45, 46], but require more user-effort to set up.

To raise confidence in a design, it would be prudent to *apply the best tools at each stage in the design.* Indeed, the recent model checking of the MESI coherence protocol of the GEMS memory simulator (and of Gem5) has found bugs [12] (MESI+PUTX-Race among others). Independent of the memory system, PipeCheck [16] can be used for model checking of pipeline abstractions (albeit against selected litmus tests), which also uncovered a bug in Gem5 (LQ+no-TSO). None of the above approaches can ensure the correctness of the interaction between components as we observed with several of the studied bugs (§5.3).

The recently published and concurrently developed CCICheck [47] (based on PipeCheck [16]), provides a methodology for verifying pipeline and memory system (with focus on coherence protocol) together. Broadly, their motivation is similar, in that the interaction between components is crucial in enforcing the consistency model, and unconventional protocols cannot easily be verified using traditional approaches (e.g. TSO-CC is also used as a case-study). By using abstract axiomatic models of pipeline (like PipeCheck) and memory system, the result is exhaustive (on input litmus tests).

While extremely valuable at an early stage in the design, the above approaches are only tractable with abstractions of the relevant parts of a full-system functional design, and thus are complementary to McVerSi.

Memory system verification: Verifying the detailed implementation of the memory system in isolation can be be done in simulation. Wood et al. [23] present a methodology where

Table 6: Maximum total *transition coverage* observed across all simulation runs.

Protocol	*McVerSi-ALL (1KB)*	*McVerSi-ALL (8KB)*	*McVerSi-Std.XO (1KB)*	*McVerSi-Std.XO (8KB)*	*McVerSi-RAND (1KB)*	*McVerSi-RAND (8KB)*	*diy-litmus*
MESI	60.9%	82.3%	62.3%	81.9%	60.9%	81.9%	66.5%
TSO-CC	51.8%	63.1%	50.8%	41.2%	51.8%	62.6%	54.8%

a stub CPU takes control of the operations being issued to the memory system; tests are randomly selected operations from user "action/check scripts". A similar approach is taken by [48]. None of the approaches automatically feed back coverage metrics into the test generation.

The "witness string" method [7, 49] generates test vectors for RTL simulation of the coherence protocol which cover distinct states, thereby improving test quality and reducing redundant simulation time. The witness strings are generated with the Murφ model checker, based on a model of the protocol. This approach, however, depends on an external tool and is not tightly coupled with the simulation tool.

In the presence of a detailed FSM of the coherence protocol, [50, 51] propose methods to automatically inject events into the memory system to cover previously uncovered states and transitions. This requires detailed knowledge of the memory system's FSM, and in the absence of other control logic (e.g. core pipeline), is a feasible approach to generating high coverage. Yet, it would be much more difficult to accomplish in a full-system, where the test input to the system does not directly control the memory system, and instead is subject to other constraints of the control logic.

While these approaches target simulation, unlike them, McVerSi targets *full-system* simulation, and also demonstrates checking a complete axiomatic MCM.

Full-system verification: Related work in this area has primarily focused on the problem of checker complexity due to limited visibility in a post-silicon environment. In the absence of conflict order visibility, checking an axiomatic MCM has been proven to be NP-complete [33].

To address the complexity of MCM checking in simulation, *relaxed scoreboards* (operational models) have been proposed [35, 34] to monitor every memory operation's correct behaviour. While this would even allow using real workloads and monitor the system on-the-fly, the test generation method is independent of the proposed checking method. We find that checking an axiomatic MCM is fast enough for the relatively short tests used in our GP-based approach.

TSOtool [20, 21] and derivative algorithms [22, 52] propose approximate solution to the MCM checking problem in a post-silicon environment, due to the limited conflict order visibility. Hardware extensions to facilitate fast checking in post-silicon have been proposed, e.g. via counters [18, 53] or even re-partitioning of the cache to log ordering information [19]. While throughput in post-silicon environments is generally higher, and therefore all use user constrained random tests, all of these approaches are only applicable at the very latest stages in a design.

Manually directed short tests, also called *litmus tests*, are also very common. More recently, diy [17] automates litmus test generation, using knowledge of the MCM to generate short tests which may trigger interesting behaviour. Litmus tests have the advantage that they are self-checking, and therefore are more portable and simpler to set up for a wide variety of MCMs.

The above approaches target post-silicon testing, and none are specifically optimized for simulation.

Hardware support for MCM verification: An alternative approach to ensure correctness, is fault-tolerance via hardware support for detecting MCM violations dynamically [54, 55, 56, 57], and recovering from them. Our proposal, which focusses on test generation, is orthogonal to these works. In particular, Romanescu et al. [57] focus on detecting address translation (AT) related bugs with the help of their AT-aware MCM specifications. Our framework targets a full-system environment, including the TLB and MMU, and thus is also capable of detecting AT bugs (we did not detect any). Note, however, that our framework currently does not stress AT related aspects (it does not generate synonyms, memory mapping operations, etc.), which we reserve for future work.

8. CONCLUSION

We have presented McVerSi, a test generation framework for fast memory consistency verification in full-system simulation. At later pre-silicon design stages, it is imperative to rigorously verify the full-system. Due to the complexity of the implementation at this stage, verification methodologies with rigorous test generation are of great importance to raise the designer's confidence.

In the domain of simulation-based MCM verification there is need for an approach which automatically improves test quality based on feedback from the simulation. This is a difficult search problem with many hidden variables, especially in multiprocessor systems, where the interleaving of threads is inherently non-deterministic. Indeed, the enforcement of a MCM is what brings order into the non-deterministic world of multiprocessors.

Our key contribution is a GP-based test generation approach, which generates effective MCM tests. By proposing a novel crossover which favours non-determinism, the generated tests increase the probability of the implementation having to work harder to enforce the required ordering guarantees of the MCM. Then, by using coverage as the fitness function, our approach evolves high-quality tests automatically. Our results show that, compared with alternative test generation approaches, we find all 11 considered bugs consistently, providing much higher guarantees about the classes of bugs McVerSi is capable of finding within practical time bounds. While it may be conceivable to achieve similar results via manual test generation, our approach automatically explores tests satisfying the coverage criteria without user intervention.

The framework we present offers the building blocks for researchers and industrial designers alike, to evaluate coherence protocols and other microarchitectural artifacts to adhere to the promised consistency model early in the design cycle.

We provide a simulator-independent C++ library (including consistency model descriptions, checker, and test generator): `https://github.com/melver/mc2lib`.

Acknowledgements

We would like to thank Jade Alglave, Stephan Diestelhorst, Paul Jackson, Susmit Sarkar and the anonymous reviewers for their helpful comments and advice. This work is supported by EPSRC grant EP/M027317/1 to the University of Edinburgh.

9. REFERENCES

[1] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[2] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[3] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *HPCA*, Feb. 2014.

[4] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory," *ACM Trans. Program. Lang. Syst.*, 2014.

[5] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency: Weak behaviours and programming assumptions," in *ASPLOS*, 2015.

[6] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. J. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, 2009.

[7] D. Abts, S. Scott, and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1," in *IPDPS*, p. 11, 2003.

[8] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, "Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking," in *CAV*, pp. 123–136, 2002.

[9] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou, "Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee," in *FMCAD*, pp. 81–88, 2006.

[10] C.-T. Chou, P. K. Mannava, and S. Park, "A Simple Method for Parameterized Verification of Cache Coherence Protocols," in *FMCAD*, pp. 382–398, 2004.

[11] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. R. Tuttle, and Y. Yu, "Checking Cache-Coherence Protocols with TLA$^+$," *Formal Methods in System Design*, vol. 22, no. 2, pp. 125–131, 2003.

[12] R. Komuravelli, S. V. Adve, and C.-T. Chou, "Revisiting the complexity of hardware cache coherence and some implications," *TACO*, 2014.

[13] K. L. McMillan, "Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking," in *CHARME*, pp. 179–195, 2001.

[14] F. Pong and M. Dubois, "Verification Techniques for Cache Coherence Protocols," *ACM Comput. Surv.*, vol. 29, no. 1, pp. 82–126, 1997.

[15] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2011.

[16] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models," in *MICRO*, 2014.

[17] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models (extended version)," *Formal Methods in System Design*, vol. 40, no. 2, pp. 170–205, 2012.

[18] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan, "Fast complete memory consistency verification," in *HPCA*, pp. 381–392, 2009.

[19] A. DeOrio, I. Wagner, and V. Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," in *HPCA*, pp. 405–416, 2009.

[20] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan, "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model," in *ISCA*, pp. 114–123, 2004.

[21] C. Manovit and S. Hangal, "Efficient algorithms for verifying memory consistency," in *SPAA*, pp. 245–252, 2005.

[22] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang, "Fast and Generalized Polynomial Time Memory Consistency Verification," in *CAV*, pp. 503–516, 2006.

[23] D. A. Wood, G. A. Gibson, and R. H. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design & Test of Computers*, vol. 7, no. 4, pp. 13–25, 1990.

[24] C. Ioannides and K. Eder, "Coverage-directed test generation automated by machine learning - A review," *ACM Trans. Design Autom. Electr. Syst.*, vol. 17, no. 1, p. 7, 2012.

[25] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[26] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[27] K. Gharachorloo, "Memory Consistency Models For Shared-Memory Multiprocessors," Tech. Rep. CSL-TR-95-685, 1995.

[28] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[29] SPARC International, Inc., *The SPARC Architecture Manual: Version 8*. 1992.

[30] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *ISCA*, pp. 15–26, 1990.

[31] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, "An Axiomatic Memory Model for POWER Multiprocessors," in *CAV*, pp. 495–512, 2012.

[32] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER multiprocessors," in *PLDI*, pp. 175–186, 2011.

[33] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, no. 4, pp. 1208–1244, 1997.

[34] A. Saha, N. Malik, B. O'Krafka, J. Lin, R. Raghavan, and U. Shamsi, "A simulation-based approach to architectural verification of multiprocessor systems," 1995.

[35] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz, "Verification of chip multiprocessor memory systems using a relaxed scoreboard," in *MICRO*, pp. 294–305, 2008.

[36] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.

[37] M. Srinivas and L. M. Patnaik, "Genetic Algorithms: A Survey," *IEEE Computer*, vol. 27, no. 6, pp. 17–26, 1994.

[38] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin, *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[39] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir, "A genetic approach to automatic bias generation for biased random instruction generation," in *CEC*, 2001.

[40] F. Corno, F. Cumani, and G. Squillero, "Exploiting Auto-adaptive μGP for Highly Effective Test Programs Generation," in *ICES*, pp. 262–273, 2003.

[41] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, pp. 33–42, 2009.

[42] F. Vavak and T. C. Fogarty, "Comparison of Steady State and Generational Genetic Algorithms for Use in Nonstationary

Environments," in *ICEC*, pp. 192–195, 1996.

[43] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *ISCA*, pp. 48–59, 1995.

[44] S. Park and D. L. Dill, "Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions," in *SPAA*, pp. 288–296, 1996.

[45] F. Pong and M. Dubois, "Formal Verification of Complex Coherence Protocols Using Symbolic State Models," *J. ACM*, vol. 45, no. 4, pp. 557–587, 1998.

[46] F. Pong and M. Dubois, "Formal Automatic Verification of Cache Coherence in Multiprocessors with Relaxed Memory Models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 9, pp. 989–1006, 2000.

[47] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface," in *MICRO*, 2015.

[48] F. Pong, M. C. Browne, G. Aybay, A. Nowatzyk, and M. Dubois, "Design Verification of the S3.mp Cache-Coherent Shared-Memory System," *IEEE Trans. Computers*, vol. 47, no. 1, pp. 135–140, 1998.

[49] Y. Chen, D. Abts, and D. J. Lilja, "State Pruning for Test Vector Generation for a Multiprocessor Cache Coherence Protocol," in *15th IEEE International Workshop on Rapid System Prototyping*, pp. 74–77, 2004.

[50] I. Wagner and V. Bertacco, "MCjammer: Adaptive Verification for Multi-core Designs," in *DATE*, pp. 670–675, 2008.

[51] X. Qin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," in *DATE*, pp. 3–8, 2012.

[52] A. McLaughlin, D. Merrill, M. Garland, and D. A. Bader, "Parallel Methods for Verifying the Consistency of Weakly-Ordered Architectures," in *PACT*, 2015.

[53] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li, "Linear Time Memory Consistency Verification," *IEEE Trans. Computers*, vol. 61, no. 4, pp. 502–516, 2012.

[54] A. Meixner and D. J. Sorin, "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures," *IEEE Trans. Dependable Sec. Comput.*, vol. 6, no. 1, pp. 18–31, 2009.

[55] A. Muzahid, S. Qi, and J. Torrellas, "Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically," in *MICRO*, pp. 363–375, 2012.

[56] X. Qian, J. Torrellas, B. Sahelices, and D. Qian, "Volition: scalable and precise sequential consistency violation detection," in *ASPLOS*, pp. 535–548, 2013.

[57] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin, "Specifying and dynamically verifying address translation-aware memory consistency," in *ASPLOS*, pp. 323–334, 2010.