MC Mutants: Evaluating and Improving Testing for Memory Consistency Specifications

Reese Levine
UC Santa Cruz
USA

Tianhao Guo*
New York University
USA

Mingun Cho*
UC Davis
USA

Alan Baker
Google
Canada

Raph Levien
Google
USA

David Neto
Google
Canada

Andrew Quinn
UC Santa Cruz
USA

Tyler Sorensen
UC Santa Cruz
USA

ABSTRACT

Shared memory platforms provide a memory consistency specification (MCS) so that developers can reason about the behaviors of their parallel programs. Unfortunately, ensuring that a platform conforms to its MCS is difficult, as is exemplified by numerous bugs in well-used platforms. While existing MCS testing approaches find bugs, their efficacy depends on the testing environment (e.g. if synthetic memory pressure is applied). MCS testing environments are difficult to evaluate since legitimate MCS violations are too rare to use as an efficacy metric. As a result, prior approaches have missed critical MCS bugs.

This work proposes a mutation testing approach for evaluating MCS testing environments: MC Mutants. This approach mutates MCS tests such that the mutants simulate bugs that might occur. A testing environment can then be evaluated using a mutation score. We utilize MC Mutants in two novel contributions: (1) a parallel testing environment, and (2) An MCS testing confidence strategy that is parameterized over a time budget and confidence threshold. We implement our contributions in WebGPU, a new web-based GPU programming specification, and evaluate our techniques across four GPUs. We improve testing speed by three orders of magnitude over prior work, empowering us to create a conformance test suite that reproduces many mutated tests with high confidence and requires only 64 seconds per test. We identified two bugs in WebGPU implementations, one of which led to a specification change. Moreover, the official WebGPU conformance test suite has adopted our approach due to its efficiency, effectiveness, and broad applicability.

*Work done while at UC Santa Cruz.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575750

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; • **Computing methodologies → Parallel programming languages**.

KEYWORDS

memory consistency, parallel programming models, mutation testing

ACM Reference Format:
Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2023. MC Mutants: Evaluating and Improving Testing for Memory Consistency Specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3575693.3575750

1 INTRODUCTION

A memory consistency specification (MCS) for a shared memory runtime platform defines the legal values that loads are allowed to observe, and thus, is crucial for program reasoning. For efficiency reasons [16], many platforms provide a *relaxed* MCS, which allows complex behaviors. Validating that a platform implementation honors its MCS is a difficult, yet important, since a bug may lead to rare, non-deterministic application errors [3, 35].

Early MCS testing work executed sequences of memory accesses [17, 39] and then analyzed a trace of the observed values. More recent work uses formal specifications to derive small concurrent tests, often called *litmus tests*, which are then executed for many iterations, checking for a violation at each iteration [5]. Litmus testing strategies have found numerous bugs in runtime platforms [1, 4, 24, 29, 30].

1.1 Motivating Examples

The effectiveness of litmus testing approaches is *extremely* sensitive to the testing environment, i.e. the context around the litmus test that creates stress on the platform. We use the two litmus tests of Fig. 1 to illustrate this. The CoRR (Coherence of Read-Read) test, shown in Fig. 1a, tests if the first read in a thread (operation ⓐ)

thread 0	thread 1
❶ v0 = atomicLoad(x)	ⓒ atomicStore(x,1)
❷ v1 = atomicLoad(x)	

Condition: v0 == 1 && v1 == 0

(a) Coherence of Read-Read (CoRR) litmus test

thread 0	thread 1
❶ atomicStore(x, 1)	ⓓ v0 = atomicLoad(y)
❷ fence(release)	ⓔ fence(acquire)
❸ atomicStore(y, 1)	ⓕ v1 = atomicLoad(x)

Condition: v0 == 1 && v1 == 0

(b) Message Passing rel/acq (MP-relacq) litmus test

Figure 1: Two litmus tests that illustrate bugs in WebGPU found by our work. We observed executions that resulted in the condition being true, even though it is disallowed by the MCS. All memory is initialized to 0.

can observe an updated value (from operation ⓒ), while a following read (operation ❷) reads a stale value (from the initial state). A straightforward testing environment (i.e. one without stress) observes no violations, but when executed in a stressful testing environment (e.g. as used in [1, 24]), the test evinces a bug in the WebGPU platform when executing on an Apple device with an Intel GPU.[1]

The MP-relacq (Message Passing relacq) litmus test, shown in Fig. 1b, writes to location x (❶) and sets a flag (❸) in thread 0, then reads the flag (ⓓ) and reads the data (ⓕ) in thread 1. The fence instructions synchronize across the threads. A violation occurs if thread 1 observes the updated flag without observing the updated data. While existing stress testing environments did not reveal any violations, our novel parallel approach was able to observe violations in the WebGPU platform on AMD GPUs. This resulted in a fix to an AMD Vulkan compiler and a specification change to the WebGPU MCS [9].

When testing, it is impossible to tell if an unobserved illegal execution is not allowed or if it is simply rare and was not exposed by the tests. Thus, effective MCS testing requires tuning testing environments to reduce the possibility of a missed bug. However, legitimate MCS violations are exceeding rare, and thus cannot be used as a metric for such tuning efforts. Existing approaches have tuned their testing environments by maximizing the number of weak, but allowed, behaviors of some litmus tests, yet have not provided justification on why this metric is valid. Finally, existing MCS testing techniques do not provide confidence in their ability to find MCS bugs. Consequently, it is difficult to gauge whether it is worth the time to perform more testing, especially when running in time-constrained environments such as a conformance test suite. The result of these limitations has been a lack of adoption of existing MCS testing approaches.

[1]The bug is in the WebGPU implementation, whereby Google Chrome layers over Apple's platform-level Metal API. We have filed an issue with Apple and await a resolution.

1.2 Our Approach: MC Mutants

In this work, we propose *Memory Consistency Mutants* or MC Mutants, a methodology that quantitatively evaluates MCS testing environments using black-box mutation testing [13, 15]. Litmus tests are *mutated* in such a way that the erroneous behavior that the test was checking is are now allowed. A test environment *kills* the mutated test (mutant) by observing the newly allowed execution. The number of mutants that are killed is called the *mutation score*. Additionally, MC Mutants can be used to record the speed at which the mutants are killed, called the *mutant death rate*.

Intuitively, mutants check for an allowed weak memory or fine-grained interleaving behavior that is closely related to a disallowed behavior of an MCS litmus test. Our main contribution arises in formally defining this notion of *related*; MC Mutants generates mutants systematically by disrupting edges in a happens-before cycle of a disallowed behavior, corresponding to a small change in the program syntax of the litmus test.

The effectiveness of a testing environment can be evaluated by its mutation score (and mutant death rate) when executing mutants. Thus, a test environment must aim to maximize legal behavior in the mutants that closely relates to the illegal behavior in the original litmus tests. Hence, there are two constraints upon which MC Mutants depends.

First, the legal mutant behavior must be *observable* on the testing platform. That is, even if the mutant behavior is allowed by the specification, it may not be observable on the testing platform. If the mutant behavior is not observable, then MC Mutants will be unable to evaluate the testing environment with respect to the given mutant and corresponding litmus test. Our results (see Sec. 5.2) show that most mutant behavior is observable on GPUs across a range of mainstream vendors. However, this might not hold for all platforms, or mutants, in which case the mutants should be pruned according to the expected observable behavior on the implementation (discussed more in Sec. 3.4).

Second, the validity of MC Mutants depends on a correlation between observing legal mutant behavior and real MCS bugs. We validate this correlation by showing that a testing environment's ability to kill a mutant is highly correlated with its ability to find related MCS bugs in three cases (Sec. 5.4). This also means that testing environments that do not reliably kill mutants are not able to expose the MCS bugs.

MC Mutants enables principled evaluation and optimization of MCS test environments. An optimized testing environment can increase the confidence that MCS bugs will be found, if they exist, using the original, un-mutated test suite. We utilize MC Mutants to develop two novel testing environment improvements: a parallel testing environment and a testing confidence strategy. Using a combination of these contributions, we show that both bugs we identified in Fig. 1 are rapidly and reliably revealed in testing environments optimized using MC Mutants. We now detail our testing environment contributions.

Parallel testing environment (PTE). While previous work ran litmus tests in parallel on multicore CPUs [5], the effectiveness was limited by the low number of cores found on traditional SMP CPU machines. As the number of parallel tests increases, it becomes difficult to scale thread-to-test assignment in a way that is safe,

efficient, and effective; especially in the presence of tunable stress parameters. Given this, prior techniques for running parallel litmus tests do not generalize to GPUs.

We overcome these challenges with a novel parallel testing technique that uses a parallel permutation strategy based on modular arithmetic of co-prime numbers to assign test instances to threads with low overhead. This approach, which we call *Parallel Testing Environments*, or PTE, allows us to increase mutant death rates by on average three orders of magnitude over previous work. Moreover, MC Mutants shows that parallel testing synergizes with the stress testing environment given in prior work [24], which increases the mutant death rate by an additional 43%. The efficiency gained using PTE is critical for observing the MP-relacq bug (Fig. 1b). We were unable to observe it through extensive stress tuning using techniques from prior work. However, under PTE, it readily appears at a rate of 10.4 violations per second.

MCS Test Confidence. It is impossible to have complete confidence that testing will uncover bugs, as litmus tests are nondeterministic. However, we can utilize MC Mutants to provide a *reproducibility score* for a testing environment. This is a statistical confidence that a testing run will kill a mutant given the mutant death rate on previous runs. It is parameterized by a time budget, i.e. how long each test can be run for. Our results, across four GPUs spanning four mainstream vendors, that on our tuned parallel test environment can provide a mutation score of 82% and a reproducibility score of 99.999% with a budget of 64 seconds per test, matching the maximum mutation score achieved by prior work with $(1/4096)^{\text{th}}$ the time budget. Reproducibility scores provide a principled way to evaluate the relative effectiveness of test environments at finding bugs.

Evaluation specification. We evaluate our approach in WebGPU [41], a new web-based *general purpose* GPU (GPGPU) framework. WebGPU has a hierarchical execution model similar to other GPU platforms, e.g. CUDA and OpenCL. Threads are partitioned into workgroups, and there are several different memory regions.

However, the WebGPU MCS differs from other specifications in that it targets devices from a wide range of vendors, e.g., Intel, AMD, Apple, and NVIDIA.[2] In fact, it has the *largest* set of backends targeted by any GPGPU MCS. To provide a portable abstraction across these devices, WebGPU currently has relatively few options for synchronization; there are no sequentially consistent atomic operations as seen in NVIDIA's PTX [28], or OpenCL [10]. However, the few ordering properties that WebGPU provides, such as coherency across atomic accesses, must be preserved across all devices. Given this unique context, MCS testing is both *critical*, as it must ensure a single consistent MCS on a diverse set of backends, and *challenging*, as testing environments must be effective across many devices.

This work targets only one level (or *scope*) of the GPU execution hierarchy, specifically threads that are in different workgroups. We pick this scope for three reasons: (1) it is pragmatic, as there is no other way for inter-workgroup threads to communicate without the expensive routine of stopping and relaunching the kernel, (2) prior work developed stress techniques targeting this scope [24], providing testing environments that we can tune and evaluate, and (3) WebGPU does not (yet) expose the subgroup (or warp in CUDA) scope, and thus it cannot be tested. However, MC Mutants applies generally to MCS testing, and we aim to apply it to the more complete GPU execution hierarchy as the specification, stress testing techniques, and application use-cases continue to evolve.

Using MC Mutants, a set of 20 litmus tests and 32 mutant tests were created. We evaluate these tests across four GPU devices, spanning four vendors: Apple, AMD, Intel, and NVIDIA, for a total of 128 mutant/device combinations. One major impact of this work is its adoption into the WebGPU conformance test suite (CTS) for testing the MCS (see Sec. 5.5).

Contributions. In summary, our contributions are:

(1) MC Mutants: A mutation testing approach that can evaluate MCS testing environments (Sec. 3).
(2) A novel parallel test environment that executes litmus test instances in parallel (Sec. 4.1).
(3) An MCS test confidence strategy using a reproducibility score parameterized over a time budget (Sec. 4.2).
(4) An MC Mutants implementation and evaluation in WebGPU. Our evaluation spans 4 different GPUs from 4 different vendors and shows that: (Sec. 5)
   - MC Mutants generates a test suite consisting of 20 unmutated litmus tests and 32 mutated litmus tests.
   - parallel testing environments are able to kill 81% more mutants with an average mutant death rate over 2000× higher than prior work.
   - our MCS test confidence can explore trade-offs in reproducibility and time for test engineering.

We highlight two impacts of this work: first, it identified two MCS bugs in WebGPU implementations, one of which led to a specification change. Second, the work has been adopted into the WebGPU CTS (Sec. 5.5). The effectiveness of parallel test environments was a critical factor in getting the tests added to the CTS, as the full CTS must be run often during development. The strategies we developed for evaluating MCS test confidence allowed us to propose a time budget of approximately one minute on average desktop hardware for the full MCS test suite, adding little overhead to the full CTS.

2 BACKGROUND

We now provide background on memory consistency specifications (Sec. 2.1) and litmus tests (Sec. 2.2). We constrain the formalism to our case study specification, WebGPU (Sec. 2.3). As WebGPU is still a working draft, it does not yet have a formal MCS, though it is expected to be based on the Vulkan memory model [19]. Because of this, the MCS for WebGPU described here is not official. Instead, it should be viewed as a non-controversial foundation on which to explore new testing techniques.

2.1 Memory Consistency Specification (MCS)

The MCS defines two properties for shared memory concurrent programs: (1) the definition of a data-race, i.e. unsynchronized concurrent memory accesses that results in undefined behavior,

[2] With mobile support planned in the future, this will include even more vendors, e.g. Qualcomm, ARM and Imagination.

and (2) for well-defined programs, the values that memory loads may observe.

Data races. An MCS defines two types of memory locations, atomic and non-atomic. Atomic locations can be operated on by atomic operations, e.g. atomic load, atomic store, atomic read-modify-write (RMW). A data-race occurs when two threads access the same memory location without sufficient synchronization. The exact rules defining the required synchronization can be complex and have been studied extensively [11]; however, if all memory accesses occur on atomic memory, and use atomic operations, then the program is *trivially data-race free* (what we call TDRF). By their nature, data races invalidate a program and allow for undefined behavior. Therefore, we consider only the portion of the MCS that pertains to TDRF programs by focusing on programs that use atomic operations.

Sets and relations for MCS. Executions can be formally reasoned about under an MCS, e.g. to determine if an execution is allowed or not. Following work in [6, 11], an execution is viewed as a set of events and relations, summarized in Tab. 1. In our presentation, we do not consider non-atomic memory accesses or fences other than release/acquire fences. Additionally, atomic memory operations are not parameterized over a memory order (e.g. à la C++). This simplified model is sufficient for our WebGPU case-study.

An execution consists of events E, which can be reads (R), writes (W), read-modify-writes (RMW) or fences (F). The R, W, and RMW events contain memory locations, and the W and RMW events contain a value to store.

All reads (or RMWs) are related to one write (or RMW) through the reads-from (rf) relation. Events from the same thread are related through program-order (po) if they are sequenced in the program source (e.g. using ;). The *po-loc* relation is po, but only relates events that target the same memory location. All writes and RMWs to the same location are ordered through coherence (co), a superset of *po-loc*, which intuitively represents the order in which writes reach global visibility. The from-reads (fr) relation is derived from co and rf; it relates a read or RMW r to a write or RMW w if r reads-from a w' that is earlier in co than w. The communication (com) relation is simply the union of rf, co, and fr.

One fence (f_r) synchronizes-with (sw) another fence (f_a) if: (1) f_r and f_a are in different threads and (2) there is a write or RMW w in po after f_r, and there is a read or RMW r before f_a, and r reads from w.

Formal definitions of MCS. An MCS defines the happens-before (hb) relation across events in candidate executions. Once specified, executions can be determined if they are legal or not, given the following properties of hb: (1) A read must return the latest value from a write in hb order and (2) the hb order must be acyclic.

The strongest MCS is sequential consistency (or SC) [25], which states that hb must be a total order across all memory accesses and it must respect per-thread program order (po). In many CPU-centric programming languages (e.g. C++ and Java), SC is the default behavior for race-free programs. It can be defined by instantiating hb to be equal to $po \cup com$.

However, the SC MCS disallows many compiler and architecture optimizations. As a result, enforcing the SC MCS on a modern system requires using expensive fence instructions, either by the compiler or programmer. Most GPU languages are not SC by default, and many (including SPIR-V [21] and Metal [7]) do not provide the means for a programmer to achieve SC behavior, even if using fence instructions.

Table 1: Relations and sets used in MCS

Sets	**Description**
Reads (R)	An atomic operation that reads from an atomic memory location
Writes (W)	An atomic operation that writes to an atomic memory location
Read-modify-writes (RMW)	An atomic operation that both reads from and writes to an atomic memory location in one indivisible action
Fences (F)	An operation that performs a release/acquire fence
Relations	**Description**
program-order (po)	events from the same thread in instruction order
po-loc	po restricted to events that target the same memory location
reads-from (rf)	relates a W or RMW a to an R or RMW b, if b reads the value that a wrote
coherence (co)	a total order over writes or read-modify-writes to the same location
from-read (fr)	relates an R or RMW a to a W or RMW b if a obtained its value from a different W or RMW b' that comes before b in co
synchronizes-with (sw)	relates a release/acquire fence f_a, to another f_r if they are ordered through po, rf and po
communication (com)	The union of rf, co and fr

Despite this, there exists a baseline common to all languages we are aware of: SC-per-location [6], sometimes called coherence [11]. This specification states that for every atomic memory location a, there is an hb ordering across all memory accesses to a such that hb respects *po-loc*, i.e. program order per location. This MCS can be defined by setting hb to be $po\text{-}loc \cup com$.

We consider one more addition to hb related to our target platform of WebGPU: the release/acquire fence; and we will refer to this model as the rel-acq-SC-per-location MCS. To add this synchronization construct, the following relation is added to hb: $po; sw; po$ where the ; operator is sequenced-with. Intuitively, this states that if two fences f_r and f_a have synchronized, i.e. they are part of sw, then any event e such that $e \xrightarrow{po} f_r$ happens before any other event e' if $f_a \xrightarrow{po} e'$. This is follows the same definition as the sw relation in the C++ memory model [11].

2.2 Litmus Tests

A litmus test is a small concurrent program which tests if a platform implementation conforms to an MCS. Not only are they used in testing [6], but also as examples in MCS documentation [11], and can even be used to derive an MCS [12]. A litmus test has many candidate executions, given as a set of events and relations. Each time the test is run, one of the candidate executions occurs, which may or may not be valid under a given MCS.

Figure 2a shows a candidate execution of the CoRR litmus test, which is shown as an executable program in Fig. 1a. This execution is disallowed by the SC-per-location memory model, as it contains the following cycle in hb: $b \xrightarrow{fr} c \xrightarrow{rf} a \xrightarrow{po\text{-}loc} b$.

Figure 2b shows a candidate execution of the message-passing litmus test from Fig. 1b, in which release/acquire fences are used to synchronize. Thread 1 reads a stale value (0) from the initial state, even if the threads have synchronized through the x memory location (using rf and a fence). This execution is disallowed by the rel-acq-SC-per-location MCS as follows: $a \xrightarrow{hb} f$ from the release/acquire hb rule because $a \xrightarrow{po} b \xrightarrow{sw} e \xrightarrow{po} f$ is part of $po; sw; po$. Because the SC-per-location constraints also include com in hb (and fr is part of com), we also get $f \xrightarrow{hb} a$. Thus, this execution creates a cycle in hb.

After executing a litmus test, it can be determined which candidate execution occurred by capturing each load value, e.g., to a unique variable, and examining the final state of memory. In some cases, it may be required to add an additional observer thread if the execution contains several steps of co, as the order that the observer thread sees updates can be used to infer the co order.

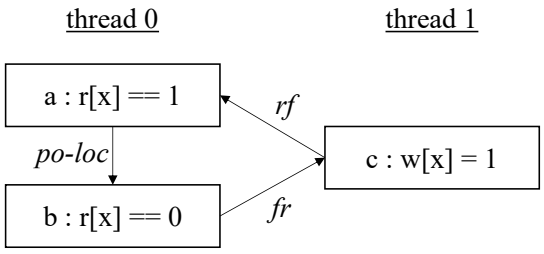
2.3 WebGPU

We apply MCS testing to WebGPU [41], a new, web-based, cross-platform GPU programming framework. Like most GPU frameworks, a WebGPU application consists of two parts: the host, which runs on the CPU; and the device, which runs on the GPU. The host is written in JavaScript and orchestrates the device computation through the WebGPU API [41]. The device program (called a *shader*) is written in the WebGPU Shading Language (or WGSL) [40]. WebGPU compiles to platform specific shading languages and runs on their respective platforms: Metal [8], SPIR-V [22], and HLSL [32] for Apple, Vulkan and DirectX (Microsoft) systems, respectively. As mentioned in Sec. 1.2, the diversity of backends makes WebGPU is a compelling platform to innovate MCS testing methodologies such as MC Mutants.

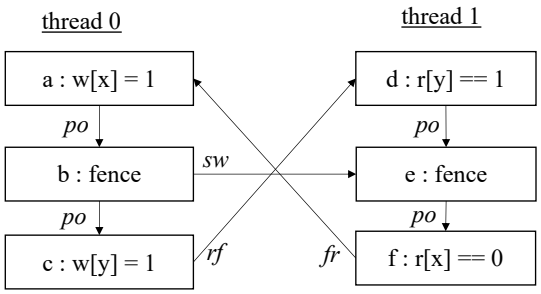
WGSL execution model. The execution model of WGSL is hierarchical; the base unit of execution is a *thread*,[3] which is a single stream of computation. Threads are partitioned into equally sized *workgroups*, which has a queryable size defined on a per-device basis, specified by the host at runtime. Finally, the host specifies the number of workgroups to run the shader with, modeled as a 3-dimensional grid with a queryable size. Threads at different levels of the hierarchy have different abilities to interact. Threads in the same workgroup can synchronize using a *control barrier* (analogous to the CUDA `__syncthreads()` construct).

WGSL memory consistency specification. Using synchronization on GPUs is a complex topic due to the hierarchical programming model. Threads at different levels of the hierarchy use different mechanisms to communicate, and composing these layers requires careful formal analysis, which can often be the subject of entire papers [28]. Our goal in this paper is not to offer a complete formalization of the WGSL MCS; instead, we consider a non-controversial, yet pragmatic, subset of the model and focus on how to effectively test it.

[3]In WGSL, this is called an *invocation*, but we use "thread" in this document as our ideas extend to more general concurrent programming



(a) Disallowed execution of the CoRR litmus test



(b) Disallowed execution of the MP-relacq litmus test

Figure 2: Candidate executions for the litmus tests shown in Fig. 1 illustrating behaviors disallowed by the WebGPU MCS.

As mentioned in Sec. 1.2, our efforts are restricted to inter-workgroup interactions. Unless explicitly mentioned, we will assume that all threads are in different workgroups. WGSL supports atomic operations on atomic memory locations. The atomic operation can be a load, store, or RMWs (e.g. compare-and-swap). WGSL must accommodate the weakest MCS of its targets; in this case, that happens to be Apple's Metal shading language [7]. Its MCS is based on C++, with the restriction that the default, and only, supported memory order is *relaxed*. Given this, WGSL inherits this syntax and the corresponding orderings properties. This specification corresponds to the SC-per-location MCS discussed in Sec. 2.1.

Additionally, previous versions of the WGSL specification defined control barriers to provide release/acquire fence semantics across workgroups. Thus, by using these barriers as fences, this version of the WGSL MCS corresponds to the release-acquire-SC-per-location MCS discussed in Sec. 2.1. As we describe in Sec. 5.4, our work helped discover that this specification was too strong, and later versions of the WGSL specification weakened its MCS to simply SC-per-location on inter-workgroup memory.

3 MC MUTANTS

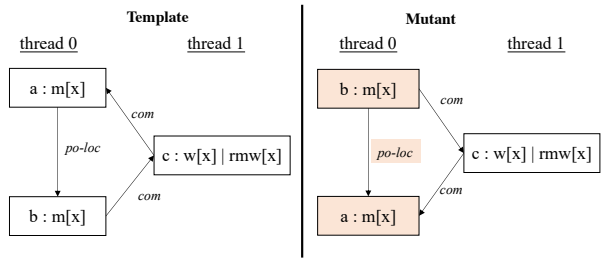
MC Mutants is a methodology for evaluating MCS testing environments. This approach uses cyclic candidate executions from litmus tests, as described in Sec. 2.2. The methodology then mutates cyclic executions by disrupting one of the edges and generates mutated test programs by reconstructing the instructions that led to the

execution. The mutated test now has an execution that is allowed by the MCS, but is closely related to a behavior that is disallowed by the MCS.

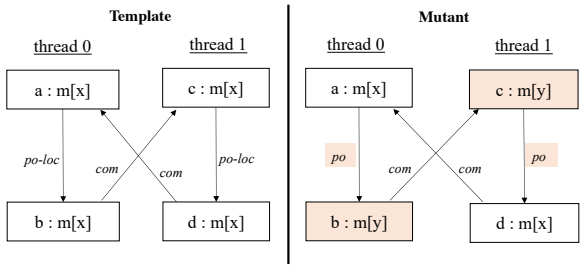
In order to ensure that mutant programs can be reconstructed from the mutant execution, the mutation process focuses on disrupting relations that correspond to the *syntax* of the program, as opposed to relations that arise dynamically. For example, the *po* relation arises from the program syntax, i.e. the order of instructions in the test. Similarly, the *sw* relation arises from fence instructions in the program syntax. This is important for our black-box mutation approach, as our only interface to the platform is with programs (and their syntax). Thus, our mutations encode modifications that we can actually implement. The behaviors allowed under the mutated test are behaviors that will appear in the original test if there is a bug in the implementation of the MCS, since the MCS restricts correct executions to *not* disrupt these relations.

MC Mutants uses a set of *mutator* functions and a function that converts executions into programs. A mutator consist of an abstract happens-before cycle T (similar to those from Alglave et al. [4]) and an edge disruptor process E. The mutator returns L, a set of *conformance* litmus tests that contain disallowed candidate executions based on T, and M, a set of mutated tests containing candidate executions (closely related to the disallowed behaviors in L) for which the behavior is now allowed. A testing environment can be evaluated as follows; the ability of a testing environment to kill (i.e. observe) mutants in M corresponds to its ability to observe an implementation error corresponding to E when running L.

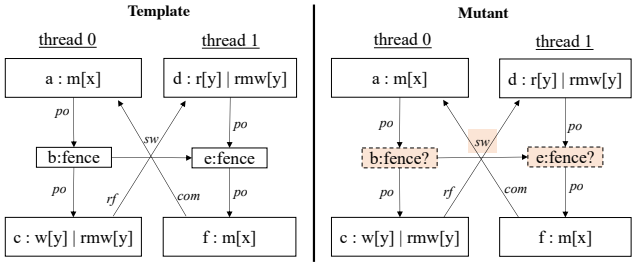
This paper describes three distinct mutators, summarized in Fig. 3. Each mutator enumerates all possible syntactic edge disruptions, i.e. *po*, *sw*, and *po-loc*, and can be seen as a complete set of mutants for these abstract happens-before cycle templates. Overall, there are 20 conformance tests and 32 mutants; the totals broken down by mutator are shown in Tab. 2.



(a) 3-event reversing *po-loc* mutator



(b) 4-event weakening *po-loc* mutator



(c) 4-event weakening *sw* mutator

Figure 3: The three cycles used to instantiate conformance tests and mutants, with conformance test templates shown on the left and mutant templates on the right; *m[x]* signifies an abstract memory event which may be instantiated as a read, write, or RMW. The disrupted events and edges are highlighted in red.

3.1 Mutator 1: Reversing *po-loc* on Three Events

Mutator 1 takes in a three-event cycle, (LHS of Fig. 3a). The template has two threads; thread 0 has two memory accesses ordered by *po-loc*, (events *a* and *b*). Thread 1 executes one memory access (*c*). Executions of litmus tests that correspond to this cycle are disallowed by any MCS that provides SC-per-location, e.g. WebGPU.

The template can be instantiated by concretizing each of the abstract memory events (i.e. m[x]) to either a read, a write, or an RMW. For example, the CoRR litmus test of Fig. 2a is one such instantiation where events *a* and *b* are read events, and event *c* is a write event. Note that this also allows *com* to be refined into one of its constituent relations (i.e. rf, fr, or co).

The conformance set. For an MCS that provides SC-per-location, a set of conformance tests can be generated by instantiating the execution for all combinations of reads, writes, and RMWs such that the cycle can be created. As a *com* relation must contain at least one write, event *c* must be a write or an RMW event. Then, *a* and *b* can be any combination of reads, writes, or RMWs.

When instantiating a test with an RMW event, only the portion of the RMW (the read or the write) that does not change the *hb* relations of the original template are considered. For example, the CoRR litmus test can have the second read in thread 0 replaced with an RMW but not the first. This is because an RMW is conceptually a read followed by a write. The second read in thread 0 can have a trailing write while maintaining the *hb* cycle. If an RMW was instantiated with the first read, then there would conceptually be a write in between the first and second reads, which could interfere with the cycle. While there may be multiple RMW variants with the same memory structure as one of the original four tests, for our evaluation only the variant with the maximum number of plain loads and stores replaced with an RMW is included.

Finally, writes are concretized using a unique increasing value to store and an observer thread is included for the special case where all memory events are concretized as writes, as they must observe a specific chain of *co*.

Table 2: The mutators defined in Fig. 3 and the number of conformance tests and mutants generated by each.

Mutator	Conformance Tests	Mutants
Reversing *po-loc*	8	8
Weakening *po-loc*	6	6
Weakening *sw*	6	18
Combined	20	32

The edge disruptor. Mutator 1 disrupts the *po-loc* edge between a and b by swapping them in program order (see the right hand side of Fig. 3a). This removes the disallowed cycle from the original template, i.e. the behavior is allowed on a system that provides only SC-per-location. In fact, the behavior is allowed under sequential consistency given the execution order of b, c, a.

For a testing environment to kill these mutants, it requires the fine-grained interleaving of an event from thread 1 between two events of thread 0. While it may seem straightforward, the behavior is not always easily observed, especially on GPUs. Pilot experiments show that this fine-grained interleaving is only observable on one out of the four GPUs that we evaluated in this study if they are executed in a test environment without added stress.

This mutant captures the cases when the underlying system, for whatever reason (e.g. re-order buffers), can re-order events a and b. If a testing environment cannot expose such fine-grained interleavings, it likely will not be effective at exposing MCS bugs. In fact, this mutant has a death rate nearly identical to the CoRR bug observed on the Apple system with an Intel GPU, discussed in Sec. 5.5, showing that the ability to kill this mutant corresponds closely to observing a real MCS bug.

The mutants. Similar to the conformance tests, the mutant tests can be obtained by instantiating the disrupted template (right-hand side of Fig. 3a) with all possible combinations of memory accesses (reads, writes, or RMWs).

3.2 Mutator 2: Weakening *po-loc* on Four Events

Mutator 2 (Fig. 3b) has a two-thread four-event cycle. Both threads have memory accesses ordered by *po-loc* and cross-thread *com* edges. An MCS that provides SC-per-location disallows these executions.

The conformance set. Following the rule that the *com* relation must contain at least one write, the mutator function instantiate a set of 6 tests. As with Mutator 1, writes are concretized using a unique increasing value and an observer thread is included for the special case where all memory events are concretized as writes.

The edge disruptor. This disruption weakens *po-loc* to *po* by having b and c operate on a second location, y, instead of x. This turns the test into one of the weak memory tests in [4]. Killing the mutant then means observing a weak behavior that is allowed under a relaxed MCS, but has been shown to require stress to observe [24].

This disruptor captures errors that might occur when memory locations are aliased or computed dynamically, as an implementation may not correctly determine that the memory locations are the same. Indeed, as Sec. 5.4 shows, one of these tests corresponds to a coherence issue previously observed on NVIDIA GPUs.

The mutants. By instantiating the disrupted template in the same way as the initial template, the mutator function instantiates six mutants.

3.3 Mutator 3: Weakening *sw* on Four Events

The final mutator also uses a two-thread four-event cycle, but adds release/acquire fences (shown in Fig. 3c). This behavior is disallowed in the MCS release-acquire-SC-per-location.

The conformance set. Using plain atomic loads and stores, a set of 3 tests are initialized. These correspond to the MP-relacq litmus test execution from Fig. 2b as well as two other common litmus tests execution, called load buffering and store in prior work [6]. Since release/acquire synchronization requires a store following a fence to synchronize with a read preceding a fence, events c and d must be a write and a read, respectively. This precludes the direct addition of other common weak memory litmus tests such as store buffering. However, like in Mutator 1 more tests can be instantiated using RMWs.

Replacing c and/or d with an RMW achieves the required synchronization. Using this strategy, a further 3 tests are generated, corresponding to the store buffer, read, and 2+2 write weak memory litmus test executions [6]. In an MCS that provides sequentially consistent fences, the addition of RMWs would be unneeded for the generation of these three tests, but even so we believe using release/acquire fences and RMWs to "mimic" sequentially consistent behavior is an interesting feature of a MCS to understand and test.

The edge disruptor. To disrupt this template *sw* is weakened, removing the necessary synchronization for disallowing the weak behavior. To weaken *sw*, either one or both of the fences is removed. Removing fences is analogous to the MP-relacq bug we discovered and described in Sec. 1.1, in which atomic operations were mistakenly weakened in an intermediate representation. Killing the mutant then depends on a stressful environment that can reveal weak memory behaviors, even in the presence of partial synchronization.

The mutants. Unlike the previous 2 mutators, each combination of memory accesses generated using the template leads to 3 instantiated mutants, as the mutant can have one or both fences removed.

3.4 Observing Mutant Behavior

As briefly mentioned in Sec. 1.2, the mutation score is only useful as an evaluation metric if the mutant behavior is observable on the devices being evaluated. In some cases, the specification is more permissive than the implementation, and as such, an implementation may give a low mutation score, even if it is being thoroughly tested. For example, very few relaxed behaviors are observable on an x86 device, however, a language that targets x86 (e.g. C++) might allow many relaxed behaviors. Applying MC Mutants to test C++ conformance on x86 as-is would likely provide a low mutation score.

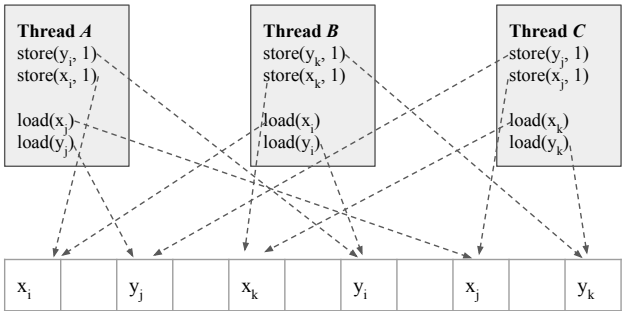


Figure 4: A visualization of how multiple threads coordinate to run parallel instances of litmus tests.

In our study, we find that the four GPUs we evaluate are able to observe most of the mutant behaviors (83.6%), thus we keep our mutation tests as-is. However, if this was not the case (e.g. as in the C++ and x86 example), then the mutation tests should be pruned. That is, each mutant test m should be analyzed under a *precise* model of the expected observed behavior of the implementation. For example, on x86 this would be the TSO memory model [33]. If m is not observable on the implementation, then it can be pruned from the mutant test cases.

4 IMPROVING TESTING ENVIRONMENTS

Litmus tests are run repeatedly to account for the non-determinism of concurrency. They are often executed in a larger context, called a *test environment* that encompasses various parameters, e.g. memory locations and thread affinity, which can influence the rate that different test behaviors are observed. Using MC Mutants, testing environments can be evaluated using a mutation score and mutant death rates of the mutants generated in Sec. 3.

We now describe two improvements to the state of the art MCS test environments: (1) a Parallel Test Environment (PTE), which uses the thousands of threads available in a GPU to run many test instances in parallel, and (2) an MCS Test Confidence, which provides statistical confidence around the ability of a test environment to kill a mutant given a confidence threshold or time budget.

4.1 Parallel Test Environment (PTE)

MCS testing typically runs at most a few test instances of a test at a time, since the technique was initially developed for small core-count CPUs. MCS testing on highly parallel devices, e.g. GPUs, inherited single-instance design [1, 24] due to the difficulties in efficiently and safely scaling the number of parallel tests, especially under tunable stress testing parameters. Additionally, because GPUs sequentialize divergent control flow execution within a warp, efficient parallel tests should limit the amount of control flow divergence in the testing program. Alas, single-instance design poorly leverages the large number of threads enabled by highly parallel devices.

To address this, we propose a Parallel Test Environment (PTE), where a number of workgroups are designated as *testing workgroups*, in which constituent threads coordinate to run many instances of a litmus test in parallel. For example, two testing workgroups, running with 256 threads each, can run 512 instances of a two-threaded litmus test per iteration. As described in Sec. 5.2, this leads to orders of magnitude improvements in testing efficiency.

Implementation. The number of test instances per iteration is calculated by multiplying the number of testing workgroups by the number of threads per workgroup. Each test instance t_i is then assigned memory locations to operate on. In a weak memory test like MP, t_i will be assigned locations x_i and y_i, while in a coherence test like CoRR, t_i will be assigned only x_i. Next, two test instances t_i and t_j are assigned, in that order, to a thread A. A runs thread 0's instructions from t_i and thread 1's instructions from t_j. The other half of t_i and t_j are assigned to some other thread (not necessarily the same one) in the opposite order of their assignment on thread A, guaranteeing that all threads' instructions are executed for both t_i and t_j.

Figure 4 shows a configuration of test instances and memory locations assigned to threads for the MP litmus test, with fences elided. Note that no pair of test instances are assigned to the same two threads, increasing the different thread interactions of a test run. Memory locations are randomly distributed throughout memory.

Test Instance Assignment. There are two difficulties in implementing parallel tests on GPUs: (1) maintaining simple (i.e., non-diverged) control flow to avoid sequentialized execution and (2) pairing threads for each test instance. PTE uses a parallel permutation function that requires only a few operations per thread, has no divergent control flow, and avoids simple patterns, such as mapping a value n to $n + 1$, which has been shown to be ineffective in prior work [24]. Specifically, the function works as follows: for each value, v, in a sequence of N consecutive natural numbers, the function produces the result $(vP) \mod N$, where P is a number that is co-prime to N.

As an example, thread A in Fig. 4 might be index i in the total list of threads, and would execute the first set of instructions for t_i, storing 1 to x_i and y_i. After executing the permutation function A would then execute the second set of instructions for t_j, loading x_j and y_j. As some other thread had its native thread id associated with the store instructions of t_j, each test instance is fully executed.

For a test instance t_i, the permutation function can also be used to spread memory locations x_i and y_i across the testing region by having x's location associated directly with a test instance t_i, and permuting y's location. Prior work has shown that using random native thread ids for testing can increase weak observations [1, 35]. The permutation functions achieve a similar goal, stressing cache protocols that must handle thousands of threads accessing cache lines which must be kept coherent.

Lastly, if there are only two testing workgroups, at least one of A, B, or C will be in different workgroups, and if there are three or more workgroups, A, B, and C will all be in different workgroups. Architectures and scheduling algorithms differ across GPU vendors, so it may be advantageous to have workgroups communicate in ways that vary spatially and temporally. Therefore, test instances are *striped* across workgroups, so if thread 0 in workgroup A communicates with some thread in workgroup B, thread 1 in workgroup B communicates with some thread in workgroup C.

Algorithm 1 Merging Test Environments

```
1:  // t: the mutant test to find a final environment for
2:  // E: the set of environments that the mutant ran in
3:  // D: the set of devices that ran the mutant (across the environments E)
4:  // r: the reproducibility score target, 0 < r < 1
5:  // b: the maximum time the test should run, b > 0
6:  function MergeEnvironments(t, E, D, r, b)
7:      ceilingRate ← ⌈−log_e(1−r)⌉ / b
8:      e_r ←Ø
9:      n_r ← 0
10:     minRate_r ← ∞
11:     for e ∈ E do
12:         n_c ← 0
13:         minRate_c ← ∞
14:         for d ∈ D do
15:             // rate() calculates the death rate of t on d in e
16:             rate_c ← rate(t, d, e)
17:             if rate_c ≥ ceilingRate then
18:                 n_c ← n_c + 1
19:             end if
20:             if rate_c > 0 then
21:                 minRate_c ← min(minRate_c, rate_c)
22:             end if
23:         end for
24:         if n_c > n_r ∨ (n_c == n_r ∧ minRate_c > minRate_r) then
25:             e_r ← e
26:             n_r ← n_c
27:             minRate_r ← minRate_c
28:         end if
29:     end for
30:     return e_r
31: end function
```

Additional parameters. Prior work [1, 24] has proposed additional heuristics and parameters for testing environments, such as shuffling thread affinities, and utilizing extra threads to stress memory. These heuristics are incorporated into PTE, so that the parallel tests can also be subject to these testing techniques. Altogether, prior work [24] has provided 17 parameters that can be changed to create different testing environments. It is infeasible to examine the full space of combinations of these parameters, so a number of random configurations are run in order to find effective test environments, and MC Mutants can be used to evaluate their effectiveness.

4.2 MCS Test Confidence

We present our second contribution to MCS testing environments, which is built on MC Mutants: MCS Test Confidence. Given a set of tests from MC Mutants (both the conformance and mutated tests), MCS Test Confidence explores the trade-off space between testing time and the probability that the conformance tests will reveal a bug captured by MC Mutants.

Prior work [24] derived an equation that relates the number of observations to the probability that the observation will be observed in subsequent runs. If a litmus test is executed N times, and a behavior of interest is observed x times, then the probability that a subsequent run of N iterations will reveal at least one behavior of interest is given by $1 - e^{-x}$. For example, if x is 3, then there is a 95% probability that another run of N iterations will reveal that behavior of interest; we call that probability the *reproducibility score*.

MCS Test Confidence builds on that insight and combines it with MC Mutants. For example, if a testing environment can provide an average mutant death rate of 1 per second for each mutant, then the 20 conformance tests of Sec. 3 require 3 seconds of testing time each (for a total of 1 minute of testing time) in order to have a reproducibility score of 95%, i.e. a 95% probability that the mutants will be killed in a subsequent test run.

This analysis can be applied to evaluate a testing environment, especially when deciding how long to run each test for. The results can then be applied when curating MCS tests for inclusion in a *conformance test suite* (or CTS). A time budget for testing can be selected such that it provides sufficient confidence across many devices.

Per test specialized testing environments. Ideally, a test environment can be hyper-tuned per test and per device . However, when curating tests for a CTS, it is only feasible to create a testing environment per test for each test, as these are known at contribution time; there may be no a priori knowledge of the devices the CTS will execute on. Therefore, a test environment for a specific test needs to be effective on multiple devices. We now discuss how the MCS Test Confidence approach can be used to create such a per-test specialized testing environment.

For each mutant from MC Mutants, results are collected by running it in a set of identical test environments, generated by randomly instantiating parameters, on a variety of devices. Then, one single environment per test is chosen, using the function shown in Alg. 1. The idea is to choose the test environment that maximizes the number of devices on which the mutant death rate is higher than some ceiling rate, which can be calculated using the desired reproducibility score and time budget targets of the MCS Test Confidence equation. The equation in line 7 is the inverse of $1 - e^{-x}$, and calculates the number of behaviors that need to be observed divided by the time budget to get a final rate.

If two environments end up killing the mutant at a high enough rate on the same number of devices, the environment that maximizes the minimum non-zero rate is chosen. For mutants that reach the ceiling rate on all devices, this increases the minimum reproducibility score beyond the target on that mutant. On mutants that did not reach the target rate on at least one device, the chance that the mutant will be reproduced on that device despite missing the target reproducibility score is maximized.

Another useful property of breaking ties using the minimum non-zero rate is that it leads to *stable* environments. If an environment chosen by the algorithm for a given reproducibility score r and time budget t meets or exceeds the target mutant death rate on all devices, the same environment will be chosen by another run of the algorithm that uses values r', t' such that $r' \leq r$ and $t' \geq t$.

One other consideration when calculating reproducibility scores is the *total reproducibility score*. Consider a CTS with 20 MCS tests and associated test environments, each which have been found to kill their associated mutants with 95% reproducibility. The chance that on any specific run all 20 mutants are killed is then $.95^{20}$, or 35.8%. The CTS would need to be run three times, on average,

Table 3: The devices included in our study, along with how many compute units (CUs) they contain and a short name we will use throughout the text.

Vendor	Chip	CUs	Type	Short Name
NVIDIA	GeForce RTX 2080	64	Discrete	NVIDIA
AMD	Radeon Pro 5500M	24	Discrete	AMD
Intel	Iris Plus Graphics	48	Integrated	Intel
Apple	M1	128	Integrated	M1

to observe a run where all 20 mutants are killed. Increasing the per test reproducibility allows for a much more stable CTS; a per test reproducibility score of 99.999% and 20 tests leads to a total reproducibility score of 99.98%, meaning that a test run will only fail to kill all mutants on 1 out of 500 runs.

5 EVALUATION

We now evaluate our testing environment innovations using MC Mutants and compare to prior work.

5.1 Experimental Setup

As described in Sec. 3, we instantiate 20 conformance tests and 32 associated mutants. We evaluate test environments on the 32 mutants and measure their efficacy using the mutation score and mutant death rate. As mentioned throughout, our case-study specification is WebGPU. We empirically found that Google Chrome had the most stable WebGPU implementation, and thus, we used it exclusively. We evaluate on four GPU devices, summarized in Tab. 3. These devices span four vendors and sample both integrated and discrete GPUs to provide a broad foundation upon which to evaluate MC Mutants and parallel testing environments. We note that WebGPU is not yet supported on mobile devices.

We tune testing environments by randomly generating parameters and executing the mutants in each environment. We complete tuning for Parallel Test Environments (**PTE**), and the single instance techniques proposed in prior work [24] (**SITE**). We note that **SITE** is strictly an improvement on the testing strategies used in 'litmus' [5] since it includes all of the 'litmus' testing parameters (documented in [5, Sec. 3]), with the exception of parallel testing, which has not been efficiently implemented on GPUs before our work. SITE additionally includes memory stressing threads, which 'litmus' does not.

We perform a tuning run over 150 testing environments, running the SITEs for 300 iterations (executing 300 test instances) and the PTEs for 100 iterations (executing 125K test instances on average). We run more instances on SITE tests as to provide them more opportunities to kill mutants. The number of testing environments was chosen based on the ability to do tuning runs in a reasonable amount of time (e.g. overnight).

In addition to single instance and parallel testing environments, we also evaluate two testing environments that perform no stress heuristics. The first, **SITE Baseline**, executes a single instance of the mutants for 300 iterations without any added stress and uses 32 workgroups, of which two workgroups have one thread execute each thread in the mutant. The second, **PTE Baseline**, runs parallel instances of the mutants for 100 iterations without any added stress and uses 1024 testing workgroups, each of which includes 256 threads.

The total tuning time, across all four devices, was 9.27 hours for the parallel environments and 16.38 hours for the single instance environments. In contrast, prior work ran tuning for multiple days on each chip [24]. Our focus on real-world impact in the form of adoption into conformance test suites encourages us to explore test environments that quickly and reliably kill mutants.

5.2 Mutation Score and Mutant Death Rates

Figure 5 outlines the mutation score, defined as the total number of mutants killed in at least one test environment, and the average mutant death rate, defined as the average of the maximum death rates of each mutant in the specified category. Specifically, figures 5a, 5c and 5e outline the mutation scores for each mutator, while figures 5b, 5d, and 5f outline the average mutant death rate for each mutator. Figures 5g and 5h show the mutation score and rate averaged across all mutators, while figures 5i and 5j show the mutation scores and mutant death rates averaged across all mutation types and devices.

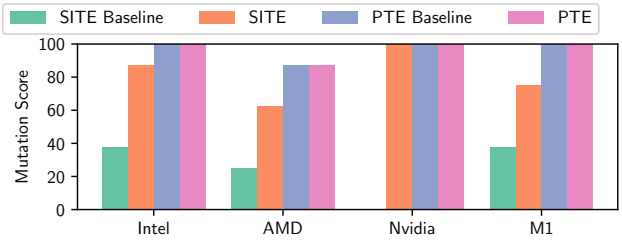
5.2.1 Results summary. We first summarize the performance of PTE compared to SITE in terms of mutation score and mutant death rate. We observe that PTE outperforms SITE by both mutation score and by mutant death rate. Across all devices and all tests, PTE achieves a mutation score of 83.6%, whereas SITE achieves a score of 46.1%, an improvement of 81.4% (see Fig. 5i). Figure 5i also outlines the importance of stress; SITE-baseline only achieves a mutation score of 6.3%, while PTE-baseline only achieves a mutation score of 72.7%.

PTE is even more impressive when considering mutant death rate, shown in Fig. 5j. PTE achieves an average mutant death rate across all tests of 35K mutants per second, 2731× better than the mutant death rate of SITE.

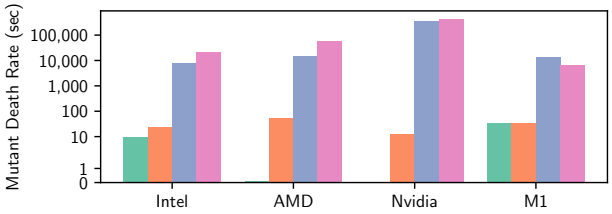
5.2.2 Detailed Analysis. Next, we provide a detailed analysis over the results shown in Fig. 5.

Mutation score. PTE achieves a higher mutation score on 3/4 architectures (AMD, NVIDIA, and M1), which enables PTE to provide a higher aggregate mutation score than SITE. In particular, SITE does very poorly on NVIDIA and M1, where it is unable to kill any weakening *po-loc* mutants (see Fig. 5c), and only rarely kills weakening *sw* mutants (see Fig. 5e). SITE outperforms PTE on Intel in our experiments, but this is not a fundamental limitation of PTE since SITE is a strict subset of PTE. There exists a PTE equivalent to each SITE, as PTE allows environments that execute only one instance of the test per iteration. Rather, our testing did not identify a PTE that matches the outperforming SITE because the state space of PTE is much larger.

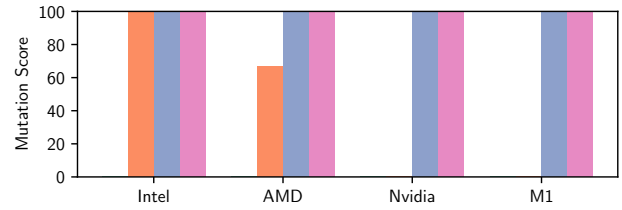
Mutant death rate. PTE achieves a higher mutant death rate on all architectures. SITE does poorly on NVIDIA and M1 (see Fig. 5h), largely because of a rate of 0 mutants per second for *sw* and weakening *po-loc* mutants.



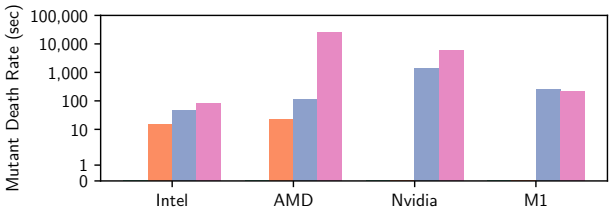
(a) Reversing *po-loc* mutation scores (8 mutants)



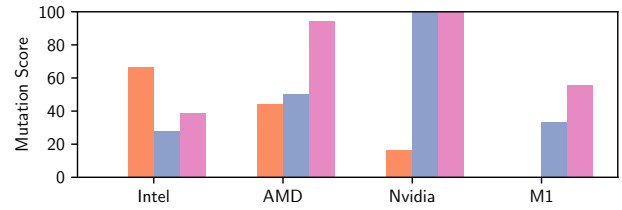
(b) Reversing *po-loc* average mutant death rates



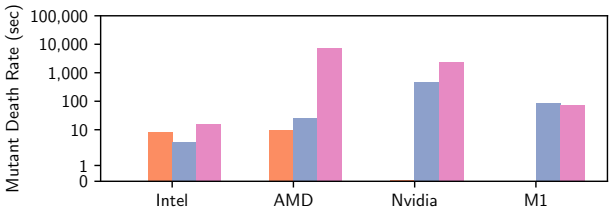
(c) Weakening *po-loc* mutation scores (6 mutants)



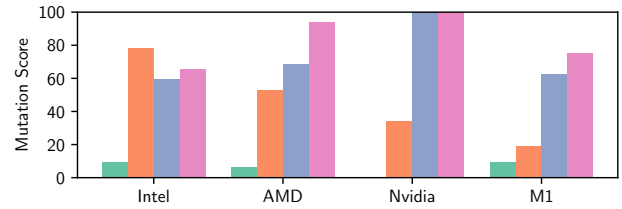
(d) Weakening *po-loc* average mutant death rates



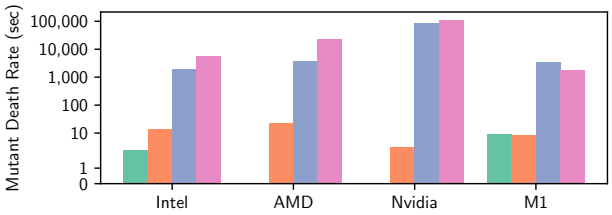
(e) Weakening *sw* mutation scores (18 mutants)



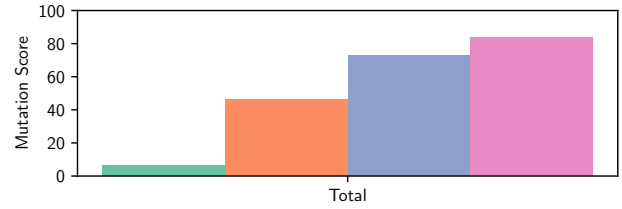
(f) Weakening *sw* average mutant death rates



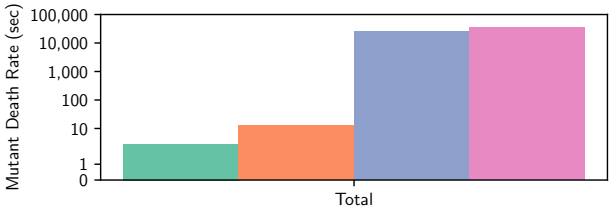
(g) All tests per device mutation scores (32 mutants)



(h) All tests per device average mutant death rates



(i) All devices/tests mutation scores (128 mutants)



(j) All devices/tests average mutant death rates

Figure 5: Mutation scores and mutant death rates.

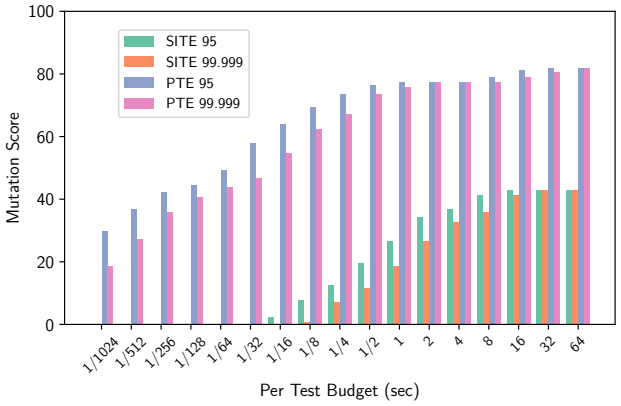


Figure 6: Analyzing the impact of time budgets and reproducibility targets on mutation scores.

Stress and PTE. We observe that stress improves the performance of PTE in the aggregate, improving the mutation score from 72.7% for PTE-baseline to 83.5% for PTE and improving the average mutant death rate from 24,400 for PTE-baseline to 35,000 for PTE. However, these improvements are not uniform across devices; stress offers some benefit to PTE mutation scores and mutant death rate on Intel and AMD, no benefit to PTE in terms of mutation scores on NVIDIA, and only negligibly benefits PTE in terms of mutant death rate on NVIDIA. Finally, stress improves the mutation score of PTE on M1, but decreases the mutant death rate.

Effectiveness across mutators. The highest mutant death rates occur on reversing *po-loc* (Fig. 5b) mutants, with average rates of 22K, 58K, 428K, and, 6.5K mutants killed per second on Intel, AMD, NVIDIA, and M1 respectively. The lowest rates occur on the weakening *sw* (Fig. 5h) mutants. This is to be expected as the reversing *po-loc* mutator produces mutants that are allowed under sequential consistency, and requires only fine-grained interleavings to kill the mutants. On the other hand, killing a mutant from the weakening *sw* mutator requires observing a weak behavior, potentially with partial synchronization, i.e. a fence on one of the threads.

5.3 Reproducible Mutants

We next evaluate whether MCS test confidence enables us to create a single test environment for each litmus test that is suitable for testing multiple devices, e.g. as is required by a CTS (see Sec. 4.2).

Fig. 6 shows the results of combining test environments per test using different time budgets and two reproducibility score targets, 95% and 99.999%. 95% reproducibility corresponds to killing a mutant 3 times in the allotted time budget and leads to a total reproducibility of 36.5% with 20 tests, as explained in Sec. 4.2. Due to this relatively low reproducibility, we consider 95% to be a lower bound on an appropriate reproducibility score target. We choose 99.999% as the maximum target; it corresponds to running a specific litmus test once per minute for a year and only seeing 5 non-reproducible runs, and leads to a total reproducibility of 99.98% with 20 tests.

PTE achieves an 82% mutation score with a 64 second time budget and a reproducibility target of 99.999%. In contrast, SITE only achieves a mutation score of 43% with the same constraints. PTE testing environments are almost twice as effective at killing mutants when using a single test environment across architectures.

The difference between PTE and SITE is even more striking at lower testing time budgets. SITE mutation scores degrade rapidly as the time budget decreases, until the rate drops to zero at 1/32 of a second. In contrast, PTE is still able to kill 36% of mutants with 95% reproducibility when given a time budget of 1/1024 of a second. Moreover, PTE achieves roughly the same mutation score as SITE's maximum score (43%) when given a time budget of only 1/64 of a second–i.e. PTE achieves the same mutation score with $(1/4096)^{\text{th}}$ the time budget. This is an important result for large software CTS consideration, as testing time is at a premium.

Table 4: The Pearson Correlation Coefficient (PCC) between killing mutants and observing real bugs. Values for PCCs range from 0 to 1, and while it varies across domain, correlations greater than .8 are generally considered to be very strong.

Vendor	Failed Test	Mutant Type	PCC
Intel	CoRR	Reversing *po-loc*	.996
AMD	MP-relacq	Weakening *sw*	.967
NVIDIA	MP-CO	Weakening *po-loc*	.893

5.4 Correlation Analysis

This work discovered two bugs which we reported to the platform maintainers. One of these, observing disallowed behaviors in the MP-relacq test, was only uncovered on AMD devices using PTE. This bug led to both a fix to AMD Vulkan drivers and a change to the WebGPU specification [9]. The second, observing disallowed behaviors in the CoRR litmus test on Intel GPUs running WebGPU's implementation in Chrome over Apple's Metal API, has been reported to Apple.

We performed a correlation analysis between real-world bugs and MC Mutants to determine whether killing a mutant correlates with the ability to identify real MCS bugs. We use our two new bugs and recreated a coherence violation in NVIDIA Kepler GPUs [1]. This previous coherence violation manifests as a violation of one of the weakening *po-loc* (Fig. 3b) conformance tests, in which thread 0's memory accesses are writes, and thread 1's memory accesses are reads. We call this test message passing coherence (MP-CO).

We executed the conformance test that reveals each bug and the test's associated mutant(s) for 100 iterations in 150 random parallel testing environments. Tab. 4 shows the Pearson Correlation Coefficient (PCC), which measures linear correlation and ranges from -1 to 1, between the mutant death rate and the observed bug rate across all the environments for each bug.

The results indicate high correlation between killing mutants and identifying MCS bugs. In particular, the PCC for each bug is larger than .89; according to the Student's t-test, the probability of such a PCC occurring due to random chance is less than 10^{-6}%. Thus, we show that a testing environments ability to kill mutants (generated by MC Mutants) correlates to its ability to find real bugs in the conformance test suite (also generated by MC Mutants). We

emphasize that this case study used a mutant from each of our three mutators and spans three GPU vendors (AMD, NVIDIA, and Intel), showing that our mutators are all effective and the approach is portable across the diverse backends targeted by WebGPU.

5.5 Impact

Our work led to the discovery and reporting of two MCS bugs to platform maintainers. Additionally, our work on improving test environments has led to the official WebGPU CTS adopting our PTE for MCS testing [26]. The WebGPU CTS is part of Chromium's continuous integration process as development on WebGPU compilers continue, and runs on a variety of GPUs (e.g. NVIDIA, Intel) and platform implementations (e.g. Vulkan, Metal, Direct3D). We expect the diversity of devices to grow as more browsers and platforms, especially mobile ones, implement the WebGPU standard. At least one parallel test has also been included in the Vulkan CTS as a result of the bug found on AMD drivers [23].

6 RELATED WORK

Memory Model Testing. Memory model testing started with work that executed long hand-written sequences of memory accesses or generated pseudo-random programs and analyzed the output to determine what behaviors were allowed by a machine [17, 39]. Litmus tests have been formalized [5] and synthesizers from templates of relations have been developed [4]. Further work on synthesizers can generate litmus tests directly from an MCS [38]. Litmus tests have been used to reveal the details of GPU architectures and propose formal models [1]. GPU cache coherence has been tested using tools that autonomously generate random patterns of memory requests and immediately detect any inconsistencies [36].

In contrast, our work designs a methodology to evaluate testing environments for MCS testing: MC Mutants. We evaluate prior work [24] using MC Mutants and provide quantitative results on its effectiveness. Furthermore, we provide two improvements to testing environments: (1) we improve testing efficiency, similar to [31], but with more generality as we execute generic litmus tests rather than bespoke algebraic sequences; (2) we provide a novel method to curate conformance test suites with quantitative analysis about their ability to catch bugs related to mutations, which has not been discussed in prior work.

Mutation testing. Mutation testing was originally designed as a form of white-box testing for sequential programs [15]. It has since been applied to concurrent programs while enforcing deterministic test outcomes [14]. Recent work has examined white-box mutation testing in the context of non-deterministic, flaky tests, using a test re-running strategy to increase the confidence in mutation scores [34]. Our approach similarly addresses non-determinism; however, we evaluate testing environments rather than testing suites and we target memory consistency runtime platforms rather than parallel software.

Black-box testing is used to test a system without modifying its internal behavior. Black-box tests are usually derived from a system's specification, which led researchers to develop specification based mutation testing [13], where test data is generated from a specification that has itself been mutated. Over the years, specification based mutation testing has been applied to predicate calculus, network protocols, finite state machines, and security policies [20]. Our approach builds on these foundations using well-developed MCS formalisms.

Related Memory Model Specifications. Many ISA and language level MCS have been formalized, including x86 [33], ARM and Power [2], and C++ [11], to name a few. Higher level languages have also started to formalize specifications for accessing shared memory, including Javascript [37]. Work has also been done on formalizing memory models for heterogeneous devices [18, 28].

While we provide a simple formalism of WebGPU (Sec. 2.3) it is largely based off the Vulkan memory model [19] and simple instatiations of the parameterized memory models in [6]. We leave a more complete formalism of WebGPU's MCS to future work.

7 CONCLUSION

In this paper, we present MC Mutants: the first principled methodology for evaluating MCS testing environments. We then use MC Mutants to evaluate two novel improvements to MCS testing environments: a parallel testing strategy that is orders of magnitude more efficient than prior work, and a testing confidence approach which can explore the trade-off space between testing time and confidence. We evaluate our approach in the new WebGPU framework and identified two MCS bugs, one of which led to a significant specification change. Finally, our approach has been used to curate MCS tests that have been incorporated into the WebGPU CTS.

8 DATA-AVAILABILITY STATEMENT

The code used to collect our results and the data itself are available openly [27].

9 ACKNOWLEDGMENTS

We would like to thank the reviewers, whose feedback strengthened this paper and improved its clarity. We thank the Khronos Memory Model Task Sub-group (TSG), who have provided feedback and support on our work. We specifically thank Jeff Bolz (NVIDIA) for his help in adding one of our parallel tests to the Vulkan CTS, Tobias Hector (AMD) for validating the message passing issue on the AMD implementation, and Robert Simpson (Qualcomm) for organizing our discussions with the Memory Model TSG. We also thank Kyle Little for his help in reviewing our artifact. This work was supported by a gift from Google.

REFERENCES

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/2694344.2694391

[2] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The Semantics of Power and ARM Multiprocessor Machine Code. In *Workshop on Declarative Aspects of Multicore Programming*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1481839.1481842

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software Verification for Weak Memory via Program Transformation. In *European Symposium on Programming, ESOP (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 512–532. https://doi.org/10.1007/978-3-642-37036-6_28

[4] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, 258–272.

[5] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. https://doi.org/10.1007/978-3-642-19835-9_5

[6] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

[7] Apple. 2021. Metal Shading Language Specification, Version 2.4. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf.

[8] Apple. 2022. Metal. https://developer.apple.com/documentation/metal/. Retrieved March 2022.

[9] Alan Baker. 2021. Fixes to memory model for barriers and atomics. https://github.com/gpuweb/gpuweb/pull/2297.

[10] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, 634–648. https://doi.org/10.1145/2837614.2837637

[11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/1926385.1926394

[12] James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models From Framework Sketches and Litmus Tests. In *Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/3062341.3062353

[13] Timothy A. Budd and Ajei S. Gopal. 1985. Program Testing by Specification Mutation. *Computer Languages* 10, 1 (1985), 63–73. https://doi.org/10.1016/0096-0551(85)90011-6

[14] R. Carver. 1993. Mutation-Based Testing of Concurrent Programs. In *Proceedings of IEEE International Test Conference - (ITC)*. 845–853. https://doi.org/10.1109/TEST.1993.470617

[15] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[16] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/106972.106997

[17] S. Hangal, D. Vahia, C. Manovit, J.-Y.J. Lu, and S. Narayanan. 2004. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *International Symposium on Computer Architecture (ISCA), 2004.* 114–123. https://doi.org/10.1109/ISCA.2004.1310768

[18] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-Race-Free Memory Models. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery. https://doi.org/10.1145/2541940.2541981

[19] Jeff Bolz. 2022. Vulkan Memory Model. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#memory-model.

[20] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[21] Khronos Group. 2021. SPIR-V Specification Version 1.6, Revision 1. https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.html.

[22] Khronos Group. 2022. Vulkan 1.3 Core API.

[23] Khronos Groups. 2022. Test message passing using permuted indices. https://github.com/KhronosGroup/VK-GL-CTS/commit/0f0473342f80ab4ddcdd3588c034fc41b285e6ca.

[24] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of Empirical Memory Consistency Testing. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428294

[25] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[26] Reese Levine. 2022. Add comprehensive memory model tests. https://github.com/gpuweb/cts/pull/1330.

[27] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2022. MC Mutants Artifact. https://doi.org/10.5281/zenodo.7196061

[28] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/3297858.3304043

[29] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. 2017. RTLcheck: Verifying the Memory Consistency of RTL Designs. In *International Symposium on Microarchitecture, MICRO*. ACM. https://doi.org/10.1145/3123939.3124536

[30] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. *CoRR* abs/1611.01507 (2016). arXiv:1611.01507

[31] Themis Melissaris, Markos Markakis, Kelly Shaw, and Margaret Martonosi. 2020. PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing. In *International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO50266.2020.00037

[32] Microsoft. 2020. Programming guide for Direct3D 11. https://docs.microsoft.com/en-us/windows/win32/direct3d11/dx-graphics-overviews.

[33] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics*. 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

[34] August Shi, Jonathan Bell, and Darko Marinov. 2019. *Mitigating the Effects of Flaky Tests on Mutation Testing*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/3293882.3330568

[35] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Programming Language Design and Implementation PLDI*. ACM. https://doi.org/10.1145/2908080.2908114

[36] Tuan Ta, Xianwei Zhang, Anthony Gutierrez, and Bradford M. Beckmann. 2019. Autonomous Data-Race-Free GPU Testing. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 81–92. https://doi.org/10.1109/IISWC47752.2019.9042019

[37] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and Mechanising the JavaScript Relaxed Memory Model. In *Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery. https://doi.org/10.1145/3385412.3385973

[38] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/3009837.3009838

[39] William W. Collier. 1994. ARCHTEST. http://www.mpdiag.com/archtest.html.

[40] World Wide Web Consortium (W3C). 2022. WebGPU Shading Language: Editor's Draft. https://gpuweb.github.io/gpuweb/wgsl/.

[41] World Wide Web Consortium (W3C). 2022. WebGPU: W3C Working Draft. https://www.w3.org/TR/webgpu/.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains information for both collecting analyzing the results we present in the paper. On the collection side, we provide the means to run the exact experiments included in the paper. There are four devices included in the main study in the paper: an NVIDIA GeForce RTX 2080, an AMD Radeon Pro, an Intel Iris Plus Graphics, and an Apple M1. Additionally, a device running using Nvidia's Kepler architecture (such as an NVIDIA GeForce GTX 780) was used to reproduce a bug found in earlier work. Using the exact devices from the paper will show very similar results to ours, but any GPU can be used to evaluate the way in which we collect and analyze data.

On the analysis side, we include the results from running the experiments on the four devices in the paper, as well as the analysis tools we used to generate the main figures in the paper. This is all done using python scripts, with a couple common packages installed using pip: numpy, matplotlib, pandas. Additionally, reviewers can use the analysis tools to check the results of their own data collection, and we'd encourage them to send us the results!

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Google Chrome, OSX, Windows, Linux
- **Hardware:** NVIDIA GeForce RTX 2080, AMD Radeon Pro, Intel Iris Plus Graphics, Apple M1.
- **Execution:** 2-4 hours
- **Metrics:** test time, number/rate of weak behaviors
- **Output:** json
- **Experiments:** Python scripts
- **How much disk space required (approximately)?:** 2-4 MB
- **How much time is needed to prepare workflow (approximately)?:** 10-15 minutes
- **How much time is needed to complete experiments (approximately)?:** 2-4 hours
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Yes, DOI: 10.5281/zenodo.7196061

A.3 Description

A.3.1 How to access. The data and analysis scripts are available on Github at https://github.com/reeselevine/mc-mutants-artifact. We will strive to keep the information on this repository up to date as the software used to run our code changes.

The source code for the hosted webage is also available on github: https://github.com/reeselevine/webgpu-litmus. Disk space required is minimal, on the order of a couple megabytes.

A.3.2 Hardware dependencies. To reproduce the exact results from the paper, the following devices are required: NVIDIA GeForce RTX 2080, AMD Radeon Pro, Intel Iris Plus Graphics, Apple M1. Devices from the same manufacturer/architecture generation will likely produce similar results to the ones in the paper.

Additionally, to reproduce the bug used in the correlation analysis from an older paper, an NVIDIA Kepler architecture GPU is required, such as an NVIDIA GeForce GTX 780.

A.3.3 Software dependencies. To collect the results using the hosted webpage, Google Chrome is required.

To run the website locally, Google Chrome Canary is required, as well as node.js and npm.

To run the analysis scripts, python3 is required, as well as the packages numpy, matplotlib, and pandas. These can be installed using pip.

A.3.4 Data sets. All data used for the results in the paper is included in the Github repository.

A.4 Installation

There are two parts to the installation/validation: data collection and results analysis.

Data Collection: Visit the hosted website or clone it and run it using the instructions on its Github page. If running locally and using Chrome Canary, WebGPU must be enabled by typing in "chrome://flags" in the address bar, searching for the flag "enable-unsafe-webgpu", and switching it to "Enabled".

To validate it works, visit https://gpuharbor.ucsc.edu/webgpu-mem-testing-artifact/tuning/ and press the "PTE" preset. This will take a while to run, so as a sanity check, reduce the number of configurations to 1. The test run should complete with no errors, and the results should be available for download when clicking the "All Runs: Statistics" button.

Results Analysis: Clone the repository and install the python packages needed using pip. To validate everything is working, run the `mk_figure5.py`, `mk_figure6.py`, and `mk_table4.py` scripts (i.e. `python3 mk_figure5.py`). If everything is set up correctly, the figures from the paper will be available under the `figures` directory, and the correlation numbers from Table 4 will be printed directly to the terminal.

A.5 Experiment workflow

All of the results used in the paper were collected using WebGPU, which runs from the browser. As mentioned, we host the website at https://gpuharbor.ucsc.edu/webgpu-mem-testing-artifact, which runs the code used to collect the data for the paper directly from the browser.

The tabs on the left side of the page contain links to many different litmus tests. Each include the ability to set parameters, run different configurations, and see results. To run the experiments included in the paper, go to the Tuning Suite tab. There, you will see four preset buttons, "SITE Baseline", "SITE", "PTE Baseline", and "PTE". These presets correspond to the four environments described in Section 5.1 of the paper. Don't worry about setting any other parameters; once you've clicked the preset you'd like to run, press the "Start Tuning" button. When the experiment is complete, which may take 2-4 hours per test, the results are available for download as a json file from the "All Runs: Statistics" button.

Similarly, the buttons under the "Correlation Tests" heading can be used to replicate the correlation study in the paper (assuming access to the GPUs used). Each will run the same configuration used in the paper, and the results analysis below can be used to check the correlations between the conformance tests and mutants.

A.6 Evaluation and expected results

All of the data collected and included in the paper are also included in this repository. Specifically, the folders `site_baseline`, `site`,

`pte_baseline`, and `pte` contain the results for each of the four devices. `correlation_analysis` contains the results of running the mutants and the kernel that observes a bug in the three devices described in Section 5.4 of the paper.

All the figures in the paper can be reproduced exactly using the data in the directories. If the reviewers have collected data on one/all of the devices used in the paper, replace the json file with the one downloaded after running the corresponding tests. The new results will be included in the generated graph instead of the ones we have collected.

To generate the graphs included in Figure 5 of the paper, run `python3 mk_figure5.py`. The resulting pdfs will be written to the figures/ directory. Similarily, `mk_figure6.py` creates Figure 6, and `mk_table4.py` prints out the data included in Table 4 of the paper.

The other file included in this repository, `analysis.py`, contains code for parsing the results of a tuning run. There are three different analyses that can be performed. To see the possible command line arguments, run `python3 analysis.py -h`.

Mutation Scores and Mutant Death Rates: Given a result file (e.g. `pte/amd.json`), running `python3 analysis.py --action mutation-score --stats_path pte/amd.json` will print out the number of mutants that were caught by the tuning run, as well as the average mutant death rate. These numbers are broken down by mutant category and combined across all categories, as shown in Figure 5 of the paper.

Merging Test Environments: Given a directory of result files (e.g. `pte`), running `python3 analysis.py --action merge --stats_path pte` will print out the number of tests in the PTE datasets that are reproducible across the four devices at a given reproducibility score target and time budget, as described in Section 4.2 of the paper. In fact, the function `merge_test_environments` in `analysis.py` implements Algorithm 1 of the paper, combining environments on a per test basis. To change the reproducibility score target and time budget, command line arguments `--rep` and `--budget` can be used. Therefore, `python3 analysis.py --action merge --stats_path pte --rep 99.999 --budget 4` will find the number of tests that can be reproduced with 99.999% confidence at a time budget of 4 seconds per test.

Correlation Analysis: Given a result file (e.g. `amd.json` in `correlation_analysis`), the command `python3 analysis.py --action correlation --stats_path correlation_analysis/amd.json` will print out a table showing the correlation between the number of weak behaviors (or bugs observed) per test in the dataset. For example, the result of the above command is a 4x4 table showing the correlation between an unmutated conformance test, Message Passing Barrier Variant, and its three mutants, as defined by Mutator 3 in Section 3.3 of the paper. Notice that while not all the tests are highly correlated, there is a 96.7% correlation between the mutant Message Passing Barrier Variant 2 and the conformance test, which is the number reported in Table 4 of the paper for this bug. While we only use correlation analysis in our paper to show the relation between observing bugs in conformance tests and weak behaviors in mutants, it can also be used to show general correlation between any pairs of tests in a dataset.

Received 2022-07-07; accepted 2022-09-22