Autonomous Data-Race-Free GPU Testing

Tuan Ta
School of Electrical and Computer Engineering
Cornell University
qtt2@cornell.edu

Xianwei Zhang
AMD Research
Advanced Micro Devices, Inc.
Xianwei.Zhang@amd.com

Anthony Gutierrez
AMD Research
Advanced Micro Devices, Inc.
Anthony.Gutierrez@amd.com

Bradford M. Beckmann
AMD Research
Advanced Micro Devices, Inc.
Brad.Beckmann@amd.com

Abstract—As the deep learning and high-performance computing markets continue to grow, hardware designers are increasingly optimizing future GPUs to run compute (a.k.a. GPGPU) workloads. A key area of optimization for these compute-oriented designs, which was not emphasized when GPUs exclusively executed graphics workloads, is inter-thread data sharing and synchronization. GPU cache coherence protocols now support these operations and are governed by a specified memory consistency model. In general, current GPU models are based on sequential consistency for data-race-free (SC for DRF), which mandates data written to memory must be globally visible only after certain synchronization points. GPU coherence protocols based on such relaxed memory models are particularly difficult to design and test due to the large number of memory accesses that may be reordered. This leaves GPU hardware designers struggling to validate the correctness of GPU cache coherence optimizations.

To address this issue, this paper introduces a novel, completely autonomous random testing methodology for complex GPU cache coherence protocols. Our framework continuously generates sequences of memory requests with minimal user intervention using a mix of load, store, and atomic operations. The tester dynamically and autonomously checks each response against an expected global view of memory and immediately detects any inconsistencies in a target coherence protocol, providing designers detailed feedback on the issue. We then demonstrate the methodology on the popular cycle-level gem5 simulator by replacing its GPU core model with our unique testing framework. The results show that the GPU tester can cover 94% and 100% of all reachable state transitions in L1 and L2 caches respectively of a representative GPU coherence protocol. This coverage is 6.25% and 25% higher than the one achieved by a wide selection of 26 applications. In addition, the tester runs more than 50 times faster than those applications, which enables efficient and fast protocol debugging.

Index Terms—Cache coherence, central processing unit (CPU), graphics processing unit (GPU), memory consistency, simulation, testing.

I. Introduction

GPUs are the de facto high-throughput, programmable accelerators targeting a wide breadth of compute applications. In several emerging GPU applications, threads often communicate with each other using fine-grained synchronization via shared virtual memory, stressing the GPU's cache coherency protocol [38]. Therefore, hardware designers are frequently relying on pre-silicon testing methods to effectively, and correctly, model modern GPU coherence protocols in order to reduce time-to-market.

Modern cache coherence protocols are particularly difficult to test due to their state space explosion and tight coupling with relaxed consistency models. Formal verification strives to completely validate a protocol specification or implementation, but it is slow and requires substantial manual effort [12]. The challenges in formal verification are so immense that improvements in the space are noteworthy as demonstrated by a slew of recent publications [17][27][39][40][41][42][45].

Meanwhile, cycle-level simulators strive to model modern cache controllers with the necessary fidelity to make accurate performance projections. Such simulators are designed to balance accuracy, simulation speed, and configurability. The gem5 simulator [5], along with its Ruby cache and network models [29], is one such simulation platform, and it is widely used by industry and academia because of its flexibility and fidelity. Recently, an industrial-quality GPU compute model was added to gem5 [15], providing designers all the necessary infrastructure to evaluate the rich GPU protocol design space. However, rapidly testing newly developed GPU coherence protocols is still a challenging problem.

Rapid protocol testing is important even when formal verification has validated a protocol's state transitions. For instance, a protocol's implementation may contain bugs due to a variety of reasons that are independent of the protocol's state transitions [9][11][16]. More pragmatically, formal verification is time consuming and requires expertise unknown by many designers. Thus, hardware designers may choose to simply tolerate implementation bugs and rely on data from an independent functional memory model. However, such separation gives designers little confidence the modelled protocol is correct and can accurately project a future system's performance.

Alternatively, designers can tightly integrate the functional and timing models [6] and use application-level testing with limited success. However, most well-optimized applications do not stress synchronization and protocol corner cases. Furthermore, our results show this application-based testing process is time consuming and ad hoc since millions of instructions need to be simulated in detail to complete an application. Alternatively, designers can write focused microbenchmarks that stress certain memory interactions. However, compiler optimizations and unanticipated timing delays can make this approach extremely difficult. The results of both approaches are inadequate, leaving lingering bugs that

978-1-7281-4045-2/19/$31.00 ©2019 IEEE

lessen a protocol designer's confidence in a protocol's implementation.

To avoid the inadequate coverage and long latency of application-based testing, while also evading significant formal verification effort, prior work in the CPU space has proposed autonomous randomized testing. Specifically, Wood *et al.* [43] proposed random test generation for stressing cache controllers using false sharing and leveraged the fact that strong CPU protocols were building blocks to implement the sequentially consistent (SC) memory model. By using the SC memory model, their approach always knows correct values for variables, requires no prior knowledge of the protocol's state transition details, stresses race conditions, and finds bugs quickly compared to application-based testing. A CPU-only random tester [5] was designed using a similar philosophy and has been proven to be highly effective [29]. However, GPU protocols operate under relaxed memory consistency models [21][25][34], and simple testing approaches, such as the approach proposed by Wood *et al.*, are not directly applicable to the more complicated semantics.

To combat the complexity of relaxed consistency, Adve and Hill proposed Data-race-free (DRF) semantics [1] to simplify their understanding. DRF models provide formal and precise definitions that clearly specify to programmers what is permissible, and what is not. DRF models also allow for many low-level hardware and software optimizations to take place underneath these well-defined semantics. As a result, DRF semantics have been very successful in the industry and have directly influenced many well-known memory models, such as those in C++ [8] and Java [28].

Building on the success of DRF semantics, this paper introduces a new testing methodology capable of autonomously stressing GPU protocols designed to support relaxed consistency. Dynamic "happens-before" relationships expressed by atomic operations in DRF model are captured in our random testing framework. The framework disallows data races in its generated sequence of memory accesses to shared variables to deterministically reason their values at any time.

Overall, the paper makes the following contributions:

- The paper introduces a novel DRF random testing methodology applicable to GPU protocols supporting relaxed memory model.
- Following the methodology, we provide a random, highly configurable GPU tester, which generates sequences of memory requests with respect to the specified memory consistency model.
- We then evaluate our GPU tester against the application-based testing approach. Our results show that the tester identifies bugs much quicker and covers more coherence state transitions than running a wide selection of GPU applications. The tester covers 93.8% and 100% of all reachable state transitions in L1 and L2 caches respectively of a representative GPU coherence protocol. The GPU tester achieves this coverage more than 50 times faster than it takes application-based testing to reach similar or lower coverage.
- Finally, we show that the GPU tester can complement an existing CPU tester to verify an integrated CPU-GPU coherence protocol in a heterogeneous system.

II. BACKGROUND

A. Sequential Consistency for Data-Race-Free

While SC offers programmers a simple model to reason about relative orders of concurrent memory operations, the model prevents hardware from reordering memory operations to achieve better performance. Therefore, many relaxed memory models have been proposed as solutions to improve hardware performance. The SC for DRF model 0 proposed by Adve and Hill [1] is one of them. Under SC for DRF, a program behaves as if it ran in an SC model if it has no data races. The behavior of data races under SC for DRF is undefined, so conflicting memory accesses like concurrent store operations to the same variable need to be ordered by some synchronization mechanisms such as memory fence.

B. Random CPU Coherence Testing Frameworks

There are several existing random testing frameworks for CPU coherence protocols. Wood *et al*. [43] proposed an automated testing framework that verified the functionality of cache controllers in multi-processor systems. Their tester is an N-state finite state machine issuing a sequence of loads and stores. The state machines replace complex CPU micro-architectural models, so they are much faster than simulating real applications with detailed CPU models. The tester uses pseudo-random numbers to generate memory addresses, so it can access arbitrary cache lines and trigger false sharing scenarios. The tester assumes a strong memory model, so it can rely on the issuing order of load and store operations to determine values of shared variables. This assumption simplifies the tester's implementation by allowing it to test CPU protocols supporting strong memory model without knowledge of the underlying protocols. Hangal *et al.* [17] built a tool called TSOTool to verify total-store-order (TSO) compliant CPU memory systems. The tool generates data-race programs that issue load and store operations to a target memory system. Values of those operations are observed and checked against a set of TSO-defined axioms to detect any violation of the TSO model.

C. Naively Applying CPU Coherence Testing to GPUs

Different from CPU protocols, GPU protocols operate under relaxed memory models. Such relaxed models may make the order of load and store operations from a GPU core appear differently with respect to other GPU cores unless there are explicit synchronization operations. A coherence protocol supporting a relaxed memory model may leverage this property to lazily perform memory operations until encountering an explicit synchronization operation enforcing them to appear globally to other cores. Since existing CPU testing frameworks

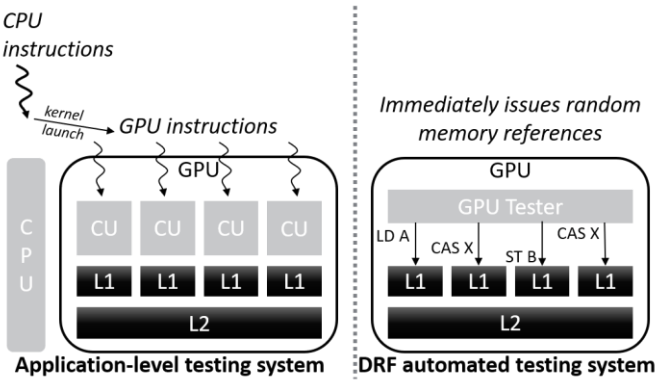


Fig. 1. Example testing systems in gem5 using applications and GPU tester.

including ones proposed by Wood *et al.* and Hangal *et al.* do not support such synchronization operations required by relaxed memory models, they are not able to use observed values of load and store operations to infer global states of shared variables. Therefore, such CPU testing frameworks are not directly applicable to GPU protocols.

D. The gem5 simulator

gem5 [5] is a highly configurable and modular simulation infrastructure for computer-system architecture research. It is widely used by academic and industrial researchers alike. gem5 includes several CPU models and supports all major ISAs, including ARM and x86. In addition to its classic memory model, gem5 includes the Ruby [29] memory system. Ruby is a set of cache and interconnect models that allow users to quickly specify new cache coherence protocols by using its SLICC domain-specific language. Recently, AMD released an open-source GPU compute model into the gem5 public repository [15] along with a representative GPU cache coherence protocol called *VIPER* [4]. VIPER is a write-through protocol with release consistency semantics and was the base protocol used by Power *et al.* [36] and Hechtman *et al.* [19].

III. DRF GPU Testing Framework

Our DRF GPU testing framework randomly generates memory operations and dynamically tracks the "happens-before" relationship between them to detect any coherence violation. The framework does not model microarchitectural details of GPU cores. Instead, it randomly generates and sends streams of memory accesses including loads, stores, atomics, and memory synchronization requests through memory ports as a proxy GPU device. Fig. 1 shows the main differences between running applications in a real GPU system and running the GPU tester without detailed a GPU core's model.

The remainder of this section describes the details of our testing approach. Specifically, Section III.A describes how streams of memory accesses are generated to stress a GPU memory system while adhering to the DRF memory model. Next, Section III.B explains how the tester reflects a generic GPU thread model to resemble GPU's memory traffic patterns. Then Section III.C describes how the tester maintains data integrity at any point in time. Finally, Section III.D presents how

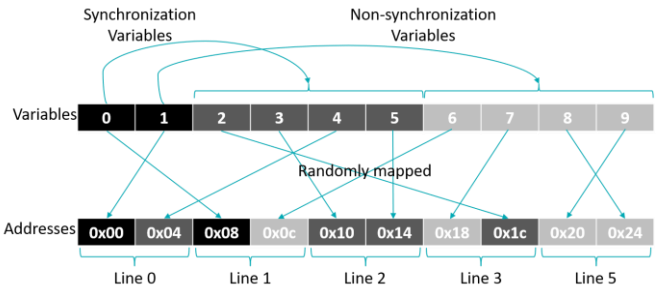


Fig. 2. An example of a random mapping between variables and real addresses.

the tester can generate a detailed log of memory transactions in case of failures to help ease the debugging process.

A. DRF Memory Access Generation

The GPU tester maintains a set of shared variables accessible by load, store, and synchronization operations. In DRF relaxed model there are two types of variables: atomic/synchronization and normal/non-synchronization variables. The tester generates load/store operations accessing non-synchronization variables and atomic operations accessing synchronization variables. This constraint obeys the DRF memory model.

The tester randomly maps variables to a range of memory addresses. This random mapping enables the tester to generate conflicting memory operations accessing different variables mapped to the same cache line. For example, in Fig. 2, synchronization '*variable 0*' and non-synchronization '*variable 4'* can co-locate in *'cache line 0'*. Since cache controllers operate at the granularity of cache lines, such random mapping can help trigger false sharing cases when load, store, and atomic operations race for the same cache line. False sharing accesses are major sources of bugs in a coherence protocol's specification and implementation.

DRF programs avoid data races through well-defined critical sections that are guarded by atomic operations. A critical section starts once a thread acquires a lock which is implemented as an atomic variable. It ends once the thread releases the lock. Concurrent threads trying to update a shared variable must successfully acquire a shared lock to avoid any data race. The tester replicates the behavior of a DRF program by generating and issuing streams of *episodes* to memory. Like a critical section, an episode is a well-defined sequence of memory operations guarded by atomic operations. For example, in Fig. 3, *episode 0* (Eps 0) is a sequence of four memory operations. It begins with an atomic operation with acquire semantics to a synchronization variable. The acquire semantics orders the atomic operation before any subsequent memory operations in the episode. Following the first atomic operation are load and store operations. An episode ends with another atomic operation with release semantics that orders this atomic operation after any previous memory operations in the episode.

The tester generates load and store operations in an episode so that they do not form any data race with any other active episodes in the system. For example, in Fig. 3, Eps 0 and 1 are both active at the same time, and they both try to update and

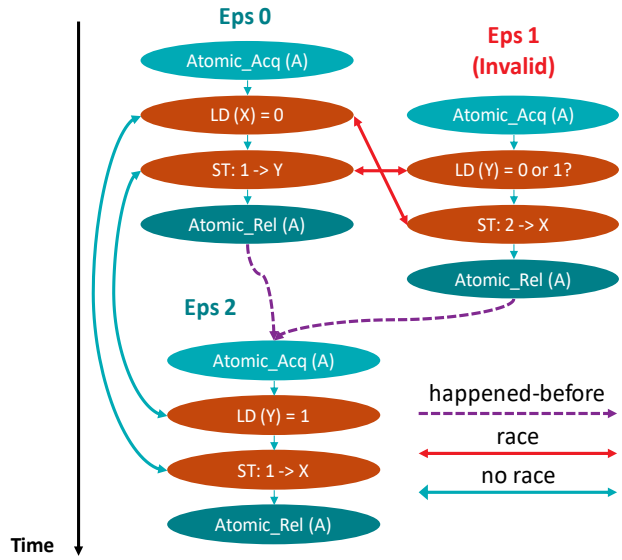


Fig. 3. Valid and invalid generated episodes. Episode 1 violates DRF constraints. Both episodes 0 and 1 try to read and update the same variables, which creates data races. This type of episode is not generated by the tester. Synchronization between episode 0.

access the same variables, which forms a data race between them. In order to prevent two active episodes from having racy memory accesses, when the tester initializes an episode, it selects a synchronization variable to acquire. Based on the selected synchronization variable, the tester determines active episodes that acquire the same synchronization variable and therefore potentially form a data race with the new episode. The tester then generates all load and store operations for the new episode so that no possible data race can be formed with any active episode by applying the following two rules.

- No load or store operation is generated for a shared variable being stored by an active episode.
- No store operation is generated for a shared variable being loaded by an active episode.

B. GPU Thread Model in the Tester

Although the tester does not model microarchitectural details of GPU cores, it does resemble how real GPUs core would send many accesses to the memory system in parallel. The tester consists of multiple threads that are connected to a memory system. Each tester thread acts like a real GPU thread and issues its sequence of memory accesses to memory. Multiple threads grouped together in a wavefront (WF) operate in lockstep. Threads in a WF can proceed to their next memory operations only when all other threads in the WF finish their current operations. This lockstep execution is similar to how GPU threads operate in a single-instruction, multiple-thread (SIMT) execution model. However, unlike an actual GPU model, the tester takes advantage of the fact that within the simulator it can directly attach to the cache hierarchy, bypassing the GPU core model and reducing simulation time.

The tester can also be extended to evaluate any system configuration; therefore, the user can configure a multi-GPU system with a varying number of caches and diverse topologies. As long as the system under test has a DRF memory model, the tester will work seamlessly.

TABLE I. GPU L1 CACHE EVENTS

Event	Description
Load	Data read request from GPU
StoreThrough	Data write request from GPU
Atomic	Data atomic request from GPU
TCC_Ack	Data response from GPU L2
TCC_AckWB	Write completion ack from GPU L2
Evict	Flash invalidation request from GPU
Repl	Cache replacement request

TABLE II. GPU L2 CACHE EVENTS

Event	Description
RdBlk	Data read request from GPU L1
WrVicBlk	Data write request from GPU L1
Atomic	Data atomic request from GPU L1
AtomicD	Atomic completion ACK
AtomicND	Atomic incompletion ACK
Data	Data response from memory
L2_Repl	Cache replacement
PrbInv	Invalidation request from other L2
WBAck	Write completion ACK from memory

C. Data and Forward Progress Checks

To provide useful information to a hardware designer, the tester must know whether loaded values are consistent with expected values, and that the overall system is making forward progress (i.e., there are no deadlocks). Our testing infrastructure can do this in a fully autonomous manner, without receiving any guidance from the user, because it maintains a reference memory of all variables and their values. With respect to release consistency, the tester understands that a newly written value becomes globally visible to other threads after the episode retires (i.e., when it finishes executing its last atomic operation with release semantics) and updates its internal storage appropriately. Using its up-to-date storage of all variables, the tester can immediately identify inconsistent loaded values and generates an error message outlining details of the bug.

The tester is able to detect if a memory system is not making any forward progress for an unreasonably long period of time, which may be due to a deadlock in the protocol's implementation. The tester records a timestamp for each outstanding request when it is issued to the memory system. Periodically, the tester runs a forward progress check and detects whether any outstanding requests have not received a response for more than a specified number of cycles. If so, the tester raises a flag to signal a potential deadlock happening in the memory system. This threshold is chosen heuristically to be one million CPU cycles in our evaluation but is also configurable. This check complements the default deadlock detection mechanism implemented within Ruby and finds deadlock issues within the GPU memory system interface.

TABLE III. TESTER CONFIGURATIONS

Test Type	GPU Tester	Application	CPU Tester
Protocol	GPU_VIPER	GPU_VIPER	MOESI_AMD_Base
System size	8 CUs	8 CUs	2/4/8 CPUs
Cache size	Small: 256B L1, 1KB L2 Large: 256KB L1, 1MB L2 Mixed: 256B L1, 1MB L2	16KB L1, 256KB L2	Small: 256/512B Corepair Large: 256/512KB Corepair
Test length	Permutation of [100, 200] actions/episode and [10, 100] episodes/wavefront	Simulation time varies	100/10K/100K/1M loads
Address range	Small: 10 atomic locations, 1M regular ones Large: 100 atomic locations, 1M regular ones	16 GB	512 MB
Test runs	Test 0, Test 1, …, Test 23	HACC, Square…, FFT	Test 0, Test 1, …, Test 23

D. Event Logging

A key requirement for any testing infrastructure is providing detailed information when problems are detected. Our tester prints out all memory transactions related to a specific failed operation by automatically creating detailed event logs during runtime. For example, when a loaded value is not consistent with the tester's internal state, the tester prints out information about the last transaction that updated the variable. The information includes which thread, thread group, and episode issued the transaction, when the transaction happened, and what value was written to the variable. Such information is helpful in aiding a memory system designer when attempting to narrow down a specific window of activities causing the problem.

IV. EVALUATION

We use gem5 simulator [15] and a representative GPU coherence protocol called VIPER [4] to evaluate efficiency of the GPU tester. We compare its performance and coherence state coverage against the application-based testing approach that runs a set of diverse GPU applications. We then use an existing CPU protocol tester in gem5 to show how our GPU tester can complement the CPU tester in testing a coherent CPU-GPU memory system.

VIPER is a write-through protocol performing store operations immediately using per-byte dirty masks and does not stall for exclusive permissions. The protocol supports two synchronization operations: *load acquire* and *store release*. When an L1 cache receives a load acquire, it invalidates all cache entries to remove any possibly stale data. A store release operation waits for all write-through operations to complete so that the most up-to-date data are available in memory. VIPER represents common characteristics of coherence protocols used in AMD and NVIDIA GPUs including read-initiated invalidations and write-through L1 caches.

The GPU's memory system has per compute unit private L1 data caches and an L2 cache shared among GPU cores. The CPU includes two cores with private L1 data caches and a shared L2 cache. The system directory is being shared by the CPU and GPU. Fig. 4 shows all coherence state transitions. All possible cache events in the GPU's L1 and L2 caches are shown in TABLE I and TABLE II respectively. It is important to note that the tester is not specific to the VIPER protocol. The tester can support other GPU protocols as well with minimal extensions.

	A	I	V
Atomic	Undef	Stall	Stall
Evict	Stall	-> I	-> I
TCC_Ack	-> I	-> V	Undef
TCC_AckWB	-> A	-> I	-> V
Repl	-> A	Undef	-> I
StoreThrough	Undef	-> I	-> I
Load	Undef	-> V	-> I

GPU L1

	A	I	IV	V
RdBlk	Stall	-> IV	Stall	-> V
AtomD	-> I	Undef	Undef	Undef
Data	-> A	Undef	-> V	Undef
Atomic	-> A	-> A	Stall	-> A
WrVicBlk	Stall	-> I	Stall	-> V
WbAck	-> A	-> I	-> IV	-> V
PrbInv	-> A	-> I	-> IV	-> V
AtomND	-> A	Undef	Undef	Undef
L2_Repl	-> A	-> I	Stall	-> I

GPU L2

Fig. 4. State transitions in GPU L1 and L2 cache. "Undef": the transition is undefined. "Stall": the transition stalls the cache controller. "-> X": transition to state X. State "V" means the cache line is in the valid state. State "I" means the cache line is invalid. State "IV" means the cache line is waiting for refill data. State "A" means the cache line is doing an atomic access and waiting for an Ack to complete the access.

For example, some protocols may support extra cache invalidation or cache flush operations at synchronization points, which requires adding such semantics to memory fence operations and possibly changing how data are validated with respect to the target memory model in the tester.

We sweep over different configurations as summarized in TABLE III. The GPU and CPU tester runs are permutations of various parameters (e.g., cache size, address range, and test length). Application-based tests are composed of various applications from compute [2][3], HeteroSync [37] and MI applications, including DNNMark [12], DeepBench [32], and MIOpen-benchmarks [14]. Descriptions of the applications are provided in TABLE IV.

For fair comparison, applications should be diverse enough to stress various cache transitions. We use a data locality characterization approach adopted by Koo *et al.* [24] to characterize data locality of each application. We profile the cache line usages between WFs and classify them into streaming (never reused), intra-WF (reused within the same WF), inter-WF (reused by different WFs) and mixed-WF (mixed reuse). Fig. 6 reports the breakdown of the four categories, and the results show that our selected applications demonstrate vastly different behaviors and thus access cache differently. In addition, our further profiling studies show that certain MI and HeteroSync applications are heavily stressing atomic operations. Transition coverage (i.e., fraction of activated transitions out of all

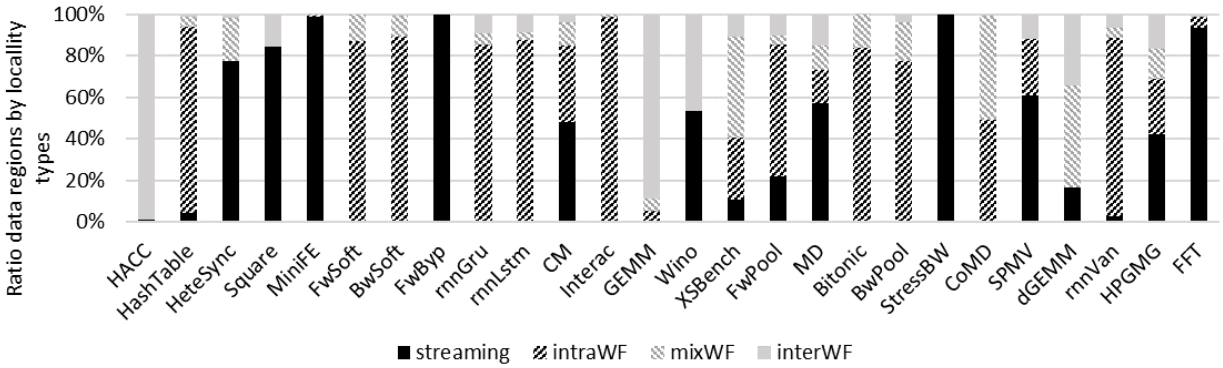
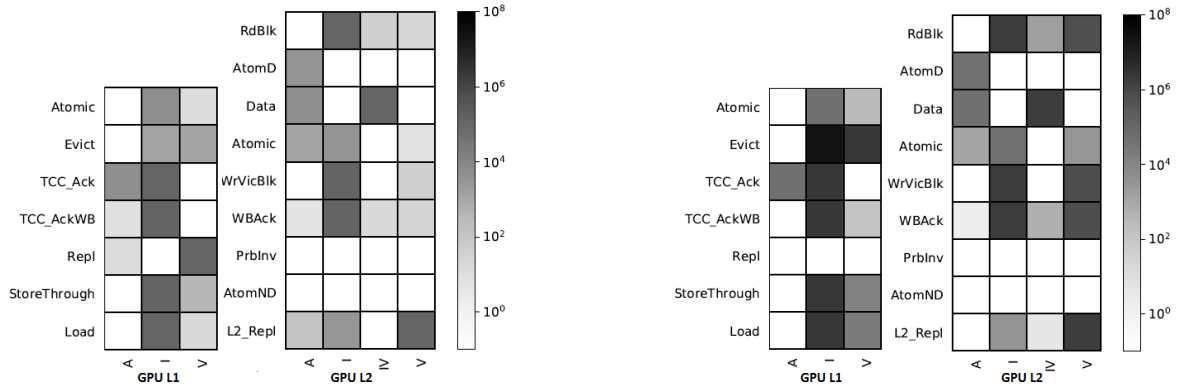


Fig. 6. Data locality in selected applications.



(a). Small Cache Size (b). Large Cache Size

Fig. 5. GPU L1 and L2 transition hit frequency with different cache sizes. Small cache size: 256B 2-way L1, and 1KB 2-way L2. Large cache size: 256KB 16-way L1, and 1MB 16-way L2.

defined/reachable transitions) and execution time (i.e., time to finish a simulation run) are collected to evaluate the GPU tester and compare against real applications and the CPU tester.

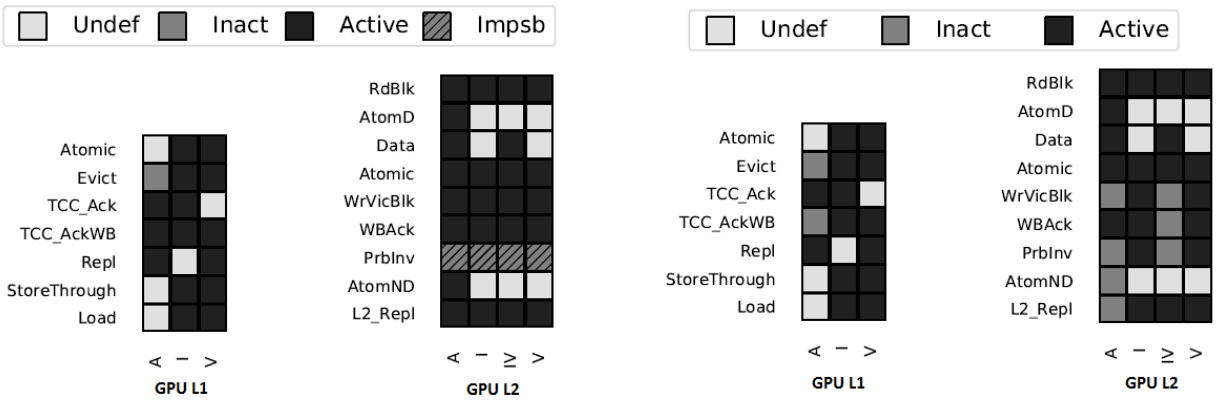
In this section, we first show how the high configurability of GPU tester helps explore different parts of the tester configuration space. Second, we compare GPU tester and applications in terms of their cache transition coverage and testing time. Third, we describe how well the GPU tester complements CPU tester to increase the overall state transition coverage in the coherence directory in a shared CPU-GPU system.

A. Exploring the Tester Configuration Space

Due to the diversity of memory systems, and to support testing future designs, the GPU tester must be highly configurable so that users can easily test a wide range of designs. In addition, configurability allows the tester to achieve high coverage in a short amount of time. Cache size, memory address range, cache line size, synchronization vs non-synchronization variable ratio, the episode length, etc. are all configurable in the tester. Users can choose to vary values of some parameters to focus on certain corner-case state transitions. In this section, we show how changing cache size parameter can stress different subsets of possible state transitions in GPU L1 and L2 caches.

Fig. 5 shows heat maps of transition hit frequency in two different testing configurations: small and large GPU caches (i.e., other parameters including test length and episode length are identical). The shade of each transition in the maps corresponds to how frequently it is hit by the tester. The large cache testing case (Fig. 5.(b)) triggers cache hit transitions (e.g., [Load, V] in GPU L1, [StoreThrough, V] in L1 and [RdBlk, V] in GPU L2) more frequently than the small cache testing case (Fig. 5.(a)) does due to the difference in cache capacity. However, the small cache testing case is more effective in triggering cache replacement-related transitions (e.g, [V, Repl], [A, Repl] in L1, [V, L2_Rep] in L2) due to its relatively small cache capacity.

In addition to cache size, other parameters including address range and episode length can be configured to target different subsets of an entire state transition space. For example, smaller address range increases the number of sharing accesses between threads, which stresses transient states. Increasing the length of episodes induces more non-synchronization accesses per synchronization access, which is likely to increase inter-episode interactions. By combining various configuration sets that stress different parts of a protocol, the tester can hit individual transitions in much shorter period than memory accesses generated by actual GPU applications.



(a). GPU Tester (b). All Applications

Fig. 7. Comparison of the GPU L1 and L2 transitions covered by GPU tester and applications

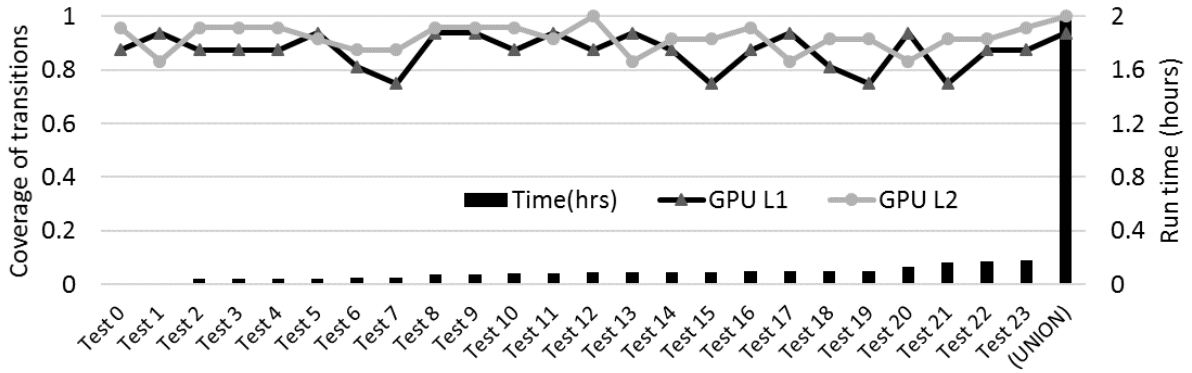


Fig. 8. Transition coverage of GPU tester on the GPU L1 and L2 caches.

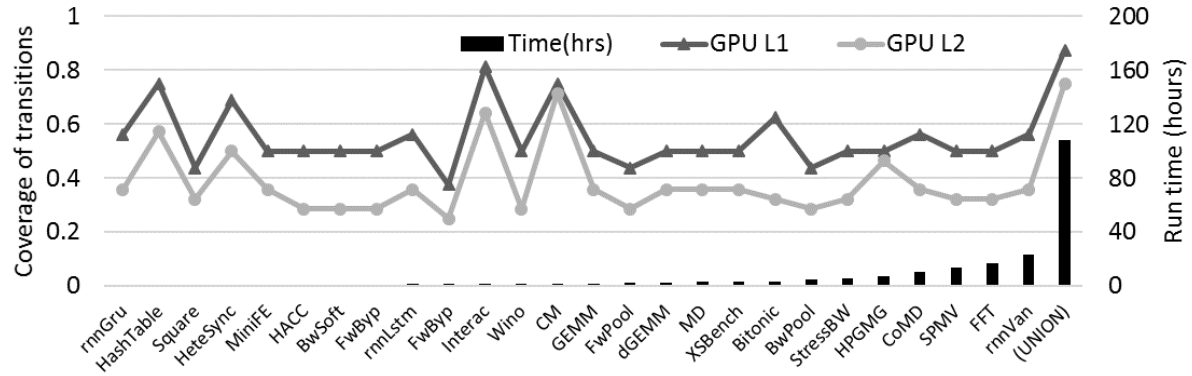
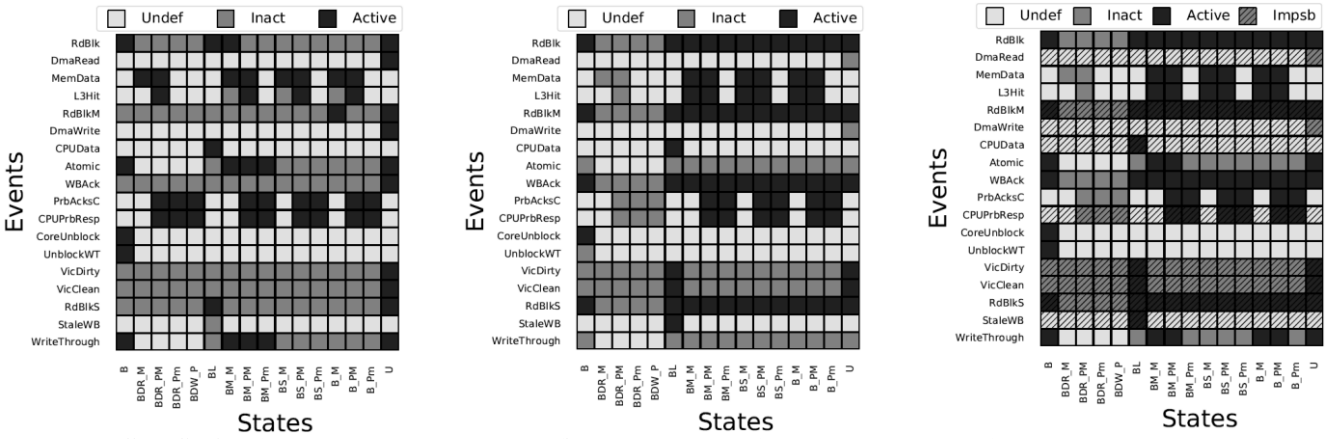


Fig. 9. Transition coverage of applications on the GPU L1 and L2 caches.

B. GPU Tester versus Applications

Applications or unit tests are typically used to perform regression testing. However, as we will show, application-based testing is a slow process, and tailoring an application suite to provide the right amount of coverage is inefficient. To quantify testing efficiency, we classify transitions (e.g., [*A*, *E*, *B*] where state *A* transitions into state *B* via event *E*), in the Ruby system into four classes as illustrated in Fig. 7:

- *Undefined (Undef)*: no defined transitions from *A* via *E*. If event *E* occurs while a block is in state *A*, this triggers a fault signaling that the protocol implementation is faulty.
- *Inactive (Inact)*: there are possible transitions from *A* via *E*, but the transitions are never observed during testing.
- *Active (Active)*: there are defined transitions from *A* via *E*, and the transitions are activated during testing.



(a). All Applications (35.2%) (b). CPU Tester (49.7%) (c). GPU+CPU Tester (56.6%)

Fig. 10. System Directory transitions covered by different test types.

- *Unreachable by GPU tester (Impsb):* transitions from *A* via *E* are impossible to be reached by GPU tester because of the CPU-related constraints. For instance, in L2, *PrbInv*-triggered transitions are not reachable by the GPU tester since there is only one single L2 cache in the testing system. *PrbInv* events are only triggered by a remote L2 cache (e.g., CPU L2 cache) to invalidate a target cache line. In an application-based testing system, those transitions can be reached since the CPU L2 cache can send an invalidation request to invalidate a cache line in the GPU L2 cache.

Following the classifications from the GPU L1 and L2 comparison in Fig. 7, we see that GPU tester and applications are showing the same undefined squares (i.e., transitions), but the GPU tester shows more active ones, indicating that more transitions are hit and covered by the GPU tester than when using applications. Fig. 8 and Fig. 9 further compare the transition coverages and testing time of the GPU tester and applications respectively. Results are reported in the order of run time, and the final "(UNION)" shows the union of the coverage of all tests and the total cumulative run time. Our GPU tester is able to reach 94% coverage on the GPU L1 and 100% coverage of all reachable transitions on the GPU L2, whereas application-based tests reach 6.25% and 25% lower coverage on the L1 and L2 respectively. The inactive transitions are mainly contributed by corner cases (e.g., store hits on a pending atomic operation) and CPU or DMA related requests (e.g., the GPU's L2 probe-invalidate transitions). Further, the GPU tester runs more than 50x faster to reach similar or higher coverage since the tester does not model any detailed GPU pipeline. This much higher testing speed translates directly to the debugging efficiency since bugs can be reproduced much faster than they are in the application-based testing approach.

Fig. 9 further shows that the union coverage of the applications is dominated by the *Interac* and *CM* applications that use more atomic operations and thus activate more transitions than the remaining applications. This behavior of application-based testing shows that not all applications can be effectively used to test most state transitions. Only few selective applications can reach a large space of all possible state transitions. Unfortunately, choosing such effective applications is challenging without a deep understanding of their memory access patterns.

C. Complementing the CPU tester

To provide complete memory testing one would also need to test the CPU-side caches. For an APU, the system directory is typically shared by the CPU and the GPU, each of which is issuing different request types to the directory and thus can activate some unique transitions. Therefore, to fully test the system the CPU must also issue requests to the directory. Fig. 10.(a)-(b) illustrate the directory coverage of applications and the CPU Tester.

Given that GPU and CPU testers are stressing different transitions in the directory, we run each separately and take the union of their coverage, with results being reported in Fig. 10.(c). As expected, the combination is capable of activating transitions that can only be uniquely triggered by either the CPU or GPU tester, and the combined coverage is 21.4% higher than the applications (56.6% versus 35.2%). However, by comparing Fig. 10.(a) to Fig. 10.(c), we see that the applications do activate some transitions that are not triggered by GPU and CPU testers, and such transitions are DMA related, which is not modeled in either tester. In addition, with respect to run time, the GPU and CPU testers are still significantly shorter (~12.6x) than the applications.

V. Case Study

In this section, we show a case study on how the tester discovered two common bugs in the GPU VIPER protocol and helped protocol designers debug them.

One of the most common protocol bugs occurs when a loaded value of a shared variable is not consistent with the last value written. This indicates the hardware model is incorrect, assuming a DRF program. When data inconsistency occurs, the bug must have happened in between the current load operation and the last write operation on the variable. When detecting the inconsistency, the tester will report detailed information about the problem.

TABLE V. AN EXAMPLE OF A READ-WRITE INCONSISTENCY BUG

	Last Reader	Last Writer
Thread ID	35	12
Thread group ID	4	2
Episode ID	727	652
Address	0x52860	
Cycle	16,905	16,098
Read/Written Value	16	17

TABLE V shows example output from the tester for the value inconsistency bug. Both the last reader and last writer tried to access the same variable at address *0x52860*. The write operation happened (i.e., at cycle *16,098* by thread 12 in group 2) before the read operation (i.e., at cycle *16,905* by thread 35 in group 4). Since the tester ensures that read and write operations on the same variables by different threads are ordered correctly by atomic operations, if the tested memory system were bug free then inconsistency should not have happened. In addition, the bug must have occurred in the period between two operations.

Given the debugging information, a protocol writer can zoom in to view that specific window, print out relevant memory transactions initiated by the two threads, examine states of all relevant cache controllers, and diagnose the issue. In this example, the issue was that two false sharing write operations on different memory addresses of the same cache line (i.e., false sharing) race at the GPU L2 cache controller in the GPU VIPER protocol. The two operations were not serialized correctly at the controller, which made the write operation on address *0x52860* fail to update the cache line. False sharing between memory accesses is a frequent cause of protocol implementation failures. The tester can be configured so that false sharing happens more often, which helps expose hidden bugs much faster than simply running real applications, which are often designed to avoid false sharing (e.g., by padding data structures to align to cache block boundaries).

Another common protocol bug is multiple atomic operations not atomically updating shared synchronization variables correctly. The tester issues atomic operations that atomically adds a constant positive value to shared variables so that their values are always monotonically increasing. An atomic operation returns a variable's old value before the operation is performed on the variable. Since atomic operations happen atomically and always increase values of variables, their return values are unique. If atomic operations performing on a variable return duplicate values, the tester detects and reports the violating operations to the user. Using the report, a protocol writer can zoom into the small window of execution between the two operations and examine their memory transactions to find out where and when the bug happens. This way of testing atomic operations is significantly more efficient than using real applications since incorrect synchronization in real applications can lead to extremely complicated failure conditions.

VI. RELATED WORK

A. Improving Full-System Simulation Performance

Several prior works have recognized the need to improve the performance of simulation when evaluating complex full-system workloads, such as server workloads. These workloads rely heavily on library calls, OS functionality, and devices (e.g., NICs). Hardavellas *et al.* developed *SimFlex* [18], a full-system simulator designed with performance in mind. SimFlex relies on *Simics* [26], a functional full-system simulator, for the execution of workloads and the OS. In addition to relying on a functional full-system simulator, SimFlex utilizes a compile-time approach for component interconnection, which allowed compile-time optimizations across component boundaries. The most novel feature of SimFlex is its use of the *SMARTS* [44] statistical sampling methodology and extends it to multi-core systems. By using statistical sampling, simulation complexity and runtime can be reduced because many events need not be modelled, while simulation accuracy remains high.

Mauer *et al.* [31] proposed decoupling the functional and timing aspects of full-system simulation and introduce *TFsim*, a timing-first simulator. They recognize that designers typically rely on simulators that provide both functional and timing accuracy. However, such models become difficult to implement accurately and with acceptable performance as computer system increase in complexity. TFsim also relies on Simics [26] for functional simulation. TFsim first executes dynamic instructions in their timing-first model and only invokes the functional simulator at commit time. If the functional simulator detects an error, the timing simulator's state is updated to reflect the correct state, which is obtained via the functional simulator.

Narayanasamy *et al.* [33] recognized the need to emulate the OS effects when running architectural simulations. However, they also understand that correctly modelling the OS leads to increased runtime and complexity of simulation. Their work proposes *Pin-System Effect Logger (pinSEL)*, a Pin-based [23] tool for logging OS effects. pinSEL instruments binaries and runs them natively on the OS for which it was compiled. As the instrumented binary executes, pinSEL creates a log of OS effects. The log contains changes to register state, values of memory locations by loads after the system call, whether those locations were modified by the OS, interrupts, and DMA actions. With pinSEL only application instructions need to be modelled, and OS effects are emulated in a simulator using the logged events.

B. Designing Coherence Protocols for Verification

Traditional formal verification methods are complex and very slow. Many algorithms used in such methods are NP-complete, and verification methods used in industrial settings often require massive amounts of compute or expensive emulation tools. In order to alleviate some of the complexity of verifying cache coherence protocols, several prior works investigate the design of formally verifiable cache coherence protocols. Zhang *et al.* proposed fractal coherence [45], which are cache coherence protocols that are designed to have fractal

properties. With such properties the protocols can be easily verified to scale up correctly to an arbitrary number of cores. Voskuilen and Vijaykumar extend fractal coherence [40][41] in order to improve its performance. While these protocols are suitable for verification, they do suffer from performance loss and may not be suitable for complex heterogeneous systems.

C. Automatic Protocol Generation from Specifications

More recent work by Oswald *et al*. [35], and Matthews and Sorin [30], have proposed generating verifiable cache coherence protocols from theoretical principles. These works look to bridge the gap between formal verification and the design of protocols. These works remain focused on CPU protocols. In [30] atomic specifications are used to automatically generate directory protocols, and in [35] *ProtoGen* is proposed. ProtoGen is capable of generating complete MSI, MESI, and MOSI protocols given only their stable state definitions.

D. Random Testing for CPU Coherence Protocols

As previously mentioned, Wood *et al*. proposed a random testing strategy for multi-core CPU systems [43], and Martin *et al*. utilize this design to develop a CPU tester for the Ruby memory models in gem5 [29]. While this testing strategy works well for CPUs, as discussed in section 2, it is not easily extended to more relaxed GPU consistency models, or for Heterogeneous systems. We use the basic philosophy of this prior work and build upon it to develop our random testing framework for relaxed GPU consistency and coherence models.

In addition, Hangal *et al.* [17] proposed TSOTool that randomly generates programs with data races to verify memory systems implementing TSO memory model. Since the tool is focused solely on TSO memory, it is also not applicable to test GPU memory systems that adopt much more relaxed memory models.

While prior work has shown that formal verification, along with protocols designed for formal verification [40][41][45], can be effective for easing the burden of proving formal correctness for designers, and have demonstrated how theoretical frameworks can be used to generate correct protocols implementations [30][35], these approaches are not necessarily applicable for researchers who wish to quickly and accurately test state-of-the-art relaxed memory models in heterogeneous environments. For these researchers, some abstraction is tolerable, however correctness must be ensured at low cost. Therefore, our random GPU testing infrastructure provides a sweet spot for coherence and consistency verification when quickly evaluating new ideas.

VII. Conclusions

As computer systems become more heterogeneous and complex, it is important that the simulation platforms on which researchers do their work be accurate and bug free. Because formally verifying consistency models and coherence protocols is challenging, and because bugs can still creep into the implementation of formally verified protocols, it is important that researchers can quickly test their protocol implementations for correctness.

To address this challenge, we have developed a random testing framework for modern, relaxed GPU protocols in the state-of-the-art gem5 simulation platform. To implement the framework, we leverage DRF's simplified semantics rather than the more complicated semantics of relaxed consistency. Our results show that we can achieve 100% coverage of the GPU L2 cache with the GPU tester alone, nearly 25% higher coverage than tests based on applications, at approximately 54x shorter of the runtime. In addition, the system directory coverage, when combined with the CPU random tester is 56.6%, which is 21% higher than applications alone. Even when the GPU and CPU testers are run in serial, we can achieve this greater coverage, while running one or more orders of magnitude faster than running applications.

VIII. Acknowledgements

The authors would like to thank the anonymous reviewers for their feedback. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] S. Adve and M.D. Hill. "Weak Ordering–A New Definition". In the proceedings of the *17th International Symposium on Computer Architecture (ISCA)*, pp. 2–14, 1990.

[2] AMD. Compute Applications. http://github.com/AMDComputeLibraries/ComputeApps, 2017. Accessed: September 21, 2018.

[3] AMD. HCC Sample Applications. http://github.com/RadeonOpenCompute/HCC-Example-Application, 2017. Accessed: September 21, 2018.

[4] B.M. Beckmann and A. Gutierrez. "The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5". Tutorial at *the 48th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2015. http://gem5.org/wiki/images/f/fd/AMD_gem5_APU_simulator_micro_2015_final.pptx. Accessed: September 21, 2018.

[5] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. "The gem5 Simulator". In *SIGARCH Computer Architecture News*, 39(2), pp. 1–7, 2011.

[6] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. " The M5 Simulator: Modeling Networked Systems". In *IEEE Micro*, 26(4), pp. 52–60, 2006.

[7] B. Black and J.P. Shen. "Calibration of Microprocessor Models". In *IEEE Computer*, 31(5), pp. 59–65, 1998.

[8] H.-J. Boehm and S.V. Adve. "Foundations of the C++ Concurrency Memory Model". In the proceedings of the *29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLD)*, pp. 68 –78, 2008.

[9] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. "Accuracy Evaluation of gem5 Simulator System". In the proceedings of the *7th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, 2012.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S. Lee, and K. Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In the proceedings of the *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[11] R. Desikan, D. Burger, and S.W. Keckler. "Measuring Experimental Error in Microprocessor Simulation". In the proceedings of the *28th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pp. 266–277, 2001.

[12] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. "Protocol Verification as a Hardware Design Aid". In the proceedings of the *1992 IEEE International Conference on Computer Design: VSLI in Computers & Processors (ICCD)*, pp. 522–525, 1992.

[13] S. Dong and D. Kaeli. "DNNMark: A Deep Neural Network Benchmark Suite for GPUs". In the proceedings of the *10th Workshop on General Purpose GPUs* (*GPGPU)*, pp. 63–72, 2017.

[14] P. Flick. MIOpen Benchmarks. https://github.com/patflick/miopen-benchmark, 2017. Accessed: September 21, 2018.

[15] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, J. Kalamatianos, O. Kayiran, M. LeBeane, M. Poremba, B. Potter, S. Puthoor, M.D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T.G. Rogers. "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level". In the proceedings of the *24th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 608–619, 2018.

[16] A. Gutierrez, J. Pusdesris, R.G. Dreslinski, T. Mudge, C. Sudanthi, C.D. Emmons, M. Hayenga, and N. Paver. "Source of Error in Full-System Simulation". In the proceedings of the *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 13–22, 2014.

[17] S. Hangal, D. Vahia, C. Manovit, J.-Y.J. Lu. "TSOTool: A Program for Verifying Memory Systems Using the Memory Consistency Model". In the proceedings of the *31st Annual International Symposium on Computer Architecture*, pp. 114–123, 2014.

[18] N. Hardavellas, S. Somogyi, T.F. Wenisch, E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J.C. Hoe, and A.G. Nowatzyk. "SimFlex: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture. In *SIGMETRICS Performance Evaluation*, 31(4), pp. 31–35, 2004.

[19] B.A. Hechtman, S. Che, D.R. Hower, Y. Tian, B.M. Beckmann, M.D. Hill, S.K. Reinhardt, and D.A. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs". In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (pp. 189-200), 2014.

[20] D.R. Hower, B.A. Hechtman, B.M. Beckmann, B.R. Gaster, M.D. Hill, S.K. Reinhardt, and D.A. Wood. "Heterogeneous-race-free Memory Models". In the proceedings of the *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 427–440, 2014.

[21] HSA Foundation™. "HSA Platform System Architecture Specification 1.2". http://www.hsafoundation.com/standards. Accessed: February 1, 2019.

[22] HSA Foundation™. "HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG) Revision: Version 1.2". 2018.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In the proceedings of the *2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 190–200, 2005.

[24] G. Koo, Y. Oh, W. Ro and M. Annavaram. "Access Pattern-Aware Cache Management for Improving Data Utilization in GPU". In the proceedings of the *44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pp. 307–319, 2017.

[25] D. Lustig, S. Sahasrabuddhe, and O. Giroux. "A Formal Analysis of the NVIDIA PTX Memory Consistency Model". In the proceedings of the *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 257–270, 2019.

[26] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. "Simics: A Full System Simulation Platform". In *IEEE Computer*, 35(2), pp. 50–58, 2002.

[27] Y.A. Manerker, D. Lustig, M. Martonosi, and A. Gupta. "PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications". In the proceedings of the *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 788–801, 2018.

[28] J. Manson, W. Pugh, and S.V. Adve. "The Java Memory Model". In the proceedings of the *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 378–391, 2005.

[29] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M. R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood. "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset". In *SIGARCH Computer Architecture News*, 33(4), pp. 92–99, 2005.

[30] O. Matthews and D.J. Sorin. "Architecting Hierarchical Coherence Protocols for Push-button Parametric Verification". In the proceedings of the *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 477–489, 2017.

[31] C.J. Mauer, M.D. Hill, and D.A. Wood. "Full-System Timing-First Simulation". In the proceedings of the *2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 108–116, 2002.

[32] S. Narang. "DeepBench". https://svail.github.io/DeepBench/, 2016. Accessed: September 21, 2018.

[33] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. "Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation". In the proceedings of the *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance)*, pp. 216–227, 2006.

[34] NVIDIA. "CUDA Toolkit Documentation: Memory Consistency Model". https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#memory-consistency-model. Accessed: February 1, 2019.

[35] N. Oswald, V. Nagarajan, an D.J. Sorin. "ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications". In the proceedings of the *45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pp. 247–260, 2018.

[36] J. Power, A. Basu, J. Gu, S. Puthoor, B.M. Beckmann, M.D. Hill, S.K. Reinhardt, and D.A. Wood. "Heterogeneous System Coherence for Integrated CPU-GPU Systems". In the proceedings of the *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 457–467, 2013.

[37] M. Sinclair, J. Alsop and S. Adve. "HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs". In the *IEEE International Symposium on Workload Characterization (IISWC)*, October 2017.

[38] D.J. Sorin, M.D. Hill, and D.A. Wood. "A Primer on Memory Consistency and Cache Coherence". In *Synthesis Lectures on Computer Architecture*, 2011.

[39] C. Trippel, Y.A. Manerker, D. Lustig, M. Pellauer, and M. Martonosi. "TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA". In the proceedings of the *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 119–133, 2017.

[40] G. Voskuilen and T.N. Vijaykumar. "Fractal++: Closing the Performance Gap Between Fractal and Conventional Coherence". In the proceedings of the *41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pp. 409–420, 2014.

[41] G. Voskuilen and T.N. Vijaykumar. "High-Performance Fractal Coherence". In the proceedings of the *19th International Conference on*

Architectural Support for Programming languages and Operating Systems (ASPLOS), pp. 701–714, 2014.

[42] J. Wickerson, M. Batty, B.M. Beckmann, and A.F. Donaldson. "Remote-Scope Promotion: Clarified, Rectified, and Verified". In the proceedings of the *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 731–747, 2015.

[43] D.A. Wood, G.A. Gibson, and R.H. Katz. "Verifying a Multiprocessor Cache Controller Using Random Test Generation". In *IEEE Design & Test of Computers*, 7(4), pp. 13–25, 1990.

[44] R. E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling". In the proceedings of the *30th Annual International Symposium on Computer Architecture (ISCA)*, pp. 84–95, 2003.

[45] M. Zhang, A. R. Lebeck, and D.J. Sorin. "Fractal Coherence: Scalably Verifiable Cache Coherence". In the proceedings of the *43rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pp. 471–482, 2010