Jaaru: Efficiently Model Checking Persistent Memory Programs

Hamed Gorjiara
University of California, Irvine
USA
hgorjiar@uci.edu

Guoqing Harry Xu
University of California, Los Angeles
USA
harryxu@cs.ucla.edu

Brian Demsky
University of California, Irvine
USA
bdemsky@uci.edu

ABSTRACT
Persistent memory (PM) technologies combine near DRAM performance with persistency and open the possibility of using one copy of a data structure as both a working copy and a persistent store of the data. Ensuring that these persistent data structures are crash consistent (i.e., power failures) is a major challenge. Stores to persistent memory are not immediately made persistent — they initially reside in processor cache and are only written to PM when a flush occurs due to space constraints or explicit flush instructions. It is more challenging to test crash consistency for PM than for disks given the PM's byte-addressability that leads to significantly more states.

We present Jaaru, a fully-automated and ultra-efficient model checker for PM programs. Key to Jaaru’s efficiency is a new technique based on constraint refinement that can reduce the number of executions that must be explored by many orders of magnitude. This exploration technique effectively leverages commit stores, a common coding pattern, to reduce the model checking complexity from exponential in the length of program executions to quadratic. We have evaluated Jaaru with PMDK and RECIPE, and found 25 persistency bugs, 18 of which are new. Jaaru is also orders of magnitude more efficient than Yat, a model checker that eagerly explores all possible states.

CCS CONCEPTS
- Hardware → Memory and dense storage; • Software and its engineering → Software verification and validation.

KEYWORDS
Persistent Memory, Crash Consistency, Debugging, Testing

ACM Reference Format:

1 INTRODUCTION
Persistent memory (PM) technologies, such as phase change memory (PCM) [31, 51, 55], resistive random-access memory (RRAM) [50], Spin-Transfer Torque memory (STT-MRAM) [28], or 3D XPoint [11], promise to combine the performance and flexibility of DRAM with the persistency of flash storage. As commercially available in the Intel Optane memory product [12], persistent memory can interface with the processor via the memory bus, providing byte-addressable access for a program via regular store and load instructions. Such instructions bypass the OS kernel, offering a flexible and yet efficient interface to storage.

Persistent memory can potentially change the way programs manipulate data structures to achieve greater performance—with PM, programs can use a single copy of a data structure both as an in-memory working data structure and as a persistent store of the data, eliminating the serialization and deserialization process. Failures are a key challenge in realizing this approach—stores are not immediately written to persistent memory; they are initially written to the processor cache and the persistent memory is only eventually updated when the cache line is written back.

Modern processors provide special instructions to force cache lines to be written to persistent storage. Using these instructions correctly is challenging—it requires both subtle reasoning about the ordering of memory operations and attention to detail to not miss persisting any of the many stores a program may perform. Moreover, testing the correctness of persistent storage code w.r.t. failures is challenging. Exposing a bug requires that the machine fails at a specific instruction and depends on the state of the cache before the failure.

State of the art. The problem of PM consistency has received much attention. There is a line of recent work on testing/dynamically checking a PM program to find consistency-related bugs. XDeter
...
These testing-based bug-finding tools suffer from two major drawbacks: (1) They need users to add extra annotations for various cache line flushing properties, which not only incur burdens on users but also are error-prone themselves. Consequently, if the developer misses an annotation or adds an incorrect annotation, the tool will have false negatives and miss real bugs or have false positives and report bugs that are not real. (2) Violations they report are with respect to design principles and may or may not correspond to actual bugs, e.g., certain tools report data has not been flushed. However, in some cases, the data may never be accessed in future executions. Thus, the absence of a flush is a false positive that does not represent a real bug. These drawbacks call for techniques such as model checking that can exhaustively explore states without needing manual effort and provide strong witnesses (e.g., executions) for bugs exposed.

Model checking has been used extensively in the systems community (e.g., EXPLODE [53], FISC [54], or SAMC [33]) to find bugs in file/storage systems. However, there are several fundamental differences between the file system bug problem and persistent memory crash consistency problem that preclude direct application of existing model checkers in the PM setting: (1) disks have a fundamentally different programming interface than PM — updates to a disk block are only made upon making an explicit write request, (2) disks have a larger block size and therefore there are fewer possible states to enumerate, and (3) operating systems receive explicit notifications of when disk blocks are written. All of these factors combined indicate that the state space to be explored for model checking disks is significantly smaller than that for PM programs.

In fact, a recent technique Yat [29] attempts to use an eager model checking approach to enumerate all possible post-failure memory states for a PM program before it is aware of what parts of the state the post-failure execution will read from. Since the number of memory states that must be explored grows exponentially with the number of stores that have not been flushed to memory, Yat cannot scale. For example, consider the common scenario of code that allocates a cache line aligned array of n 64-bit integers, initializes the data, and crashes right before flush operations for that array. This array spans n/8 cache lines and the persistent memory copy of each cache line has 9 possible states (i.e., the initial value and the state after each of the 8 writes). Therefore, persistent memory has \(9^{n/8}\) possible states that Yat must explore.

**Our approach.** We develop Jaaru, a **fully-automated and ultra-efficient** model checker for PM programs that achieves many *orders-of-magnitude* reductions in the number of states that must be explored, compared to eager techniques such as Yat. It does not require any user annotation; as a model checker, Jaaru exhaustively explores all possible states and can potentially find more bugs than testing-based techniques.

Key to Jaaru’s efficiency is a *constraint-refinement based technique* that effectively leverages commit stores — a common programming practice in data structure implementations to drastically reduce the space of executions. We elaborate on this insight below.

As stated above, a major challenge in model checking PM programs is the enormous post-failure state space the model checker must explore — a store writes a value into the cache, and the value is not persisted until the cache line is flushed. However, when a failure occurs, it is unclear whether a cache line has been flushed yet, leading to a large number of possibilities that the model checker must explicitly enumerate.

To solve this problem, our major insight is that we can exhaustively explore all executions by enumerating only a subset of post-failure states using constraints on the time at which a cache line was previously flushed. A `clflush` or `clflushopt` instruction flushes a cache line, imposing a constraint on the possible values that a persistent variable can have after the failure. Jaaru **builds** such constraints during a pre-failure execution and **refines** them during a post-failure execution (see §3.1). Leveraging these constraints in partial order reduction [15, 56] enables Jaaru to explore exactly one post-failure state for each *equivalence class* of post-failure executions, defined by which pre-failure stores are read by post-failure loads.

To effectively leverage this insight, we made an observation that there are often many stores that have not been flushed out to persistent memory, PM programs often record in some fashion, using a commit store, whether data is in a consistent state (see §3.2). For example, when adding a subtree to a node, the store of the node pointer to the subtree is a commit store. Post-failure PM programs then read from this commit store to determine whether data is consistent. This is a common practice in data structure implementations (1) because the information about consistency also provides a reference to where the data is stored (e.g., if the pointer from the node to subtree is null, the subtree is not persisted; otherwise, it can be found by following the pointer) and (2) for efficiency purposes. Such checks explicitly prevent the post-failure execution from accessing many unflushed stores (e.g., if the pointer is null, the program cannot access any data protected by the pointer).

This pattern offers an opportunity for us to not explicitly enumerate all possible states at a failure — **lazily** enumerating the stores read by the actual loads in the post-failure execution, as opposed to eagerly enumerating all of them, reduces the number of executions to be explored from exponential in the length of the program execution to linear (see §3.2). This observation leads to the **lazy exploration approach** used in Jaaru, which does not enumerate stores until loads are executed in the recovery code.

Note that leveraging such a programming pattern leads to efficiency, but has nothing to do with the thoroughness of the state search — Jaaru always exhaustively explores all the nondeterminism that arises from the persistency of cache lines. As a result, **Jaaru does not generate any false positives or negatives** — it reports all bugs w.r.t. an input and any bug it reports must be a real bug. For programs that do not obey such a programming idiom (e.g., the recovery code directly reads the data without checking consistency), Jaaru would not miss any bug, but it would certainly spend more time on state exploration. In practice, however, Jaaru is often still efficient because PM programs are extremely unlikely to read from many non-flushed cache lines.

**Usage scenarios.** Despite the aforementioned advantages, model checking is not a silver bullet for bug-finding in PM programs. For example, even though Jaaru is orders of magnitude more efficient than existing model checkers such as Yat, Jaaru still needs to execute a program many times (e.g., between 24 and 891 in our experiments) to fully explore the state space, taking a large amount of time for
checking. Compared to testing tools such as PMTest and XFDetector, Jaaru is able to find more bugs, in a completely automated fashion. However, it has difficulty checking programs such as Redis that interact with the outside world and whose non-determinism from the network would require deterministic replay for a model checker to work. As such, the best use case for Jaaru is to exhaustively check widely-used libraries such as PMDK, finding as many potential bugs as possible before their release, while non-exhaustive tools such as PMTest and XFDetector can scalably check large programs and find bugs only when they are triggered in tests.

**Summary of Results.** We have implemented Jaaru which incorporates a full simulation of the underlying TSO memory model including support for store buffering, buffering flush operations, and buffering _sfence_ operations. We evaluate Jaaru with PMDK [13] and RECIPE [32]: Jaaru is effective at finding persistency bugs in our benchmark set. Jaaru finds 18 new correctness bugs in extensively studied PM programs, while PMTest and XFDetector finds only 1 and 4 correctness bugs, respectively.

## 2 OVERVIEW OF X86 PERSISTENT MEMORY STORAGE

We next overview the Intel-x86 persistent storage system. We refer interested readers to the Px86sim model in Raad et al. [43]. Figure 1 presents a graphical overview of the x86-TSO storage system. Each core/thread on x86 has a store buffer that buffers stores to the cache to hide the store latency. The store buffers implement bypassing — when a core performs a load, the core checks whether there is a store to the same address in its local store buffer. If so, it returns the value written by the most recent such store. Effectively, this allows the local core to observe the effect of a local store before that store becomes visible to other cores. The memory fence instruction _sfence_ waits for the store buffer to empty before future instructions can be executed. Locked _mov_ instructions also clear the store buffer before future instructions can be executed.

Stores in the store buffer are written to the cache in the order they were executed — they are written to the cache in a total order and all other threads/cores observe these stores in that same order. The cache is volatile — a power loss event will cause cached data that has not yet been written back to persistent storage to be lost. Under normal execution, cache lines are written back to main memory non-deterministically when the cache needs the space for other data. The x86 architecture provides instructions to force the cache to write data back to persistent storage. The three such instructions are: (1) the flush cache line instruction _clflush_ that flushes a cache line, (2) the optimized flush cache line instruction _clflushopt_, and (3) the cache line write back instruction _clwb_.

A key difference between these instructions is how they can be reordered across other instructions. Table 1 summarizes the instruction ordering constraints for persistent storage on x86-TSO. The _clflush_ instruction is inserted into the store buffer just like store instructions, and when it exits the store buffer it causes the cache line to be flushed to persistent memory. The _clflushopt_ instruction is inserted into the store buffer also like store instructions, but it can be reordered across store instructions to other cache lines, _clflush_ instructions to other cache lines, and other _clflushopt_ instructions. The _clflushopt_ instruction cannot be reordered across _sfence_ or locked _mov_ instructions. The store fence instruction _sfence_ also orders _clflushopt_ instructions relative to _clflush, clflushopt_, _clwb_, and store instructions. The _clwb_ instruction only writes back the contents of the cache line and does not evict it from the cache and thus has better performance. However, from a semantics perspective, the _clwb_ instruction is identical to the _clflushopt_ instruction [43], and thus we treat them identically in this paper.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Read</th>
<th>Write</th>
<th>RMW</th>
<th><em>mfsnop</em></th>
<th><em>sfence</em></th>
<th><em>clflushopt</em></th>
<th><em>clflush</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mov</em></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Table 1: Summary of reordering constraints in the Px86sim model.

A ✓ indicates that the two instructions can be reordered, and a CL indicates that the order is preserved only if they both operate on the same cache line. These constraints are also used in Raad et al. [43].

![Figure 1: An x86-TSO storage system.](image-url)

Recall that prior work (e.g., Yat [29]) on model checking persistent memory programs eagerly enumerates all possible post-failure states of persistent memory. As the number of states grows exponentially with the amount of data that has not been flushed, this approach can easily have scalability problems. Such eager approaches will explore many post-failure states that yield identical post-failure executions in which the loads read from the same stores. Dynamic partial order reduction (DPOR) [1, 15, 56] is a popular technique that can determine that these states produce the same execution, and instead explore the equivalent post-failure executions once.

### 3 BASIC IDEAS

Recall that prior work (e.g., Yat [29]) on model checking persistent memory programs eagerly enumerates all possible post-failure states of persistent memory. As the number of states grows exponentially with the amount of data that has not been flushed, this approach can easily have scalability problems. Such eager approaches will explore many post-failure states that yield identical post-failure executions in which the loads read from the same stores. Dynamic partial order reduction (DPOR) [1, 15, 56] is a popular technique that can determine that these states produce the same execution, and instead explore the equivalent post-failure executions once.
3.1 Constraint-Refinement

Traditional DPOR techniques do not consider the effect of cache line flushes and volatile memory. Naïve adaptation of these techniques in our setting would lead to the exploration of many states that are not possible due to the use of instructions such as clflush that explicitly flush cache lines.

To reduce search space, our first idea is to use clflush instructions to infer constraints on the last time each cache line was written back to persistent memory in a pre-failure execution and refine these constraints in a post-failure execution to narrow down when a cache line became persistent. For example, when a clflush instruction leaves the store buffer, it forces the cache line to be written back to persistent memory. That same cache line can later be written back to persistent memory due to space constraints in the cache. Hence, the clflush instruction essentially sets a constraint that the last time the corresponding cache line is written back to memory must be after the clflush instruction exits the store buffer.

Figure 2 illustrates the application of this idea on an execution prior to a failure. The program executes the instruction sequence on the left-hand side prior to the failure. The blue line shows the order that stores were written to the cache. Both variables x and y are located in the same cache line. After the program executes the stores y = 1 and x = 2, it performs a clflush instruction. This instruction flushes the cache line that holds x and y to persistent memory. At this point, Jaaru computes that the cache line for x and y was most recently flushed during the interval [clflush, oo) as represented by the red line in Figure 2. After the clflush, the program performs the stores y = 3, x = 4, y = 5, and x = 6. Finally, powe is lost and the program fails. The red interval indicates that when the machine is powered back up, the persistent storage may have the values 2, 4, and 6 for the variable x.

Note that there are constraints between the values for variable x and those for y since they share a cache line. For example, it is not possible for the post-failure state of the persistent memory to have y = 1 and x = 6, because the store y = 5 is ordered between y = 1 and x = 6. To ensure that variables that share a cache line have consistent values, Jaaru refines these intervals using the values observed by loads during the recovery execution. Figure 3 shows a post-failure execution. This execution reads the value 4 from the variable x. This tells us that the cache line must have been flushed some time after the store x = 4 and before the store x = 6. Thus, we can refine the interval for the most recent flush to be [x = 4, x = 6), which imposes a much tighter bound.

Since both variables x and y share the same cache line, reading the value 4 for x constrains the set of values that we can read from y. In particular, since the last flush occurred some time during the interval from x = 4 to x = 6, we know that the cache line was flushed some time after the assignment y = 3 and potentially after the assignment y = 5. Therefore, if the post-failure execution reads from y, it could only read the value 3 or 5. It could not read the value y = 1, because the fact that the read from x returned 4 tells us that the cache line was flushed after y = 1 was overwritten.

Jaaru uses this refinement-based approach to simulate cache line flushes and lazily construct the state of persistent memory after the failure, eliminating the need to eagerly explore all (equivalence classes of) states.

3.2 Leveraging Commit Stores for Additional Efficiency

Our constraint-refinement approach works well for PM programs because it effectively leverages commit stores to achieve efficiency. Commit stores are a rather common programming practice; in fact, all programs in our evaluation have such commit stores. To effectively leverage such stores, Jaaru does not eagerly enumerate all pre-failure stores; instead, Jaaru lazily enumerates a small subset of them that are actually read by a post-failure execution.

```c
void addChild(node *ptr, char *data) {
    charNode *tmp = alloc_child();
    tmp->data = data;
    clflush(tmp, sizeof(charNode));
    ptr->child = tmp;
    clflush(&ptr->child, sizeof(charNode));
}
```

Figure 4: An example program with a commit store.

To illustrate, Figure 4 presents a simple program that uses a commit store. There are two methods here — method addChild that adds a child to store data and method readChild that returns a pointer to the data stored in the child. We first discuss the addChild
method. The store at Line 3 writes a reference to the data field in the newly created child node. Next, the clflush instruction at Line 4 forces this write to persistent memory. Finally, the commit store at Line 5 makes the child node reachable from the data structure and the clflush at Line 6 makes the commit store persistent.

We next discuss the readChild method. The load at Line 10 checks whether the child field is non-null. If it is, then we know that (1) the clflush instruction at Line 6 completed and (2) the child node has been persisted and is safe to read in Line 11.

To illustrate why Jaaru leverages this pattern for efficient state exploration, let us consider a client program that executes method addChild, fails, and then calls the readChild method during recovery. Jaaru injects failures in the execution of method addChild at three points: (1) immediately before the clflush instruction at Line 4, (2) immediately before the clflush instruction at Line 6, and (3) at the end of the execution of method addChild. Injecting failures at these three points is sufficient to explore all distinct program behaviors (see §4). While Jaaru supports failure scenarios that involve crashes in the recovery routine, for simplicity, we focus on a single failure point.

To inject a failure, Jaaru stops the execution at the failure point, resets volatile memory, and starts a new execution with the same persistent memory region. In the new execution, loads from persistent memory check the stores from the pre-failure execution to determine which values the program will read from.

Let us first consider the failure immediately before Line 4. Since the clflush instruction has not executed, the write to the data field may not have been persisted. When the readChild method executes, it first reads the child field. Since the child field is null, it does not access the data field. Jaaru explores exactly one post-failure execution for this failure point.

Next, consider the failure immediately before Line 6. The data field has been persisted by the first clflush instruction, but the write to the child field has not. Thus, when the post-failure execution reads from the child field, Jaaru observes that the interval for the most recent flush of the child field is \([0, \infty)\). Jaaru then explores two executions. In the first execution, the child field is null, and this execution has the same behavior as the previously explored execution. In the second execution, the child field is non-null and thus it reads the data field. Since the interval \([\text{clflush}_4, \infty)\) for the data field’s cache line starts after the last write to the data field, the method returns the data field (clflush4 denotes the clflush instruction at Line 4).

Finally, consider the failure at the end of the execution of method addChild. At this point, both clflush instructions have executed. When the post-failure execution reads from the child field, Jaaru observes that the interval for the most recent flush of the child field is \([\text{clflush}_6, \infty)\). Therefore, the load must see the value written to the child field and thus it reads the data field. Since the interval \([\text{clflush}_6, \infty)\) for the cache line of the data field starts after the last write to the data field, the method returns data.

To illustrate why such stores are useful, consider the following scenario. Suppose that method readChild accesses the data field of the child node without first checking the commit store in Line 5. If the addChild method crashes before the first clflush instruction, there would be two different potential post-failure states for the data field. If the child node has \(n\) different cache lines that were accessed in a similar manner, then the number of post-failure states would grow to \(O(2^n)\). If the post-failure code accesses all of the child’s states, the model checker would have to explore \(O(2^n)\) executions. The commit store limits the number of unflushed stores that the post-failure program execution reads from, and thus the executions Jaaru must explore.

The complexity of model checking programs that use commit stores like this example is \(O(m^2)\) where \(m\) is the length of the execution. We obtain this complexity because the number of failure injection points is \(O(m)\), the post-failure execution involves \(O(m)\) steps, and with commit stores, we explore two executions at each failure point — a first execution that reads from the commit store and a second execution that reads the value of the memory location before the commit store.

Note that prior techniques that eagerly explore all pre-failure stores cannot take advantage of such commit stores. The key difference between prior model checkers such as Yat and Jaaru is that Yat enumerates all possible states at the failure point before executing the post-failure code (thus with a complexity of \(O(2^n)\)) while Jaaru executes the post-failure code and lazily explores pre-failure stores that are actually read by the post-failure code.

### 3.3 System Overview
Jaaru uses an LLVM compiler pass to instrument both atomic and normal memory accesses along with fences and cache flush operations. The instrumented binary is then dynamically linked with the Jaaru library. Figure 5 presents an overview of Jaaru. A failure scenario involves multiple executions — the simplest failure scenario (a single failure) involves a pre-failure execution and a post-failure execution. To simulate a failure scenario, Jaaru keeps the information about each of the executions in the sequence that comprises the failure scenario. Figure 6 shows the exploration of a failure sequence composed of a pre-failure execution and the current post-failure execution.

Jaaru uses a fork-based approach to roll back executions to simulate failures and start new executions. In each execution, Jaaru records all of the stores that have been written to the cache and the clflush instructions that have taken effect (shown with the
This section presents the model checking algorithm. We begin by presenting the following notations that we will use throughout the paper:

- We refer to an execution as \( \epsilon \).
- A given failure scenario may involve a sequence of multiple executions ending in failures. We record this sequence of executions that have been executed on the persistent store using a stack, referred to as \( \text{exec} \).
- Function \( \text{top} (\text{exec}) \) denotes the most recent execution (the current one) on the stack \( \text{exec} \).
- Function \( \text{prev} (\epsilon) \) returns the execution that immediately precedes \( \epsilon \) in \( \text{exec} \).
- A global sequence number counter \( \sigma_{\text{curr}} \) is used to assign increasing sequence numbers to stores, clflush, sfence instructions.
- Each store, clflush, and sfence instruction \( i \) is assigned a sequence number \( \sigma_i \). These numbers record the total order in which these instructions take effect in the cache.
- Each execution \( \epsilon \) has a map \( \text{getcacheline} () \) that maps an address to an interval in which the cache line was most recently flushed to persistent memory in the execution \( \epsilon \).
- Each execution \( \epsilon \) has a map \( \text{queue} () \) that maps each address \( \text{addr} \) to a sequence of tuples \( (\text{val}, \sigma) \) that record the values stored at the address and the sequence number \( \sigma \) generated at the moment that value was stored.
- We denote a thread using \( \tau \in T \).
- Each thread \( \tau \) has a store buffer \( S_\tau \) that keeps a queue of store, clflush, and sfence operations that have not yet taken effect in the cache.

- Each thread \( \tau \) has a cache line flush buffer \( F_\tau \) that stores the set of clflushopt operations that have not yet flushed the cache line to persistent storage.
- We refer to the timestamp as \( t \).

The Jaaru LLVM frontend instruments only memory operations and cache operations as those are the operations relevant to persistent storage. Jaaru implements a software simulation of those instructions with full support for the persistency semantics from the Psx86sim model \[43\]. The majority of PM-based tools have been developed for x86 since it provides the most advanced and mature architectural support for accessing persistent memory. By fully supporting x86 semantics, Jaaru satisfies the fast-growing need for a scalable and fast model checker to validate and test these programs. Although the current version of Jaaru is developed for x86, the primary idea behind it is not limited to x86 and could potentially be adapted to support other architectures such as ARM.

The TSO memory model separates the executions of stores, cache flush operations, and sfence operations into two phases: (1) the initial phase that often inserts an operation into a buffer and (2) the second phase that removes the instruction from the buffer and updates the state of the cache or persistent storage. We present our algorithm for each of the stages.

**Executing instructions.** Figure 7 presents our algorithm for the first phase of instruction execution, which inserts an instruction into each thread’s local store buffer \( S_\tau \). The sfence instruction waits until \( S_\tau \) is empty and then clears the thread’s flush buffer \( F_\tau \).

**Updating storage.** The second phase occurs when the instruction leaves the store buffer. This phase updates the storage system. Figure 8 presents our algorithm for this phase. We have four different implementations of the Evict_SB function for different types of instructions.

The Evict_SB((store, addr, val)) function handles store instructions. This function assigns a sequence number to each store. These sequence numbers enforce a total order over all writes to the cache. The function then moves the store to the queue of stores that records possible cache line values based upon the address it writes to. Finally, the function updates the timestamp \( t_{\text{CacheID} (\text{addr})} \) for the most recent write to the cache line or clflush from this thread to be the store’s sequence number.

The Evict_SB((clflush, addr)) function handles the cache line flush instruction clflush. The function first assigns a unique sequence number to the instruction. It then updates the lower bound of when the cache line was most recently flushed to be the sequence number.
and check whether there is a store to read from in the store
present our algorithm for
check whether there is a store in the current execution
was first executed, (2) the sequence number of the most recent
with later instructions is implemented by a flush buffer that is emp-
clflushopt
(1) the current sequence number when the
instruction executed by the thread, (3) the sequence number
clflush
of the most recent store to the same cache line executed by the
instruction executed by the thread, (4) the sequence number

Finally, the
Evict_FB
function handles the store fence in-
recursive call.

Evict_FB
function handles the store fence in-
instruction sfence. This sfence instruction is ordered relative to all
previous clflushopt instructions and thus it flushes the thread’s
flush buffer when it exits the thread’s store buffer.

Finally, the
Evict_FB
function handles clflushopt instructions when they are evicted from the flush buffer by an
sfence, mfence, or RMW instruction. This function updates the lower
bound of when the cache line was most recently flushed to be the
sequence number σ of the tuple in the flush buffer. Recall that
this sequence number is the maximum of the following four values:
(1) the current sequence number when the clflushopt instruction
was first executed, (2) the sequence number of the most recent
sfence instruction executed by the thread, (3) the sequence number of the most recent store to the same cache line executed by the
same thread, or (4) the sequence number of the most recent clflush
to the same cache line executed by the same thread.

Load operations. Figures 9 and 10 present our algorithm for
loads. We split handling of loads into two functions: (1) the Build-
MayReadFrom function that computes and returns a set of stores
that a load may read from and (2) the DoRead function that refines
the cache line flush intervals once Jaaru has selected a specific
store for the load to read from. Splitting the load handling into two components makes it straightforward to integrate loads into Jaaru’s exploration.

We first discuss the
BuildMayReadFrom function in Figure 9.
This function returns a set of tuples for each possible store that
the load may read from. Each tuple contains the execution e that
performed the store, the sequence number σ of the store, and the
value val stored. We use _ when the store is from the current
execution and thus does not have a sequence number that can be
used to constrain when a cache line was last flushed in the previous
execution.

Lines 2–3 check whether there is a store to read from in the
store buffer, and if so, returns the tuple for the newest such store. More
precisely, the syntax b₁, ⟨addr, val⟩, b₂ represents the state of store
buffer with b₁ being the oldest operations and b₂ being the newest
operations. A load can read from a store ⟨addr, val⟩ in the store
buffer if there are no newer stores to the same address.

Lines 4–5 check whether there is a store in the current execution
that has updated the cache. If so, they return the tuple for that store.
More precisely, the syntax m₁, ⟨val, σ⟩ represents the sequence of stores written to the cache with m₁ being the older operations. A load can read from a store ⟨addr, val⟩ in the cache queue for an address
if there are no newer stores to the same address. Otherwise, Line 6
invokes the
ReadPreFailure function to compute potential stores from
the executions before the most recent failure.

We next discuss the
ReadPreFailure function in Figure 9. This function computes the set of stores from previous executions that a
load may read from. Lines 8–9 compute the set of stores that would
have been present on the cache line for the time range specified by
the cache line’s last flush interval. Line 10 checks whether there was
a store performed before the earliest possible time for the cache line
flush. If there is no such store, it is possible that the load has read
from an earlier execution. In this case, the algorithm recursively
calls
ReadPreFailure on earlier executions and combines the set
of stores from the current execution with those returned by the
recursive call.

After the model checking algorithm has selected a store for the
load to read from, it invokes the
DoRead function in Figure 10
to refine the most recent cache line flush intervals for previous
executions. Line 2 checks whether the store is from the current
execution. If so, there is no refinement to be performed and the
function returns. Otherwise, it calls the function
UpdateRanges to refine the interval in which the last cache
write was performed.
Injecting failures. The natural points to inject failures are those immediately before operations that flush cache lines. The reason is that writes to the cache increase the set of possible post-failure executions while flushes decrease the set of possible post-failure executions. Thus, injecting failures at these points is sufficient to explore all program behaviors. Jaaru, therefore, injects failures at those points.

For long runs or scenarios in which multiple failures are injected, injecting failures before every flush can result in exploring many executions. Jaaru contains an optimization that skips injecting a failure if there have been no writes since the last injected failure. Jaaru can also support injecting failures into a post-failure execution (with a command line option). This option controls the maximum depth of the exec stack.

Locked RMW instructions. Locked (atomic) read-modify-write instructions include compare-and-swap (CAS), atomic exchange, and many atomic arithmetic instructions. On x86 these instructions also have fence-like semantics. They are equivalent to the atomic execution of the following sequence of instructions: `sfence`, load, store, and `sfence`. Jaaru implements them by atomically executing this sequence.

Mixed size accesses. C and C++ programs may access fields using stores and loads with different widths. For example, a 32-bit integer field in a union may be initialized to 0 with a 64-bit integer store and then read with a 32-bit integer load. We implement accesses that are larger than a byte as a sequence of byte accesses that are performed atomically. Thus, a 32-bit load is implemented as four 8-bit loads.

Checksum-based recovery. One approach to ensure data persistence is to write a checksum along with data — recovery code reads the checksum to verify that the data was persisted. Checksum-based recovery differs from other approaches in that the recovery code may read from a larger number of non-persisted stores. Jaaru provides special support that can exhaustively check programs that use checksum-based recovery without explicit flushes.

Debugging support. Jaaru can identify different types of bugs including missing fences, misordered flushes, missing flushes, and misordered stores that cause atomicity violations. The most common bug type we found was due to missing cache line flush instructions. Looking at the entire trace to understand a bug is not easy. Therefore, we extend Jaaru with additional functionality to help developers quickly determine why a program crashed.

Our observation is that a missing flush instruction effectively increases the number of pre-failure stores that a post-failure load may read from. Jaaru, therefore, contains optional support for flagging loads that can read from more than one store. To facilitate debugging, Jaaru prints out the load that can read from multiple stores, the source location of the load, each of the stores, their locations in the trace, and their source locations. Our experience shows that this information is very useful for quickly understanding missing flush instructions that cause the program to crash or loop.

Discussion. Existing DPOR algorithms [8, 15, 30, 45–47, 49] are not directly applicable in the setting of persistent memory. None of the traditional DPOR algorithms consider the effect of volatile memory such as caches and cache flushes. For example, a crash makes
pre-failure stores that were executed but not written to persistent memory completely disappear.

Jaaru can be viewed as implementing a form of dynamic partial order reduction that avoids exploring equivalent executions. Pre-failure executions that differ in when cache lines are flushed and thus generate different post-failure states can still yield the same post-failure executions if the post-failure executions never read from the memory locations that contain different values. Such cache line flushes can be viewed as commuting with the crash. Other cache line flushes make stores visible to post-failure loads and thus do not commute with the crash. Jaaru’s constraint refinement algorithm lazily identifies non-commuting cache line operations during the post-failure execution and effectively explores reordering such cache line flushes.

Many PM programs are multi-threaded, creating the opportunity for concurrency bugs. Jaaru does not exhaustively explore all concurrent schedules and thus does not provide any guarantees that it will find concurrency bugs. However, since Jaaru controls the concurrent schedule and fully simulates the TSO memory model, as future work, it can be used to fuzz for concurrency bugs.

5 EVALUATION

In this section, we evaluate Jaaru’s bug-finding capabilities and performance with a set of benchmarks. Our system configuration is reported in Table 2.

<table>
<thead>
<tr>
<th>#</th>
<th>Benchmark</th>
<th>Symptom</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Btree*</td>
<td>Illegal memory access at btree_map.c:89</td>
</tr>
<tr>
<td>2</td>
<td>Btree</td>
<td>Failed to open pool error</td>
</tr>
<tr>
<td>3</td>
<td>Hashmap.atomic*</td>
<td>Assertion failure at heap.c:533</td>
</tr>
<tr>
<td>4</td>
<td>CTree*</td>
<td>Assertion failure at obj.c:1523</td>
</tr>
<tr>
<td>5</td>
<td>Hashmap.atomic*</td>
<td>Assertion failure at pmalloc.c:270</td>
</tr>
<tr>
<td>6</td>
<td>Hashmap tx*</td>
<td>Illegal memory access at obj.c:1528</td>
</tr>
<tr>
<td>7</td>
<td>RBTree*</td>
<td>Illegal memory access at btree_map.c:137</td>
</tr>
</tbody>
</table>

Figure 12: Bugs found in PMDK. Bugs with a * are new bugs. Only the second bug was reported before in XFDetector [36].

<table>
<thead>
<tr>
<th>#</th>
<th>Benchmark</th>
<th>Type of Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CCEH*</td>
<td>Missing flush in CCEH constructor</td>
</tr>
<tr>
<td>2</td>
<td>CCEH</td>
<td>Missing flush in CCEH constructor</td>
</tr>
<tr>
<td>3</td>
<td>CCEH*</td>
<td>Missing flush in CCEH constructor</td>
</tr>
<tr>
<td>4</td>
<td>FAST Fair</td>
<td>Missing flush in header constructor</td>
</tr>
<tr>
<td>5</td>
<td>FAST Fair</td>
<td>Missing flush in entry constructor</td>
</tr>
<tr>
<td>6</td>
<td>FAST Fair*</td>
<td>Missing flush in btree constructor</td>
</tr>
<tr>
<td>7</td>
<td>P-ART*</td>
<td>Use of non-persistent data structure in Epoch</td>
</tr>
<tr>
<td>8</td>
<td>P-ART*</td>
<td>Use of non-persistent data structure for recovery</td>
</tr>
<tr>
<td>9</td>
<td>P-BwTree*</td>
<td>GC crash leaves data structure in inconsistent state</td>
</tr>
<tr>
<td>10</td>
<td>P-BwTree*</td>
<td>Missing flush of GC metadata pointer</td>
</tr>
<tr>
<td>11</td>
<td>P-BwTree*</td>
<td>Missing flush in AllocationMeta constructor</td>
</tr>
<tr>
<td>12</td>
<td>P-BwTree*</td>
<td>Missing flush in BwTree constructor</td>
</tr>
<tr>
<td>13</td>
<td>P-CLHT</td>
<td>Missing flush in clht constructor</td>
</tr>
<tr>
<td>14</td>
<td>P-CLHT</td>
<td>Missing flush for hashtable object</td>
</tr>
<tr>
<td>15</td>
<td>P-CLHT</td>
<td>Missing flush for hashtable array</td>
</tr>
<tr>
<td>16</td>
<td>P-CLHT</td>
<td>Flushed referenced object instead of pointer</td>
</tr>
</tbody>
</table>

Figure 13: Bugs were found by Jaaru in every program of RECIPE. Bugs with a * are new bugs.
We next discuss results for the RECIPE benchmarks. We have found 12 new bugs in the RECIPE programs. Many programs contain multiple bugs. When Jaaru has found an execution that causes the program to crash (or loop) we have examined Jaaru’s outputted trace and debugging information to understand the bug. Since these benchmarks are easier to understand than PMDK benchmarks, we have fixed the bug and used Jaaru to look for additional bugs. We continued this until the program executed correctly.

Figure 13 presents the bugs we have found. We confirmed that each bug caused the program to crash. Jaaru found bugs in every program. These bugs are primarily missing flush instructions in object constructors. All of the bugs can potentially corrupt a persistent data structure leading to data loss.

Many bugs are simple cases of forgetting to flush stores or mistakenly flushing the wrong memory location. However, we have found other kinds of bugs. In P-ART, the developer has used a vector data structure from tbb to track locks that must be unlocked in the recovery procedure. The bug is that tbb data structures do not persist across failures. In P-BwTree, Jaaru has found a logical error in the garbage collection (GC) algorithm in which failures during the GC can corrupt the GC data structures. This bug is an atomicity violation and not a case of missing flushes.

Comparing these results with the bugs found by PMTest [37] and XFDetector [36], Jaaru appears to have a stronger bug-finding ability than PMTest and XFDetector. For example, PMTest reported three new bugs and XFDetector reported four; several of these bugs were performance bugs. On the contrary, Jaaru found serious functional bugs that can corrupt data structures and lead to a crash or an assertion failure in the program. This is not surprising because Jaaru explores many more states than PMTest and XFDetector, which focus on a single execution.

Among the several bugs reported before, three were not found by Jaaru. We inspected those bugs and found it was because (1) two were performance bugs that are not our focus and (2) one was in the Redis code which we did not test. Jaaru could be extended to find performance bugs such as redundant cache flushes and fences.

**Jaaru Bug Reporting.** We presented Jaaru and the bugs found by our tool to the authors of RECIPE and we received overall positive feedback. At the time of writing, 6 out of 18 bugs found by Jaaru were fixed by the developers of RECIPE. There were 6 bugs that were related to memory allocators and garbage collectors. The RECIPE developers did not fix the persistence bugs related to memory allocators because they believe these bugs need to be addressed by the memory allocators, which is not their focus. The remainder of the bugs were already fixed before our bug report.

### 5.2 Performance

Figure 14 presents the performance results for Jaaru on RECIPE benchmarks. Providing performance results for a model checker requires first fixing the bugs we have found so that Jaaru can run to completion and fully explore the state space of the program; otherwise, it would not make sense to report running time. We have spent much time fixing all the bugs we have found in RECIPE so that the model checker can fully explore these benchmarks. The bugs in the PMDK framework are more complicated and would take more time to fix, so we did not include our performance results for PMDK. Note that Jaaru is able to model check each RECIPE program in less than 15 seconds. We next discuss our evaluation of the state space reduction that Jaaru achieves on these programs, relative to an eager model checking approach such as that implemented in Yat [29]. Since Yat is not publicly available, we have calculated the number of legal post-failure states that Yat would have to explore.

Figure 14 presents these results. Given the very large number of executions Yat would have to explore, it is unlikely to be feasible to exhaustively model check these realistic programs with Yat.

To better understand Jaaru’s effectiveness, we compare the total number of executions with the number of failure injection points in the original execution. As shown in Figure 14, Jaaru only explores a few executions per failure injection point. The number of executions per failure injection point ranges from 1.5 to slightly less than 8.

It does not make much sense to compare performance directly between Jaaru and non-exhaustive approaches such as PMTest and XFDetector, which detect bugs on single executions. However, as a reference, Jaaru incurs an overall slowdown of 736× per execution, which is on par with the overhead of XFDetector (i.e., from dozens of times to almost a thousand times as reported in the paper [36]). PMTest and Pmemcheck have much lower overhead (1.69× and 22.3×, respectively). This is because Jaaru fully simulates the x86 TSO persistence semantics while the other tools ignore the effects of store buffers.

**Key Takeaway.** Our results highlight the strengths and weaknesses of model checking: Jaaru finds more bugs without any user involvement, but cannot easily handle programs with complex interactions with the outside world. Jaaru is a good fit for checking library code that is usually small in size but has a large impact. Non-exhaustive tools such as PMTest and XFDetector should be used to check large programs such as Redis whose non-determinism from the network can give a model checker much trouble. It is also clear that the constraint refinement approach enables Jaaru to efficiently check these programs; without refinement, it would not be possible for a model checker to scale even to library code.

### 6 RELATED WORK

**Bug/crash consistency detection.** There exists a large body of work on testing [26, 29, 39, 52], checking [38, 44, 53, 54], and formally verifying [9, 10, 48] file system implementations to find and eliminate crash consistency bugs. Fuzzing techniques such as Janus [52] and Hydra [26] mutate disk images and file operations.
to explore states of file system code. Using heuristics, B3 [39] employs a bounded testing technique to explore states in a bounded space. EXPLODE [53], FiSC [54], and SAMC [33] use model checkers to systematically explore states of a file system implementation. Although crash consistency bugs in file systems bear similarities with bugs in PM programs, they are fundamentally different in the access granularity as well as how writes are performed.

There is a recent line of work on checking/testing PM programs to find bugs. In particular, XFDetector [36] uses a finite state machine to track the consistency and persistency of persistent data. PMTest [37] lets developers annotate a program with checking rules to infer the persistency status of writes and ordering constraints between writes. Pmemcheck [25] checks how many stores were not made persistent and detects memory overwrites using binary rewriting. Although these tools are able to find many bugs, none of these tools can systematically explore the state space. In particular, they simply check whether data is persisted appropriately. However, buggy data structures can have windows of vulnerability when crashes can cause failures even if all data is persisted and ordered. This motivates us to develop Jaaru, a model checker that can thoroughly explore states to find bugs.

**Model checking.** Model checking has been extensively studied. Stateless model checking techniques do not explicitly track which program states have been visited and instead focus on enumerating schedules [18–20, 40, 41]. To make model checking more efficient, researchers propose dynamic partial order reduction techniques [8, 15, 30, 45–47, 49] that exploit state equivalence to reduce search space.

Recent work model-checks multi-threaded programs against the TSO and PSO memory models [2, 23, 56] and the release-acquire fragment of C/C++ [3, 5, 14, 27].

Model checking is also widely used to find bugs in systems code. Model checkers such as EXPLODE [53], FiSC [54], and SAMC [33] check file system code. However, directly applying these techniques would dictate enumerating all possible PM states, which is not feasible given that PM is byte-addressable and has orders of magnitude more states than a disk.

Yat [29] is an attempt to model check persistent memory. It injects failures before fence operations and eagerly enumerates all post-failure states to detect potential bugs.

Agamotto [42] finds bugs in persistent memory programs by using symbolic execution. It tracks the state of persistent memory objects and their corresponding cache lines in the program, i.e., whether the cache line is modified. Agamotto updates constraints on these states as the program runs and uses them to identify different types of consistency bugs including correctness, performance, and custom user-defined bugs. It uses a priority-based static analysis to steer program execution to program states that frequently modify PM. This approach can miss bugs because it only reasons about whether stores are made persistent and does not reason about the order that stores are made persistent.

**Programming Models for PM.** There is a great deal of work on building programming systems that allow developers to use PM in a reliable way without knowing the details of PM. For example, a line of work [6, 16, 17, 34] proposes to use (software or hardware) transactions to provide (failure and thread) atomicity. Another line of work [4, 7, 22, 24, 35] advocates use of locks or synchronization-free regions [21]. Jaaru is complementary to these approaches, it can be used to check the correctness of their implementation.

### 7 CONCLUSION

Jaaru is the first efficient model checker for persistent memory programs. Jaaru uses a constraint refinement-based approach that drastically reduces the number of executions that must be explored. Jaaru is the first tool to fully model the TSO persistent memory model. Our evaluation shows that Jaaru effectively finds bugs in our benchmark applications and that Jaaru reduces the number of executions that must be explored by several orders of magnitude.

### ACKNOWLEDGMENTS

We would like to thank our shepherd, Jay Lorch, and the anonymous reviewers for their thorough and insightful comments that helped us substantially improve the paper. This work is supported by the National Science Foundation grants CNS-1703598, OAC-1740210, CNS-1703598, CNS-1763172, CCF-2006948, CNS-2007737, and CNS-2006437, as well as ONR grants N00014-16-1-2913 and N00014-18-1-2037.

### A ARTIFACT APPENDIX

#### A.1 Abstract

This artifact contains a vagrant repository that downloads and compiles the source code for Jaaru, its companion compiler pass, and benchmarks. The artifact enables users to reproduce the bugs that are found by Jaaru in PMDK (i.e., Figure 11 of the paper) and RECIPE (i.e., Figure 12) as well as the performance results to compare Jaaru with Yat (i.e., Figure 13).

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

- **Algorithm:** Lazy exhaustive model-checking
  - **Program:** Jaaru
  - **Compilation:** GCC 7.5.0 and Clang
  - **Binary:** Instrumentation LLVM pass
  - **Data set:** RECIPE and PMDK benchmarks
  - **Run-time environment:** Any system that can run Vagrant
  - **Hardware:** One 6 core 3.7 GHz Intel i7 machine with 32 GB DDR4 memory
  - **Run-time state:** Managed by our x86 simulator
  - **Execution:** Automated by our tooling system
  - **Metrics:** Crashing the program under test
  - **Output:** Program crash for bugs. Logging performance measurement for executions.
  - **Experiments:** Regenerating all bugs found by Jaaru. Reproducing performance results and comparing them with Yat (fully automated by our custom tooling)

- **How much disk space required (approximately)?** 80G
- **How much time is needed to prepare workflow (approximately)?** 1 hour
- **How much time is needed to complete experiments (approximately)?** About 20 mins
- **Publicly available?** Yes. Open-source on GitHub
  - **Code licenses (if publicly available)?** GNU GENERAL PUBLIC LICENSE Version 2
  - **Data licenses (if publicly available)?** BSD-3-Clause and Apache License 2.0.


A.3 Description

Our workflow has four primary parts: (1) creating a virtual machine and installing dependencies needed to reproduce our results, (2) downloading the source code of Jaaru and the benchmarks and building them, (3) providing the parameters corresponding to each bug to reproduce the bugs, and (4) running the benchmarks to compare Jaaru with the naive exhaustive approach (i.e., Yat). After the experiment, the corresponding output files are generated for each bug and each performance measurement.

A.3.1 How to access. All source code is open-source and available on GitHub. Our packaging requires cloning the vagrant system repository from https://github.com/uci-plrg/jaaru-vagrant. As described in the README.md file of the repository, you will need to install a VirtualBox VM and Vagrant on your machine. Then, the vagrant setup will install the required dependencies and download the source code of the tools from our git repository. Next, it builds each tool on the virtual machine.

A.3.2 Hardware dependencies. Our tooling system and Jaaru have no special hardware dependencies and it can be running on any x86 machine with at least 32GB RAM and 4 cores.

A.3.3 Software dependencies. To run our system, the following should be installed on the local machine:

- Linux (we tested on Ubuntu)
- Vagrant
- VirtualBox
- Vagrant-disksize plugin

A.3.4 Data sets. To evaluate Jaaru, our tooling system downloads the source code of RECIPE and PMDK from our git repository. We forked a branch from the original source code of these benchmarks that don’t contain our bug fixes. The tooling system automatically sets up and builds these benchmarks and runs them under Jaaru to identify bugs in them.

A.4 Installation

Please see the README.md file of the https://github.com/uci-plrg/jaaru-vagrant repository, which contains a detailed step-by-step guide to setup Jaaru on a virtual machine. Then, our scripts automatically do the following:

1. Install all the dependencies needed to install and evaluate Jaaru on different benchmarks.
2. Check out the source code for LLVM, Jaaru, Jaaru’s LLVM pass, RECIPE, and PMDK.
3. Include Jaaru’s LLVM pass to LLVM and building it
4. Set up and build Jaaru with two different configurations (One for RECIPE that uses libvmemmalloc, and one for PMDK that uses libpmem APIs).
5. Set up and building RECIPE (including CCEH, FAST, FAIR, P-ART, P-BwTree, P-CLHT, and P-Masstree benchmarks) and PMDK benchmarks.

6. Generate three scripts in the home (or ~/) directory of the virtual machine to generate the results.

Once the scripts are finished setting up the virtual machine and benchmarks, the user can use Jaaru on the virtual machine to further evaluate different benchmarks or regenerate our evaluation results.

A.5 Experiment workflow

After setting up the virtual machine, the user can use ‘vagrant ssh’ to connect to the VM and use Jaaru. The detailed instructions to run the suggested workflow is included in the README.md file of https://github.com/uci-plrg/jaaru-vagrant repository. There are three scripts in the home directory of the virtual machine that user can run:

- recipe-perf.sh: It runs the RECIPE benchmarks using Jaaru and gathers measurements to compare Jaaru against Yat. For each benchmark, the corresponding log file is generated in ~/results/recipe-performance.
- recipe-bugs.sh: It runs the RECIPE benchmarks using Jaaru and sets the corresponding parameters to reproduce each bug. For each bug, the corresponding log file is generated in ~/results/recipe-bugs.
- pmdk-bugs.sh: It runs PMDK benchmarks by using Jaaru and set the corresponding parameters to reproduce each bug. For each bug, the corresponding log file is generated in ~/results/pmdk-bugs.

In our tooling system, the timeout is used in both recipe-bugs.sh and pmdk-bugs.sh scripts to recover from segmentation fault. The timeout needs to be adjusted if the user uses a slower machine.

A.6 Evaluation and expected result

After successfully running the experiment using our scripts, the results directory is generated in the home directory. This directory contains the following results:

A.6.1 RECIPE. Performance Results: For each RECIPE benchmark, there is a -Performance file in the ~/results/recipe-performance directory (for a total of 6 files). These files contain the performance information corresponding to Figure 13.

Bugs: There are 18 files in ~/results/recipe-bugs directory. Each file contains the corresponding logs for the bug that Jaaru found. Figure 15 contains information about how Jaaru identified each bug correspond to Figure 12.

A.6.2 PMDK. There are 7 files in ~/results/pmdk-bugs directory. Each file contains the corresponding logs for the bug that Jaaru found. Figure 16 contains information about how Jaaru identified each bug correspond to Figure 11.

A.7 Experiment customization

The experiment workflow can be customized to install and run everything on the local machine instead of the virtual machine. To set up everything locally, download data/setup.sh script from the https://github.com/uci-plrg/jaaru-vagrant repository in the home directory of your local machine and run the script after installing the dependencies.
Getting stuck in an infinite loop
Segmentation fault in the program
Segmentation fault in the program
Segmentation fault in the program
Segmentation fault in the program
Segmentation fault in the program
Segmentation fault in the program
Illegal memory access in the program
Illegal memory access in the program
Illegal memory access in the program
Illegal memory access in the program
Assertion failure at pmalloc.c:270
Illegal memory access by the program
Illegal memory access by the program
Assertion failure at heap.c:533
Assertion failure at obj.c:1523
Assertion failure at pmalloc.c:270
Illegal memory access at obj.c:1528
Assertion failure at tx.c:1678

A.8 Notes
Note that the performance results generated for RECIEPE can be different from the numbers that are reported in the paper since there is non-determinism in scheduling threads; when stores, flushes, and fences leave the store buffer; and memory alignment in the malloc procedure. This non-determinism can possibly impact on the type of bugs reported in Figure 15 and Figure 16 for RECIEPE and PMDK benchmarks. Also, for some bugs, the segmentation fault (or assertion failure) occurs in Jaaru code. This is caused by illegal memory access by the program under test.

REFERENCES