Yat: A Validation Framework for Persistent Memory Software

Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson, *Intel Labs*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Yat: A Validation Framework for Persistent Memory Software

Philip Lantz   Subramanya Dulloor   Sanjay Kumar   Rajesh Sankaran
Jeff Jackson

Intel Labs

Abstract

This paper describes the design and implementation of *Yat*. Yat is a hypervisor-based framework that supports testing of applications that use Persistent Memory (*PM*)—byte-addressable, non-volatile memory attached directly to the memory controller. PM has implications on both system architecture and software. The PM architecture extends the memory ordering model to add software-visible support for *durability* of stores to PM. By simulating the characteristics of PM, and integrating an application-specific checker in the framework, Yat enables validation, correctness testing, and debugging of PM software in the presence of power failures and crashes. We discuss the use of Yat in development and testing of the Persistent Memory File System (*PMFS*), describing the effectiveness of Yat in catching and debugging several hard-to-find bugs in PMFS.

1 Introduction

We are witnessing growing interest in Non-Volatile DIMMs (NVDIMMs) that attach storage class memory (e.g., *PCM*, *MRAM*, etc.) directly to the memory controller [5]. We refer to any such byte-addressable, non-volatile memory as Persistent Memory (*PM*).

PM has implications on system architecture and software [2]. Since PM performance—both latency and bandwidth—is within an order of magnitude of DRAM, software can map PM as write-back cacheable for performance reasons. Several studies [2, 6] have shown significant performance gains from the use of in-place data structures in write-back PM. These studies also show the need for extensions to the existing memory model to allow PM software to control *ordering* and *durability* of stores to write-back PM.

However, such extensions to the memory model introduce the possibility of new types of programming errors. For instance, consider a PM software flow as shown in Figure 1. Starting at consistent state A, PM software performs two writes to PM (set $W_{A->B}$), dirtying two cachelines (L_1, L_2) in the process. For traditional block based storage, software explicitly schedules IO to make these cachelines persistent at block granularity. However, for PM-based storage these cachelines can become persistent in arbitrary order by cacheline evictions outside of software control. Hence, extra care must be taken in enforcing ordering on updates to PM. Programmers today are not used to explicitly tracking and flushing modified cachelines in volatile memory. But, this is a critical requirement of PM software, failing which could cause serious consistency bugs and data corruption.

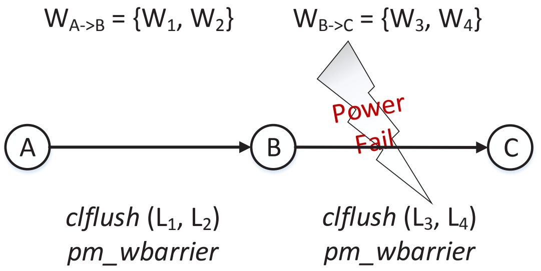

Figure 1: PM software flow

Testing for the correctness of PM software is challenging. One way is to simulate or induce failures (such as from power loss or crashes) and use an application-specific checker tool (similar to fsck for Linux filesystems) to verify consistency of the data in PM. However, this method tests only the actual ordering of writes to PM in a single flow, even though many other orderings are possible outside the control of the PM software (e.g., due to arbitrary cacheline evictions).

To overcome this challenge, we built *Yat* (meaning “trial by fire” in Sanskrit), a hypervisor-based framework for testing the correctness of PM. Yat uses a record and replay method to simulate architectural failure conditions that are specific to PM. We used Yat in validation and correctness testing of PMFS [2], which is a reasonably large and complex PM software module. Though we focus on PMFS as the only case study in this paper, the principles of Yat are applicable to other PM software, including libraries and applications [6].

Contributions of this paper are as follows:

- Design and implementation of Yat, a hypervisor-based framework for testing PM software.
- Evaluation of a Yat prototype, using PMFS as an example PM software.

2 System Architecture

We assume the Intel64 based system architecture described elsewhere [2], where software can access PM directly using regular CPU load and store instructions. Because PM is typically mapped write-back (for performance reasons), PM data in CPU caches can be evicted and made durable at any time. To give software the ability to control consistency, the architecture must provide a software visible guarantee of ordering and durability of stores to PM. The proposed architecture includes a simple hardware primitive, PM write barrier (*pm_wbarrier*), which guarantees durability of all stores to PM that have been explicitly flushed from the CPU caches but might still be be in a volatile buffer in the memory controller or in the PM module.

In Figure 1, to effect $W_{A->B}$ before $W_{B->C}$, PM software must flush the dirty cachelines L_{1-2} and make those writes durable using *pm_wbarrier*, before proceeding to writes in $W_{B->C}$. In complex software, it can be challenging to keep track of all the dirty cachelines that need to be flushed before the use of *pm_wbarrier*. We expect user-level libraries and programming models to hide most of this programming complexity from PM applications [6].

Yat is designed to help validate that PM software correctly uses cache flushes (*clflush*), ordering instructions (*sfence* and *mfence*), and the new hardware primitive (*pm_wbarrier*) to control durability and consistency in PM, even in the face of arbitrary failures and cacheline evictions. For any sequence of updates to PM, Yat creates all possible states in PM based storage and then runs the PM application's recovery tool to test recovery to a consistent state. The possible orderings are determined by the memory ordering model of the processor architecture. The proposed system (based on Intel64 architecture) follows these rules:

1. When a write is executed, it may become durable immediately or at any subsequent time up to the point where it is known to have become durable by rule 5.
2. When a write is executed that modifies a cache line that has been modified by a prior write, the later write is guaranteed to become durable no sooner than the prior write to the same cache line. This guarantee holds across cores based on Intel64 architecture.
3. When a *clflush* is executed, it has no effect until it is followed by a fence on the same processor.
4. When a fence is executed, prior clflushes on that processor take effect. All writes performed by any processor to the cache lines affected by the clflushes are flushed.
5. When a *pm_wbarrier* primitive is executed, all writes that have been flushed by rule 4 are made durable. Any writes that have not been clflushed—or that were clflushed but where no subsequent fence was executed on the same processor as the *clflush* prior to the *pm_wbarrier*—are not guaranteed to be durable.

3 Yat Design

Yat is a framework for testing PM software. We refer to the PM software being tested as *App*.

The goals of Yat are:

1) to test App for bugs caused by improper reordering of write operations; e.g., due to missing or misplaced ordering and durability instructions.

2) to exercise the PM recovery code in App in the context of a large variety of failure scenarios, such as power failures and software failures internal or external to App that cause it to abort.

To test that App applies sufficient memory ordering constraints to preserve consistency no matter when a failure occurs, Yat simulates reordering PM writes in every allowed order in which they may become durable in the PM hardware based on the rules in §2. Note that if App is multithreaded, Yat records the actual sequence of operations executed by the various threads. However, Yat does not model the non-determinism in the software as it has no knowledge of synchronization done by the software.

The operation of Yat is shown in Figure 2. Yat operates in two phases. The first phase, *Yat-record*, simply collects a trace (*App-trace*) while App is executing. Yat-record logs write and *clflush* instructions within the address range of PM, along with the explicit ordering instructions (sfence and mfence), and the new hardware primitive *pm_wbarrier*.

The second phase, *Yat-replay*, has the following steps:

Segment Yat-replay divides App-trace into segments, separated by each *pm_wbarrier* in the trace. For each segment, there is a set of writes to PM that have been executed but are not guaranteed to be durable. This set of writes is called the *active set* for that segment.

Reorder For each segment, Yat-replay selects subsets of writes in the active set that do not violate rule 2. Each such selected subset of writes is called a *combination*.

Replay For each combination, Yat-replay starts with an initial state (*App-initial-state*), applies all writes that are guaranteed to have become durable as of the *pm_wbarrier* that precedes the current segment, and then applies the writes contained in the current combination.

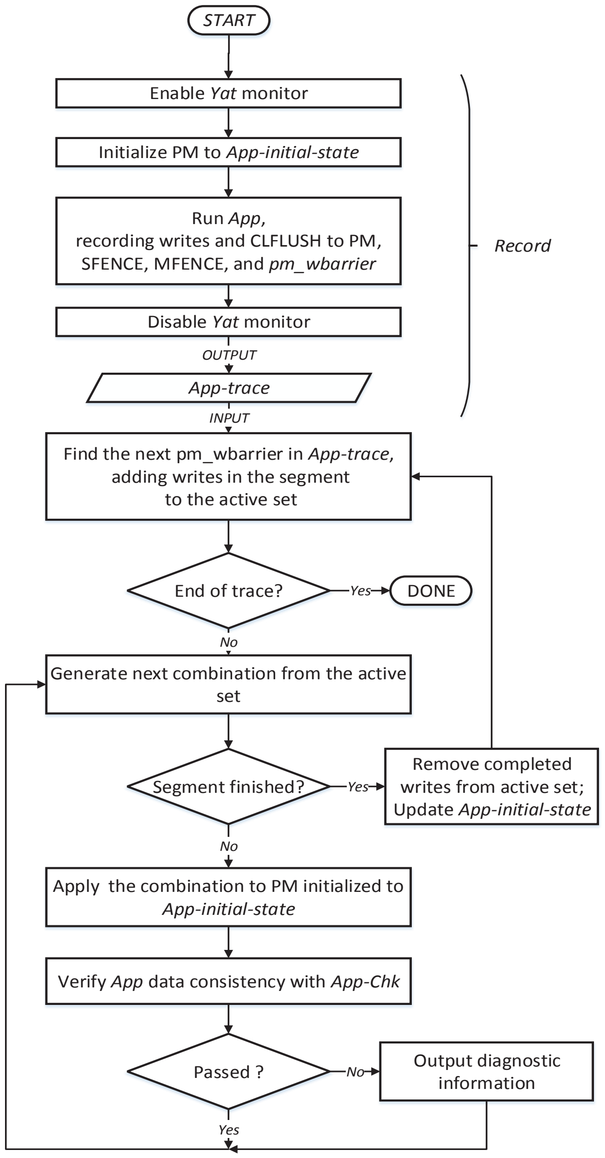
START
Enable *Yat* monitor
Initialize PM to *App-initial-state*
Run *App*, recording writes and CLFLUSH to PM, SFENCE, MFENCE, and *pm_wbarrier*
Record
Disable *Yat* monitor
OUTPUT
App-trace
INPUT
Find the next pm_wbarrier in *App-trace*, adding writes in the segment to the active set
End of trace?
Yes
DONE
No
Generate next combination from the active set
Segment finished?
Yes
Remove completed writes from active set; Update *App-initial-state*
No
Apply the combination to PM initialized to *App-initial-state*
Verify *App* data consistency with *App-Chk*
Passed ?
No
Output diagnostic information
Yes

Figure 2: Yat flow diagram

Recover Yat-replay then runs App to restore the PM state from the point of the simulated failure to a consistent state. In the case of PMFS, this step consists of simply mounting the file system. For other applications, it may involve running an application-specific recovery routine.

Verify Yat-replay then runs an application-specific data integrity checker (*App-chk*), such as those used with file systems (fsck) and databases, to verify consistency of App's persistent data. If the check fails, Yat-replay reports the point in the trace where the failure occurred, along with the combination of uncommited writes that were applied.

The number of combinations grows exponentially with the number of writes in the active set, and could become prohibitively large. However, we found this issue to be less of a problem than one might fear. Because of rule 2, the number of cache lines modified by writes in the active set is more significant than the total number of writes. For instance, in PMFS, the number of cache lines written per segment is typically between four and six, so the number of combinations is typically manageable.

Before generating any combinations for a segment, Yat can compute the maximum number of combinations for the segment based on the number of write operations and affected cache lines in the active set. If this number is below a configurable threshold, Yat exhaustively tests every combination; otherwise, it chooses combinations at random. The decision on whether to do random or exhaustive testing is made separately for each segment of the trace. The threshold for combinations per segment must take into account both Yat performance and desired test coverage for App. For testing PMFS, we used a threshold of 250 combinations per segment.

4 Yat Implementation

Although the design of Yat is independent of the software being tested, some of the implementation choices were made with PMFS in mind. PMFS is a POSIX compliant file system, designed and optimized for PM and our system architecture. PMFS maintains both meta-data and data as a B-tree in PM and uses a combination of *atomic in-place updates* and *fine-grained logging* to ensure consistent updates to the meta-data. PMFS is a reasonably complex source base, with about 10K lines of code—ideal for a realistic evaluation of Yat.

Since PMFS runs in kernel mode, we needed a layer of software below the kernel to trap accesses to PM. Yat-record is implemented using our internal Type-I research hypervisor (similar to Xen [1]), called Hypersim. Like other VMMs, Hypersim uses *Intel64 VT-x* [4] to place itself between the hardware and the guest. Unlike a VMM, Hypersim does not actually virtualize the platform, but only provides a way to observe the behavior of a single guest. Hypersim uses typical VMM memory-management capabilities, such as Extended Page Tables [4], to cause VM exits on write accesses to PM, and then logs these events. Hypersim also intercepts *clflush* and fence instructions and *pm_wbarrier* hardware primitive and logs these events. Because these instructions normally do not cause VM exits, the current implementation requires App to be recompiled, substituting illegal instructions for the *clflush* and fence instructions and the *pm_wbarrier* hardware primitive. Hypersim intercepts these illegal instruction faults and logs them. This modification is not difficult in practice because often these instructions and primitives are implemented as macros or inline functions, requiring very few changes to App. Ideally, however, Yat-record would accomplish this without modifying App's source code. We intend to explore this capability in future work, perhaps using binary translation techniques [3].

Yat-replay is implemented jointly between a shell script and Hypersim. The shell script controls the steps of replay operation, while Hypersim manages App's PM state and maintains the current segment and combination being replayed.

When starting Yat-replay, the shell script loads App-initial-state and App-trace into Hypersim's memory. For each test, the shell script calls Hypersim to apply the next combination of writes to the PM. Hypersim applies the writes in the current combination using Copy-on-Write (CoW). The shell script then invokes App to perform recovery from the simulated failure, and App-chk to verify App's data consistency following the recovery.

Either App recovery code or App-chk may report a failure or may, in fact, hang, crash, or abort, depending on the nature of the problem. Yat-replay reports any of these types of failures, along with the segment of the trace that is being replayed, and the combination.

Since Hypersim uses CoW both for writing the combination and during recovery, it can easily restore PM to App-initial-state after each test, before applying the next combination.

After Yat-replay is done applying the last combination of a segment, it automatically advances to the next segment. At the transition from each segment to the next, Yat-replay removes from the active set any writes that are known to have been made durable by the *pm_wbarrier* separating the two segments. To do this, Yat-replay modifies the PM state by applying these writes without using CoW, so they are made permanent for subsequent tests, thereby advancing App-initial-state. Yat then adds to the active set the writes in the new segment.

Yat-record has the capability of recording additional information in the trace, in the form of *annotations*, to aid in diagnosing failures. In testing PMFS, for instance, we used annotations to indicate what shell command was being executed, and, within PMFS, what transaction type was being performed. When Yat-replay reported a failure, these annotations allowed us to quickly determine what part of the original test was being replayed and what part of the PMFS code was involved. In a trace with hundreds of thousands of entries, it would have been very difficult to diagnose failures without such annotations. Examining the traces (with annotations) collected by Yat-record gave significant insight into the operation of PMFS code, and in general was very helpful in debugging and fixing several bugs.

Figure 3 shows two segments of a sample trace. The columns are as follows:

id an identifier for the trace entry

entry an indication of whether the trace entry is a write, *clflush*, *fence*, or *pm_wbarrier*

id	entry	offset	bytes	data	proc
1	W	401182	6	$<$ 6 bytes $>$	
2	W	4011c0	12	$<$ 12 bytes $>$	
3	W	201184	20	$<$ 20 bytes $>$	
4	W	4011a0	16	$<$ 16 bytes $>$	
5	W	4011cc	12	$<$ 12 bytes $>$	
6	W	001080	12	$<$ 12 bytes $>$	
7	W	00108f	49	$<$ 49 bytes $>$	
8	W	00108e	1	$<$ 1 byte $>$	
9	W	00108c	2	$<$ 2 bytes $>$	
10	C	001080			1
11	F				1
12	P				
13	W	0010c0	12	$<$ 12 bytes $>$	
14	W	0010ce	1	$<$ 1 byte $>$	
15	W	0010cc	2	$<$ 2 bytes $>$	
16	C	0010c0			1
17	F				1
18	P				

Figure 3: Excerpt of App-trace

offset the offset in the PM of the write or *clflush* operation

bytes the number of bytes written

data the data that was written

proc an indication of which processor executed a *clflush* or *fence*

For the purposes of explanation, assume that the *pm_wbarrier* preceding this excerpt made all outstanding writes durable, so the active set is empty at the beginning.

First, the 9 writes at lines 1 – 9 are added to the active set. These writes modify 3 cache lines, 401180, 4011c0, and, 001080. There are 3 writes to the first cache line, 2 writes to the second, and 4 to the third. A total of 59 combinations are generated for this segment: $(3+1) \times (2+1) \times (4+1) - 1$. After each of these 59 combinations is tested, Yat-replay processes the *pm_wbarrier* at line 12. The writes to cache line 001080 are made durable by this *pm_wbarrier*, because of the *clflush* at line 10, which was fenced at line 11. So these writes (lines 6 – 9) are removed from the active set. The other writes (lines 1 – 5) are retained in the active set. Yat-replay then proceeds to the next segment, and adds the 3 writes at lines 13 – 15 to the active set. The active set for this segment contains 8 writes to 3 cache lines, and the number of combinations is 47: $(3+1) \times (2+1) \times (3+1) - 1$.

Optimizations As mentioned before, the number of combinations in the Reorder phase grows exponentially with the number of writes, and can easily become very large. While a configurable threshold for the number of combinations provides a reasonable (probabilistic) compromise, Yat-replay performance is very important for good coverage. We optimized Yat-replay in several ways.

Even though App may be multi-threaded, the replay/test cycle is single-threaded. To increase the amount of testing that can be completed, we run multiple instances of Yat-replay in parallel. Each replay instance has a separate copy of the PM state. All the instances of

Test	writes	clflushes	pm_wbarriers	Segments		Combinations		Time	
				Total	Threshold	Total	Threshold	Total	Threshold
T1	506	372	131	131	12	15K	4K	55m	15m
T2	54K	14K	6K	6K	4K	789M	1M	5.2y	3d
T3	158K	53K	15K	14K	6K	3.7×10^{78}	2M	∞	5d

Table 1: Evaluation of PMFS test coverage with Yat

replay work from a single trace, but each is assigned a different part of the trace to work on. This results in a speed-up factor nearly equal to the number of hardware threads available.

In many places in App, a large number of consecutive writes are performed that have no temporal dependency. For example, when PMFS uses strcpy to fill in a file name, the order of the write of each character of the file name is immaterial. It would be useless (and impossible) for Yat to attempt to test every possible ordering in such a situation. To avoid this, Yat-record detects adjacent writes to a single cache line and coalesces them into a single write entry in App-trace. This also makes the trace more compact. A single write operation in the trace can be up to a full cache line. Note that this optimization can reduce the effectiveness of testing, because it neglects to test some reorderings that might be significant. An analysis of App might help determine whether it contains sequences of writes where this optimization would be unacceptable. Additional heuristics could be applied to avoid doing this in such cases.

App optimizations App is executed during the replay phase only to perform recovery. The normal, optimized code paths in App are not used at all. Under normal conditions, App recovery code would run rarely, if ever, so it would not be a target for optimizations. However, Yat runs App recovery code millions of times, so its performance has a significant effect on the amount of testing that can be completed. We found it worthwhile to track down and eliminate some bottlenecks in App recovery code, such as use of global locks.

5 Evaluation

In this section, we describe our experiences using Yat for validating meta-data consistency in PMFS. PMFS recovery is built into PMFS itself; it attempts to recover on the next mount after an unclean unmount. We implemented a separate consistency checker tool (fsck.pmfs) to check for PMFS meta-data consistency after recovery.

Examples of bugs found in PMFS Yat helped us find several bugs in PMFS journaling and recovering, and was instrumental in testing the correctness of PMFS. The earliest bugs we found using Yat were coding errors in the recovery code. We were able to find these bugs due to Yat's ability to exercise App recovery code in scenarios that are otherwise hard to create.

The second common type of bug that was easily detected by Yat was a failure to log a particular modification to the file system. In PMFS, creating a log entry causes PMFS to track dirty cache lines and later perform a *clflush* of the relevent cache line. If the log call is omitted, the *clflush* is not done, which Yat easily detects. In one case, the fields that were not logged were the access and modification times of a file; an incorrectly updated value in those fields would be difficult to detect in any other way, because the outdated value is within the expected range for the field.

A more complex example of a bug detected by Yat was a case where two inodes were being deleted on two separate threads. PMFS uses a "truncate list" to free the blocks used by a file when recovering from a failure that occurs during deletion. If a failure occurs after the truncate-list transaction is committed, recovery will process the truncate list and perform the steps to truncate the file. Essentially, the truncate list acts as a redo journal.

In the failure scenario, thread 1 starts transaction 1 and deletes inode 1, adding it to the truncate list. Before transaction 1 is committed, thread 2 starts transaction 2 and deletes inode 2, adding it to the truncate list. Transaction 2 completes and is committed, and then a Yat-simulated failure occurs.

During recovery, uncommitted transaction 1 is reverted, including removing inode 1 from the truncate list. However, because the truncate list is a linked list, removing inode 1 from the list also removes inode 2 from the list, so the blocks that had been owned by inode 2 are never freed.

This bug was detected by fsck because the link pointer of inode 2 was not cleared, as it would have been if inode 2 had been on the truncate list to be processed by the recovery code. This link field is only cleared to allow detection of this sort of problem. The cause of the bug was that the truncate list was unlocked before the containing transaction was committed, allowing another thread to modify the list. The solution was to commit the transaction before unlocking the list.

This solution led to another bug, also detected by Yat. If a failure occurs during recovery, some of the steps to truncate the inode and free its blocks may be completed, but the inode is still on the truncate list. During the next attempt at recovery, the inode is in an unexpected state and recovery fails. This bug was detected by Yat in multiple ways, depending on the point of failure during recovery. In one case, the recovery code failed due to a NULL pointer access; in another case, fsck found that the size of the inode was incorrect after recovery. To solve this, the code that processes the truncate list during recovery must not assume anything about the state of the inode and must be idempotent.

Yat performance For the performance evaluation of Yat, we tested PMFS with Yat on a system with 3GHz Intel third generation Core i5 processor and 8GB memory. The processor has 4 cores and 8 threads. The system is running a Linux 3.3 kernel (with PMFS), booted as the Hypersim guest OS.

Table 1 shows sample Yat performance when validating PMFS. The tests were performed using a threshold of 250 for the maximum number of combinations per segment. The Segments-Threshold column is the number of segments where the number of combinations would have exceeded 250. The Combinations-Total column is the total number of combinations for the test if the number of combinations for each segment were unlimited. The Time columns are the approximate times required to run the number of combinations in the Combinations columns. T1 runs 100 simple shell commands to make directory, create a file, and append small amounts of text data to the file. T2 runs 1200 commands in the same mix as above. T3 runs 75 (more complex) commands that copy large files to PMFS and tar the contents.

Replay performance is highly dependent on App and App-chk, as most of the time is spent in the recovery and checking steps. We observed that 5% of the total replay time was spent setting up each test, 65% in App recovery, and 30% in App-chk. Since Yat performance scales almost linearly with available compute power, we can further improve validation time by using more capable (even distributed) systems.

We were able to both detect and diagnose bugs by examining the trace manually. We believe that we can automate some of this. In other words, we can detect some bugs automatically simply by using some fairly simple heuristics to analyze the trace, without running the replay at all. One heuristic that is completely independent of the design of App is that if a cache line has been written to but not clflushed, and thus remains in the active set across a number of segments (10 segments, perhaps), it is likely that the *clflush* was omitted.

6 Related Work

Providing consistency and recovery in the face of failures is challenging, therefore necessitates extensive testing. In its goals, Yat is similar to previous efforts that used a combination of model-based analysis, automation, and knowledge of the application to achieve good test coverage in reasonable time [7].

The possible failure modes for storage software (such as file systems and databases) depends on the characteristics of the underlying storage medium and its interface to the rest of the system. Prior work focused on testing software built for block-based devices with a separate address space [7]. In contrast, PM is a byte-addressable storage medium that resides in the same address space as regular volatile memory. Though the challenge posed by evictions and reordering is similar for PM and block based systems, implementing a permutation-based testing framework for PM requires different techniques and optimizations. To the best of our knowledge, this paper is the first to define and address the challenges to testing software posed by PM.

7 Conclusion and Future Work

Yat, a generic PM software validation framework, has several key characteristics that make it effective for testing and debugging PM software. After capturing a sequences of writes and fence operations by App, Yat tests all permissible orderings of these operations, resulting in extensive coverage of possible error conditions in App. When it detects a failure, Yat reports the exact sequence of operations that led up to the failure, aiding in the diagnosis of the failure. Furthermore, Yat is fast enough that it is practical for use in testing real-world software, as demonstrated in the testing of PMFS.

We are looking at ways to avoid having to modify the PM application source code to cause VM exits for *clflush* and fence instructions and *pm_wbarrier* primitive. Binary translation tools potentially can address both these goals. We are also planning to port Yat from a VMM environment to a OS native environment to debug application level PM code without needing a hypervisor. This should greatly simplify the setup overhead for Yat.

References

[1] BARHAM, PAUL AND DRAGOVIC, BORIS AND FRASER, KEIR AND HAND, STEVEN AND HARRIS, TIM AND HO, ALEX AND NEUGEBAUER, ROLF AND PRATT, IAN AND WARFIELD, ANDREW. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, ACM.

[2] DULLOOR, SUBRAMANYA R. AND KUMAR, SANJAY AND KESHAVAMURTHY, ANIL AND LANTZ, PHILIP AND REDDY, DHEERAJ AND SANKARAN, RAJESH AND JACKSON, JEFF. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, ACM.

[3] INTEL. Pin - A Dynamic Binary Instrumentation Tool, 2012.

[4] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual, 2013.

[5] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. In *In International Symposium on Computer Architecture (ISCA '09).*

[6] VOLOS, HARIS AND TACK, ANDRES JAAN AND SWIFT, MICHAEL M. Mnemosyne: lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, ACM.

[7] YANG, JUNFENG AND TWOHEY, PAUL AND ENGLER, DAWSON AND MUSUVATHI, MADANLAL. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst. 24*, 4 (Nov. 2006).