On Testing Persistent-Memory-Based Software

Ismail Oukid
TU Dresden & SAP SE
i.oukid@sap.com

Daniel Booss
SAP SE
daniel.booss@sap.com

Adrien Lespinasse*
Grenoble INP-Ensimag
adrien.lespinasse@ensimag.fr

Wolfgang Lehner
TU Dresden
wolfgang.lehner@tu-dresden.de

ABSTRACT

Leveraging Storage Class Memory (SCM) as a universal memory–i.e. as memory and storage at the same time–has deep implications on database architectures. It becomes possible to store a single copy of the data in SCM and directly operate on it at a fine granularity. However, exposing the whole database with direct access to the application dramatically increases the risk of data corruption. In this paper we propose a lightweight on-line testing framework that helps find and debug SCM-related errors that can occur upon software or power failures. Our testing framework simulates failures in critical code paths and achieves fast code coverage by leveraging call stack information to limit duplicate testing. It also partially covers the errors that might arise as a result of reordered memory operations. We show through an experimental evaluation that our testing framework is fast enough to be used with large software systems and discuss its use during the development of our in-house persistent SCM allocator.

1. INTRODUCTION

The advent of Storage Class Memory (SCM) is disrupting the database landscape and driving novel database architectures that store data, access it, and modify it directly from SCM at a cache-line granularity [3, 7, 9]. However, the *no free lunch* folklore conjecture holds more than ever as SCM brings unprecedented challenges. Consistency failure scenarios and recovery strategies of software that persists data depend on the underlying storage technology. In the traditional case of block-based devices, software has full control over when data is made persistent. Basically, software schedules I/O to persist modified data at a page granularity. The user level has no direct access to the primary copy of the data and can only access copies of the data that are buffered in main memory. Hence, software errors can corrupt data only in main memory which can be reverted as long as the corruption was not explicitly propagated to storage. In fact, crash-safety for block-based software highly depends on the correctness of the underlying file system. In contrast, SCM is byte-addressable and is accessed via a long volatility chain that includes store buffers, CPU caches, and the memory controller buffers, over all of which software has little control. The SNIA [2] recommends to manage SCM using an SCM-aware file system that grants the application layer direct access to SCM with *mmap*, enabling load/store semantics. As a side effect, changes can be speculatively propagated from the CPU cache to SCM at any time, and compilers and out-of-order CPU execution can jeopardize consistency by reordering memory operations. Moreover, changes are made persistent at a cache line granularity which necessitates the use of CPU persistence primitives. This adds another level of complexity as enforcing the order in which changes are made persistent cannot be delayed like with block-based devices, and must be synchronous. In addition to data consistency, memory leaks in SCM have a deeper impact than in DRAM. This is because SCM allocations are persistent, hence, a memory leak would also be persistent.

Several proposals tackled these challenges following two main approaches. The first one focuses on providing global software-based solutions, mainly transactional-memory-like libraries, to make it easier for developers to write SCM-based software. Examples of these solutions include Mnemosyne [12], NVHeap [6], and REWIND [4]. The second and more mainstream approach is to rely solely on existing hardware persistence primitives, such as cache line flushing instructions and memory barriers to achieve consistency. Several persistent data structures were proposed following this approach, such as the CDDS B-Tree [11], the wBTree [5] and the NV-Tree [14]. Nevertheless, all approaches have in common that SCM-related errors may result in data corruption. In contrast to volatile RAM where data corruption can be cured with a restart of the program, data corruption in SCM might be irreversible as it is persistent. Therefore, we argue for the need of testing the correctness of SCM-based software against software crashes and power failures–which result in the loss of the content of the CPU cache. We remark that testing is orthogonal to devising recovery strategies that solve the challenges introduced by SCM.

In this paper we propose a lightweight automated on-line testing framework that helps detect and debug a wide range of SCM-related bugs that can arise upon software or power failures. We particularly focus on detecting missing cache line flushing instructions. Our testing framework is based on a *suspend-test-resume* approach and is able to simulate different crash scenarios including the loss of the content of the CPU cache. An important feature of our testing framework is its ability to avoid excessive duplicate testing by tracking the call stack information of already tested code paths, which leads to achieving fast code coverage. Additionally, our testing framework is able to detect errors that might arise due to the compiler or the CPU speculatively reordering memory operations. An additional capability of our testing framework is simulating crashes in the recovery procedure of the tested program, which we argue is very

*This author contributed to this work while interning at SAP SE.

DaMoN'16, June 27, 2016, San Francisco, CA, USA

ISBN 978-1-4503-4319-0/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2933349.2933354

important since hidden SCM-related errors in the recovery procedure may compromise the integrity of the data upon every restart.

We show with an experimental evaluation on a persistent B-Tree and a persistent SCM allocator that our testing framework exhibits good performance, even in the case of nested crash simulations, and is fast enough to be used on fairly large software systems. In particular, we demonstrate that taking into account the call stack can improve the testing time by several orders of magnitude.

This paper is organized as follows: Section 2 discusses related work and elaborates on our system architecture assumptions. Then, Section 3 presents our testing framework and its different optimizations. Thereafter, we evaluate our testing framework in Section 4. Finally, Section 5 concludes this paper and outlines future directions for a field, we believe, much in need of research.

2. BACKGROUND AND RELATED WORK

Crash-safety for disk-based software has been extensively researched and several tools that combine experimental testing and model checking have been proposed [10, 13]. Although they share the same goals, crash-safety testing for disk-based and SCM-based software are different in that they have to address radically different failure modes. Consistency and recovery testing of SCM-based software did not get much attention so far. Lantz et al. [8] proposed Yat, a hypervisor-based off-line testing framework for SCM-based software. Yat is based on a record-and-replay approach. First, it records all SCM write operations by logging VMM exits that are caused by writes to SCM. Persistence primitive instructions need to be replaced in tested software by illegal instructions to make them traceable by causing a VMM exit. Yat then divides the memory trace into *segments*, each of which is delimited by two persistence barriers. It considers that SCM write operations can be arbitrarily reordered within a segment, with the exception of writes to the same cache line which are considered to be of fixed order. Yat replays the trace until a non-tested segment is encountered, then runs the recovery procedure of the tested software for every possible reordering combination inside that segment. Since the number of combinations can grow exponentially, the authors propose to limit the number of combinations per segment to a certain threshold.

Given a program that covers the whole code base of the tested software, and given a sanity check program that detects any present data corruption–both of which are challenging to produce–, Yat may theoretically achieve comprehensive testing for single-threaded SCM-based software. In practice however, that may still require prohibitive testing time. In the case of a multi-threaded program, Yat records the sequence of operations executed by the various threads, which can differ between two runs due to the non-determinism of multi-threading. Hence, comprehensive testing of a recorded sequence does not imply comprehensive testing of the software.

In contrast to Yat, our testing framework performs on-line testing and is non-invasive as it does not require software changes in most cases. While it covers only partially memory-reordering-related errors, our testing framework achieves fast code coverage by limiting duplicate testing, and is able to automate crash testing inside the recovery procedure of a program, both of which Yat does not provide. We explain in Section 3.7 how our testing framework can be combined with Yat to make up for the limitations of both tools.

We conjecture that providing comprehensive consistency and recovery testing for large SCM-based software is practically not feasible. Instead, our focus is on improving the *quality* of such software by covering a wide range of SCM-related errors in a reasonable amount of time. We argue that similarly to concurrent software, providing theoretical correctness guarantees should be a prerequisite for any SCM-based software, and that experimental testing should not (and cannot) make up for the lack of such theoretical guarantees.

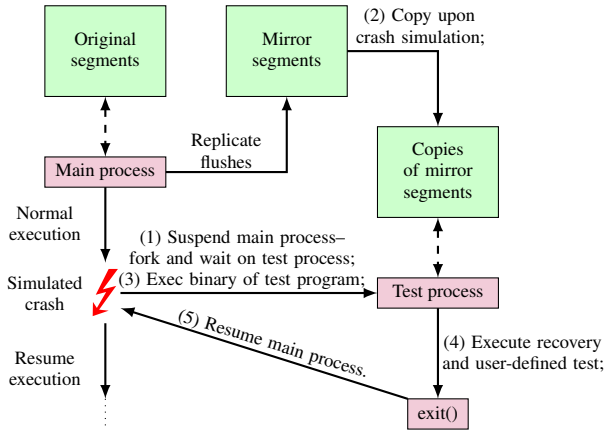


Figure 1: Illustration of crash simulation in the testing framework.

In this work we assume, without loss of generality, an Intel x86 architecture that provides the following persistence primitives:

- *mfence*: It is a memory barrier that guarantees that all load and store instructions finish executing before proceeding further.
- *clflush*: It is an instruction that evicts a cache line from the CPU cache and writes it back to memory. *clflush* is asynchronous and is not guaranteed to take place until an *mfence* is issued [1].

clflush needs to be wrapped by two memory barriers to be fully ordered and serialized. In the following we refer to a serialized *clflush* by the *persist* function. Other persistence instructions, such as *clflushopt*, *clwb*, and *pcommit*, have been announced by Intel. However, since there is currently no available hardware that supports them, we do not consider them in this work. Nevertheless, the flexibility of our testing framework makes it easy to take them into account in the modeling of SCM persistence in the future.

3. TESTING OF SCM-BASED SOFTWARE

We propose a lightweight on-line testing framework that is able to simulate software crashes and power failures that cover a wide range of consistency and recovery bugs. As an implementation example, we integrate the framework with our own persistent SCM allocator. The persistent allocator creates large files, referred to as *segments*, and then logically divides them into smaller blocks for allocation.

3.1 Crash simulation

In this section we explain how our testing framework is able to simulate software and power failures. The main challenge in simulating a power failure is to simulate the loss of the content of the CPU caches. To achieve this, we devise the following strategy that is based on a *suspend-test-resume* approach:

- For every segment created by the persistent allocator, a corresponding *mirror segment* is created.
- When a cache line is explicitly flushed, its content is copied from the original segment to the same offset in its corresponding mirror segment. Therefore, only explicitly flushed data is present in the mirror segments.
- *Malfunctions* that simulate a crash are randomly triggered when calling persistent primitive functions. These simulated crashes proceed as follows, as illustrated in Figure 1: (1) Suspend (pause) the main process by forking a test process and waiting on it; (2) the test process creates a copy of each mirror segment; and (3) the test process executes the binary of the test program which recovers

using the created copies of the mirror segments. The test process starts then the recovery procedure the same way as would have done the main process if it crashed where it was suspended. After recovery, the test process executes a user-defined consistency check procedure (step (4) in Figure 1).

- Once the test process exits, the copies of the mirror segments are deleted and the main program resumes execution (step (5) in Figure 1) until the next crash simulation.

This strategy is fully *automated* and the state of the main process is never changed during the crash simulation phase. More specifically, the state of the original segments and the mirror segments is never changed during the crash simulation phase. Capturing explicit flushes and replicating them in the mirror segments allows to simulate the loss of the CPU cache. In fact, this corresponds to the case where no cache lines are evicted from the CPU cache without being explicitly flushed. In a real scenario, some of the content of the CPU cache might have been speculatively written back. Nevertheless, we argue that by capturing only explicitly flushed data, we can more reliably detect missing flushes in the tested code.

Crash simulations are triggered inside the *Flush* function, as shown in Algorithm 1, because it is the only function that modifies the state of the mirror segments. In the case of a detected error or a crash in the recovery procedure or in the user-defined checking procedure, the test process is suspended and a debugger can be attached to both the main process and the test process. This greatly helps in finding the reason of the crash, since both the simulated crash scenario in the main process and the recovery crash in the test process can be fully traced and the content of their corresponding data structures examined. To help reproduce errors, a seeded random number generator is used to generate crash probabilities. Hence, in single-threaded execution, a bug is always reproducible as long as the same seed is used. Besides, it is possible to allow crash simulations for only a specific code region.

Algorithm 1 Random crash simulation

```
1: procedure FLUSH(PPtr PAddr, Char Content[CacheLineSize])
2:     crashProb ← rand(0, 1)   ▷ Get a random crash probability
3:     if crashProb < 0.5 then   ▷ Crash with a probability of 0.5
4:         SimulateCrash()
5:     copyToMirror(PAddr, Content)
6:     crashProb ← rand(0, 1)
7:     if crashProb < 0.5 then
8:         SimulateCrash()
```

To illustrate different classes of errors that our testing framework can detect, we consider in the following the case of a simplified array append operation. The correct code is:

```
1 array[size] = val;
2 persist(&array[size]);
3 size++;
4 persist(&size);
```

The newly appended value must be persisted *before* $size$ is incremented and persisted. Consider the following code:

```
1 array[size] = val;
2 size++;
3 persist(&size);
```

In this case there is no guarantee that $size$ will refer to only valid entries in the array after a failure, because appended values are never explicitly persisted. Our testing framework successfully detects such errors since $array$ will never be updated in the mirror segments and a simple check of the content of $array$ during a simulated crash will detect this issue.

3.2 Faster testing with copy-on-write

In practice, the cost of making copies of the mirror segments for every simulated crash is proportional to the size and number of segments. Therefore, this step can be prohibitively expensive for programs with a large memory footprint. To remedy this issue, we use *copy-on-write memory mapping*[1]. Copy-on-write enables to read data directly from the mirror segments while copying only the memory pages that are modified by the test process. When the test program terminates, the memory pages that hold the changes that were made to the mirror segments are discarded. Hence, both the original segments and the mirror segments remain unchanged during the crash simulation phase, but at a much lower cost than that of making copies of the mirror segments.

3.3 Towards faster code coverage

Triggering crash simulations purely randomly gives no code coverage guarantees as some critical paths might not be tested while others might be tested multiple times. An alternative to the purely random approach is systematic crash simulation, similar to the one followed by Yat. This approach has the disadvantage of testing the same critical path as many times as it is executed. For instance, if a program appends one thousand values to a persistent array, systematic crash simulation will test the append function of the persistent array one thousand times, leading to prohibitive testing times.

Algorithm 2 Trigger crash based on the probability of a call stack

```
1: procedure GETCRASHPROBABILITY(CallStack S)
2:     (prob, found) ← CallStackMap.find(S)
3:     if found then
4:         return prob
5:     else                          ▷ First time visiting this call stack
6:         CallStackMap.insert(S, 1)
7:         return 1                                  ▷ Simulate crash
8: procedure RUSSIANROULETTE(CallStack S)
9:     prob ← GetCrashProbability(S)
10:    crashProb ← rand(0,1)
11:    if crashProb < prob then
12:        SimulateCrash()
13:        CallStackMap.update(S, prob/2)
```

To solve this issue, we propose to capture the program call stack when a crash is simulated as a means to limit duplicate testing. Basically, we cache the call stacks of the scenarios that were already tested and avoid triggering a crash simulation when the same call stack is visited again. However, the same call stack does not mean the exact same scenario. We argue that testing the same call stack several times is beneficial in the sense that corner cases are more likely to be covered. Consider the following code for the previous example of an array append operation:

```
1 array[size] = val;
2 persist(array);
3 size++;
4 persist(&size);
```

The error is that it is always the first cache-line-sized piece of $array$ that is flushed instead of the cache line that holds the newly appended value. If a crash is simulated only once in the append operation, this error might remain unnoticed since the first appended value will be located in the first cache-line-sized piece of $array$. Therefore, we propose to exponentially decrease the probability of

[1]See the MAP_PRIVATE flag of *mmap*: `http://man7.org/linux/man-pages/man2/mmap.2.html`

simulating a crash with the same call stack. Algorithm 2 illustrates the different steps of this process. Basically, we map tested call stacks to a corresponding probability. If the call stack has never been visited before, we insert the new call stack in the map together with a corresponding probability of 1. This probability is divided by two whenever a crash is simulated at the same call stack. We use our own call stack unwinder that fetches the whole call stack to avoid any ambiguity.

3.4 Memory reordering

Although it is advised to always protect a flushing instruction with two memory barriers, it can sometimes be useful to group several flushing instructions together for optimization purposes–e.g., when the order of writes is not important. However, such optimizations may stem from wrong ordering assumptions, hence leading to errors. Consider again the previous example of an array append operation:

```
1 array[size] = val;
2 barrier();
3 flush(&array[size]);
4 /* Missing memory barrier */
5 size++;
6 persist(&size);
```

The code above tries to optimize by skipping one memory barrier that should be at line 4. As a result, lines 3 and 5 might be reordered by the CPU and the new value of $size$ might be made persistent before the newly appended value.

Algorithm 3 Stashing persistent writes

```
1:  procedure RussianRoulette(CallStack S, Map Stash)
2:      prob ← GetCrashProbability(S)
3:      crashProb ← rand(0,1)
4:      if crashProb < prob then
5:          for each subset of Stash do
6:              Copy subset to mirror segments
7:              SimulateCrash()
8:              Undo subset
9:          CallStackMap.update(S, prob/2)
10: procedure Flush(PPtr PAddr, Char Content[CacheLineSize])
11:     found ← Stash.find(PAddr)
12:     if found then
13:         Stash.update(PAddr, Content)
14:     else
15:         Stash.insert(PAddr, Content)
16: procedure Barrier
17:     if Stash is not empty then
18:         S ← getCallStack()
19:         RussianRoulette(S, Stash)
20:     for (PAddr, Content) in Stash do
21:         copyToMirror(PAddr, Content)
```

Errors caused by memory reordering are one of the most challenging situations for SCM-based software. We propose an extension to our testing framework that enables us to partially take them into account during testing. Instead of directly copying flushed cache lines into the mirror segments, we propose to stash them first in a map where the key is the persistent address of the cache-line-sized piece of data, and the value the content of that piece of data. This stash is emptied whenever a memory barrier is issued. Basically, we take into account the reordering of consecutive flushes that were not ordered by memory barriers. Algorithm 3 shows the pseudo-code of the overloaded *Flush* and *Barrier* functions, as well as the procedure *RussianRoulette* whose purpose is to trigger crash simulations. The latter function is systematically called inside the *Barrier* function, because it replaces the *Flush* function as the only one that changes the state of the mirror segments. Section 3.7 elaborates on the memory reordering cases that are not covered by our framework.

3.5 Crash simulation during recovery

Contrary to Yat, our testing framework is able to *automatically* simulate crashes during the recovery procedure. To achieve this, we allow crash simulation in both the main process and the test process. To be able to simulate crashes in the test process, we need new segment copies in which we replicate the data that is explicitly flushed by the test process. Therefore, we need to first make copies of the mirror segments before the test process starts executing. When a crash simulation is triggered in the test process, a third process which we denote as recovery test process, is forked and will perform recovery using the copies of the mirror segments with copy-on-write. We keep these copies until the test process finishes executing, upon which we delete them. Hence, only one copy of the mirror segments is needed during the lifetime of the test process regardless of the number of crash simulations that are triggered in it.

To limit the higher cost of nested testing, we propose to (1) test the software without allowing crashes in the test process; (2) devise a minimalistic test program whose crash scenarios cover the whole recovery procedure; (3) test the latter program while allowing crash simulation in the test process. The goal of the minimalistic test program is to mitigate the higher cost of nested crash simulation which requires the additional step of making copies of the mirror segments. If a crash occurs during testing, a debugger can be attached to the three processes, namely the main process, the test process, and the recovery test process. Figure 2 gives a global overview of the complete testing framwork, with all the optimizations and features discussed so far.

3.6 Testing of multi-threaded programs

Single-threaded consistency and recovery correctness in addition to concurrency correctness (e.g., no race conditions) is a good indicator of multi-threaded consistency and recovery correctness. However, there are programming errors that cannot be detected in single-threaded execution. These errors typically concern code paths that execute only in multi-threaded mode. Consider the following example where two threads try to increment one of two persistent counters, $ctr1$ and $ctr2$:

```
1  mutex m1, m2;
2  if(m1.try_lock()){
3      ctr1++;
4      persist(&ctr1);
5      m1.unlock();
6  }else if(m2.try_lock()){
7      ctr2++;
8      persist(&ctr1) /* Should be ctr2 */
9      m2.unlock();
10 }
```

The error in the code above is at line 8 where $ctr1$ is persisted instead of $ctr2$. In the case of single-threaded execution, lock $m1$ will always be successfully acquired, hence, lines 7 to 9 will never be executed and the error will not be detected.

Our testing framework is capable of testing multi-threaded programs. Indeed, it is possible to suspend all the threads of a process as long as the process keeps references of all its threads. Only when this is not the case (e.g. if some threads are detached) the program needs to be changed to provide a procedure that halts all its threads. Additionally, the call stack map and the stash of pending writes need to be made thread-safe using global locks so that only one thread is allowed to try and trigger a crash simulation at a time. Avoiding

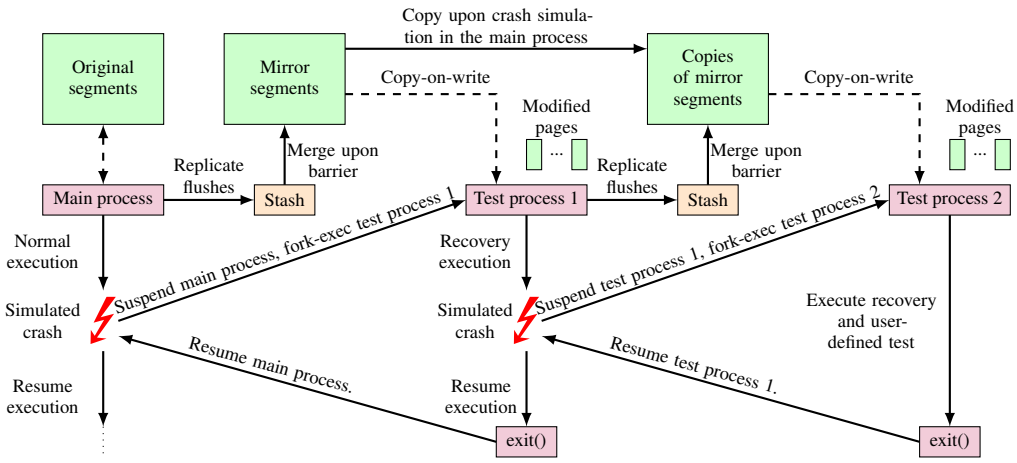


Figure 2: Global overview of the testing framework including copy-on-write optimizations, memory reordering, and nested crash simulation.

duplicate testing by caching the call stack of the thread that triggered the crash simulation becomes less effective in a multi-threaded environment. This is because the crash scenario depends on the state of all threads and not only the thread that triggered the crash simulation. Nevertheless, as discussed in Section 2, exhaustive testing for multi-threaded programs is infeasible because thread scheduling is non-deterministic, hence, two runs of the same multi-threaded program may produce different scenarios.

3.7 Limitations and complementarity with Yat

While our testing framework is able to detect a wide range of consistency-related errors, it is unable to find errors that stem from the reordering of consecutive SCM writes whose persistence is delayed but still enforced in the right order. Consider the following code snippet of an array append operation:

```
1 array[size] = val;
2 size++;
3 persist(&array[size]);
4 persist(&size);
```

The issue in the code above is that the new value of $size$ can be speculatively evicted from the CPU cache and become persistent before the newly appended value. A power failure at this same instant would leave the array in an inconsistent state. Our testing framework cannot detect this issue because it captures only *persistence primitives* and not *individual SCM writes*. If lines 3 and 4 are swapped, our testing framework would detect it as a wrong persistence order. In contrast, Yat is able to detect such errors since it captures all SCM writes.

We advocate coupling our testing framework with Yat. Our testing framework can quickly test several classes of SCM-related errors, leaving out only the class of errors described above, for which Yat can be used. Yat can eliminate the scenarios that were tested using our testing framework and focus instead only on SCM write reordering between two cache line flushing instructions. This significantly decreases the amount of combinations that Yat needs to test.

Unfortunately, there is another class of errors that neither our testing framework nor Yat can detect. Consider the example of Section 3.3:

```
1 array[size] = val;
2 persist(array);
3 size++;
4 persist(&size);
```

The error in this case is in line 2 where we persist the first cache-line-sized piece of $array$ instead of the appended value. If the test program appends only a few values which fit in a single cache line, line 2 will still persist the newly appended value. Hence, the error will remain unnoticed both in our testing framework and in Yat. If in the release version of the software the array spans more than one cache line, data consistency will not be guaranteed.

4. EVALUATION

We evaluated our testing framework on a persistent B-Tree and on a persistent SCM allocator. The allocator code base is around 7000 lines of code (excluding blank lines and comments), which is fairly large for an evaluation. We use *tempfs* to simulate SCM. We consider the following test scenarios:

- *PTree-N:* The main program consists of inserting N key-value pairs in the persistent B-Tree, then erasing them. The test program that is executed upon crash simulation consists in executing recovery and checking that the tree is in a consistent state.
- *PAlloc-N*: The main program consists of interleaving N allocation and N deallocation of blocks of sizes between 128 Bytes and 1 KB. The test program consists in executing recovery, deallocating all currently allocated blocks, then inspecting the allocator against memory leaks.

In the case of PTree-N, the persistent tree uses the persistent SCM allocator. Hence, some of the simulated crashes will be triggered inside the allocator code. This is useful because it will stress the scenarios where a memory leak might happen because the persistent tree did not properly track its own allocations or deallocations.

First, we execute the test scenarios without allowing crash simulation in the test process (i.e. no nested crash simulation). Table 1 illustrates the number of persistence primitives, the number of crash scenarios, and the total testing time with and without taking into account the call stack. We observe that when not taking into account the call stack, the number of crash scenarios grows linearly with the number of operations. When taking into account the call stack however, the number of crash scenarios grows very slowly, which allows speedups of 41.7x and 30.5x for PTree-10000 and PAlloc-10000, respectively, compared with not taking into account the call stack.

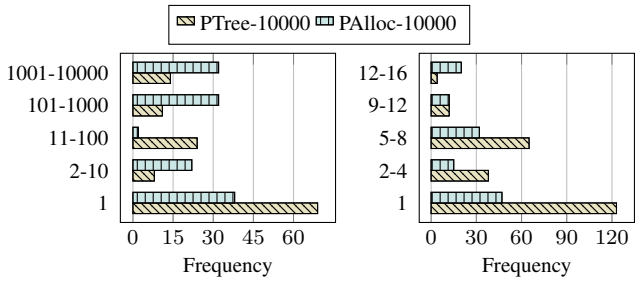
We depict in Figure 3 the frequency of simulated crashes in visited call stacks for PTree-10000 and PAlloc-10000. We observe that when taking into account the call stack, the highest number of simulated crashes with the same call stack is limited to 16. When not taking into the account the call stack, this number increases up to

Test	Main program stats		#Crash simulation		Time	
	#Flush	#Barrier	w/o CS	w/ CS	w/o CS	w/ CS
PTree-100	747	1238	966	234	20.5 s	5 s
PTree-1000	4488	8440	8524	390	179 s	8 s
PTree-10000	42494	81436	84392	834	1795 s	20 s
PAlloc-100	2254	4346	4034	322	84 s	7 s
PAlloc-1000	17643	38730	34902	452	730 s	10 s
PAlloc-10000	173479	384046	346580	702	7199 s	23 s

Table 1: Performance of the testing framework with nested crash simulation disabled. Call stack is abbreviated as CS.

Test	Time	
	w/o CS	w/ CS
PTree-100	218 s	98 s
PTree-1000	710 s	126 s
PTree-10000	6120 s	273 s
PAlloc-100	DNF ($\approx$ 1.3 d)	227 s
PAlloc-1000	DNF ($\approx$ 11.5 d)	414 s
PAlloc-10000	DNF ($\approx$ 114.8 d)	1349 s

Table 2: Performance of the testing framework with nested crash simulation enabled.



(a) w/o call stack (b) w/ call stack

Figure 3: Frequency distribution of the the number of simulated crashes per call stack. The y axis represents the considered range groups of the number of crash simulations per call stack, while the x axis represents the cardinality of these groups.

10000 (as expected). Hence, our approach efficiently limits duplicate testing. We argue again that removing all duplicate testing is not necessarily a good option: while testing the persistent allocator and the persistent B-Tree, duplicate testing allowed us to detect errors that were the result of corner cases in already visited call stacks.

As a second step, we allow nested crash simulation in order to test the crash-safety of recovery procedures. We set a limit of one day per run. We report the results in Table 2. As expected, the testing time is higher than without nested crash simulation. Still, taking into account the call stack keeps the testing cost reasonable and yields significant speedups: 22.4x and 7352.6x for PTree-10000 and PAlloc-10000, respectively, compared with not taking into account the call stack. The three call-stack-oblivious PAlloc test scenarios exceeded the one-day threshold and we report an estimation of their running time. This is explained by the fact that the recovery procedure of the persistent allocator is larger and more complex than that of the persistent tree. Hence, the number of nested crash simulations is much higher for PAlloc than for PTree.

The testing framework detected many errors, such as missing flushes, in the persistent allocator and the persistent B-Tree. Some errors however were non-trivial such as errors in the logic of recovery procedures. These errors would not have been detected without nested crash simulation. Interestingly, the call-stack-oblivious version of the framework did not detect any errors that were not detected by the call-stack-aware version. We conclude that thanks to its call stack awareness, our testing framework enables fast crash-safety testing, even with nested crash simulation, without compromising the quality of testing.

5. CONCLUSION

In this paper we presented a lightweight automated testing framework for software that uses SCM as a universal memory, especially modern database systems. Our testing framework can simulate software crashes and power failures following a suspend-test-resume approach. It makes efficient usage of copy-on-write memory mapping to speedup the testing process. Additionally, it achieves fast code coverage by caching the call stacks of the already tested crash scenarios. Our testing framework is also able to simulate nested crashes in a fully automated way, that is, crashes that occur during the recovery procedure of a test program executing in an ongoing crash simulation. Our experimental evaluation shows that our testing framework successfully finds a wide range of consistency and recovery errors, and is fast enough to be used continuously during development of fairly large software systems.

The focus of our testing framework is to improve the quality of SCM-based software rather than to achieve comprehensive crash-safety testing, which we argue is infeasible. For future work, we plan to extend our testing framework to take into account data that is speculatively evicted from the CPU cache. We also plan to refine our memory model to take into account new persistence primitives, such as *clwb* and *pcommit*. Moreover, we plan to investigate new development life cycles that are fit for SCM-based software and that involve model-based verification. Indeed, we argue that theoretical consistency and recovery guarantees must precede experimental verification. Finally, we believe that testing of SCM-based software will receive increasing attention in the near future as real SCM-based systems start to emerge.

Acknowledgements

We thank the anonymous reviewers for their constructive comments that helped us improve the paper. This work is partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "SFB 912/HAEC" as well as the Cluster of Excellence "Center for Advancing Electronics Dresden (cfaed)".

6. REFERENCES

[1] Intel®Architecture Instruction Set Extensions Programming Reference. Technical report, 2015. http://software.intel.com/en-us/intel-isa-extensions.

[2] SNIA NVM Programming Model V1.1. Technical report, 2015. http://www.snia.org/sites/default/files/NVMProgrammingModel_v1.1.pdf.

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *ACM SIGMOD*, 2015.

[4] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5), 2015.

[5] S. Chen and Q. Jin. Persistent B+-trees in non-volatile main memory. *PVLDB*, 8(7), 2015.

[6] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making

persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGPLAN Not.*, 47(4), 2011.

[7] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *ACM SIGMOD*, 2015.

[8] P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson. Yat: A validation framework for persistent memory software. In *USENIX ATC*, 2014.

[9] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main-memory databases. In *CIDR*, 2015.

[10] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *USENIX OSDI*, 2014.

[11] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *USENIX FAST*, 2011.

[12] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not., 47(4)*, 2011.

[13] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *USENIX OSDI*, 2006.

[14] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: reducing consistency cost for NVM-based single level systems. In *USENIX FAST*, 2015.