Execution of NVRAM Programs with Persistent Stack

Vitaly Aksenov[1], Ohad Ben-Baruch[2], Danny Hendler[3], Ilya Kokorin[4], and Matan Rusanovsky[5]

[1] ITMO University, *Addr:* 49, Kronverksky ave, Saint-Petersburg, Russia, 197101; *Email:* aksenov.vitaly@gmail.com; *Phone:* +79516623399
[2] Ben-Gurion University, *Addr*: P.O.B. 653 Beer-Sheva, Israel, 8410501; *Email:* bbohad@gmail.com; *Phone:* +97286461600
[3] Ben-Gurion University, *Addr*: P.O.B. 653 Beer-Sheva, Israel, 8410501; *Email:* hendlerd@bgu.ac.il; *Phone:* +97286461600
[4] ITMO University, *Addr:* 49, Kronverksky ave, Saint-Petersburg, Russia, 197101; *Email:* kokorin.ilya.1998@gmail.com; *Phone:* +79811639149
[5] Ben-Gurion University, *Addr*: P.O.B. 653 Beer-Sheva, Israel, 8410501; *Email:* matanru@post.bgu.ac.il; *Phone:* +97286461600

Abstract. Non-Volatile Random Access Memory (NVRAM) is a novel type of hardware that combines the benefits of traditional persistent memory (persistency of data over hardware failures) and DRAM (fast random access). In this work, we describe an algorithm that can be used to execute NVRAM programs and recover the system after a hardware failure while taking the architecture of real-world NVRAM systems into account. Moreover, the algorithm can be used to execute NVRAM-destined programs on commodity persistent hardware, such as hard drives. That allows us to test NVRAM algorithms using only cheap hardware, without having access to the NVRAM. We report the usage of our algorithm to implement and test NVRAM CAS algorithm.

Keywords: Concurrency · Shared memory · Persistency · NVRAM.

1 Introduction

For a long time the industry assumed the existence of two distinct types of the memory. The first one is a persistent memory that preserves its content even in the presence of hardware (e.g., power) failures. This type of memory was assumed to support mainly sequential block access with the poor performance of random access. Due to its ability to persist data this kind of memory is widely used to recover the system after a hardware failure: one can load the data from the persistent memory and restore the state of the application before the crash. The second type of the memory is DRAM that supports fast random byte-addressable access but loses its content on hardware failures. Due to its speed, this kind of memory is widely used in high-performance computations.

Nowadays, we can get benefits from both of these worlds due to the invention of Non-Volatile Random Access Memory (NVRAM)—a novel type of hardware

that combines both the persistency and the fast random access. This allows us to implement low-latency persistent data structures that require random access to the memory, e.g., binary search trees, linked lists, and etc. A lot of work has been done to come up with data structures, hand-tuned for the NVRAM [11, 10, 13]. Some authors propose techniques, that can be used to transform DRAM-resident data structures into the ones suitable for the NVRAM [9, 12].

Despite the speed of the NVRAM is compatible with the speed of the DRAM, the NVRAM is not expected to replace volatile memory totally since processor registers and the NVRAM cache are expected to remain volatile. Thus, even on NVRAM systems, a system failure leads to: 1) the loss of the results of recent computations since `x86` computations are performed using volatile processor registers, and 2) the loss of data that was written to the NVRAM cache and has not been flushed to the NVRAM.

To make sure that the written data becomes persistent, we should flush one or several cache lines to the NVRAM. Flush of a single cache line is an atomic action: if a crash occurs during cache line flushing, the whole cache line is either persisted or not. However, if we want to flush multiple cache lines at a time, a crash event can occur between flushes — in such a case, only a part of the data becomes persistent while the rest is lost.

This yields one of the major challenges of NVRAM. If a system failure happens during a complex update, when some updated values have been flushed to the NVRAM from the cache while others still reside in the cache, non-flushed memory is lost and after the restart the NVRAM appears to be in an inconsistent state.

Due to the difficulty of ensuring storage consistency in the presence of the volatile NVRAM cache, a lot of works assume the absence of such cache [3–5, 7]. However, in this work we consider real-life systems, thus we take the volatility of the NVRAM cache into account.

Another problem with the NVRAM is defining which executions are considered "correct" in the presence of hardware failures, that can lead to the loss of data. Despite a lot of correctness conditions were defined in the previous years [1, 5, 14, 17, 3], only *Nesting Safe Recoverable Linearizability* [3] describes the work with nested functions. Thus, maintaining persistent call stack is a crucial part of systems based on this concept. However, while methods of maintaining NVRAM heap are well-studied [8, 6], methods of maintaining the persistent program stack are not studied at all: other works just assume the existence of a persistent call stack [3, 4, 12].

Moreover, the persistent stack allows us to design and implement novel complex system recovery algorithms, which can be faster than traditional log-based system recovery methods. Previously, such complex algorithms were considered impractical for traditional persistent memory systems due to the high latency of random access of traditional persistent memory, following directly from its mechanical nature, but on NVRAM-based systems such complex algorithms may be found useful.

In this work, we describe an algorithm, based on the implementation of the persistent call stack, that can be used to execute NVRAM programs and recover

the system after a hardware failure while taking the architecture of real-world NVRAM systems into account. Moreover, the algorithm can be used to execute NVRAM-destined programs on commodity persistent hardware, such as hard drives. That allows us to test NVRAM algorithms using only cheap persistent hardware, such as HDD, SSD, etc., without having access to the NVRAM. We report the usage of our algorithm to implement and test correct and incorrect versions of the NVRAM CAS algorithm [3]. Also, we describe a method, that can be used to verify executions of NVRAM CAS algorithm for serializability.

The rest of the work is organized as follows. In Section 2, we discuss the system model, various failure models, operation execution model and talk about different correctness conditions, suitable for the NVRAM. In Section 3, we discuss the concept of the persistent program stack and its implementation. In section 4 we present the solutions for the challenges we faced during the implementation of our algorithm. Also, we show there the architecture of the system along with the system recovery algorithm. In Section 5, we discuss the usage of our algorithm to implement and verify the NVRAM CAS algorithm, along with the method of checking executions of the NVRAM CAS algorithm for serializability. In Section 6, we discuss the directions of the future research. We conclude our work with Section 7.

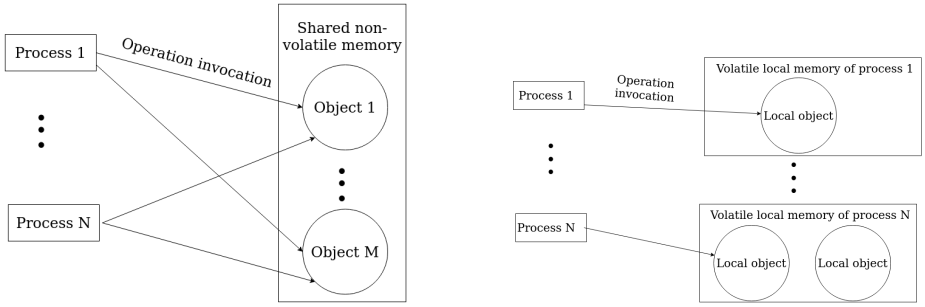
2 Model

2.1 System model

Our system model is based on the model described in [3].

There are N processes $\{p_i\}_{i=1}^{N}$ executing operations concurrently. Also, there are M objects $\{O_j\}_{j=1}^{M}$ located in the shared non-volatile memory. Processes communicate with each other by executing operations on shared objects (see Fig. 1a), that can support `read`, `write` or `read-modify-write` [15] operations.

In our model, all shared memory is considered non-volatile, i.e., it does not lose its content even after a crash event. However, we assume the existence of a volatile memory in the system. Each object LO, located in the volatile memory, is considered local to some process p. In other words, only process p can access object LO. Thus, besides being able to execute operations on shared objects, each process can access its local objects. Such objects support only `read` and `write` operations (see Fig. 1b).



(a) Execution of operations on shared objects

(b) Execution of operations on local objects

Fig. 1: System execution model

However, our model still does not reflect some properties of the real-world hardware: for example, it does not take into account the existence of the volatile NVRAM cache and the existence of shared volatile memory.

2.2 Failure model

There exist two general failure models:

- *Individual* crash-recovery model [3]. In such a model, each process can face a crash event independently of all other processes. When a process faces a crash event, it stops working until it is restarted. All data, stored in the volatile memory of the failed process, is lost. However, all data persisted to the NVRAM is not lost and remains available to the failed process after its restart.
- *System* crash-recovery model. In such a model, a crash event happens in the whole system instead of an individual process. The whole systems stops working until it is restarted. After the system restarts, the contents of all the volatile memory is lost. As in the previous model, the data, persisted in the NVRAM, is not lost and remains available to all processes after the system restarts.

Note, that the *system* crash-recovery model is a special case of the *individual* crash-recovery model, since a crash of the whole system can be represented as a set of N simultaneous crash events of individual processes — one crash event per each process. Despite the fact that *individual* crash-recovery model is a more general model, in this work we focus mainly on *system* crash-recovery model. In real-world shared memory systems multiple computational units are placed in a single server and thus a failure of a single computational unit is impossible without a failure of the entire system. That is why, in our opinion, *system* crash-recovery model describes more accurately the real-life crash event — for example, power loss.

2.3 Operation execution

We say that function `F` is being executed by process p if execution of `F` has been started by p but has not been finished yet. As described in [3], we work with the nested invocation of functions: at any moment, multiple functions can be executed by any process. It happens when function `F` invokes function `G`. Thus, executed functions in each process form a nested sequence. In the above example execution of `G` is nested into the execution of `F`.

To allow the recovery of the system, we provide each function `F` with a dual function `F.Recover`, which receives the same arguments as `F`. `F.Recover` is called after the system restart to perform system recovery and it should either finish the execution of `F` or roll back `F`.

To perform the system recovery, for each process p we should call `F.Recover` for each function `F` being executed by p at the crash moment. Moreover, recovery functions should be called in the certain order: if the execution of `G` is nested into the execution of `F`, `G.Recover` should be called before `F.Recover`. Thus, each process should perform the recovery in the LIFO (stack) order.

Also we should consider the possibility of *repeated failures* — failures which happen during the recovery procedure. Consider the system failure after `F` was invoked. After the restart, we should call `F.Recover` to complete the recovery. Suppose another system failure happens before `F.Recover` is finished. After the second restart, we should again continue the recovery at executing `F.Recover`. It means that there is no difference between the system failure happening during the execution of `F` or during the execution of `F.Recover`: in both cases, we should call `F.Recover` to complete the recovery. Thus, `F.Recover` should be designed so that it can complete the operation (or roll it back) no matter whether the crash occurred when executing `F` or `F.Recover`.

2.4 Correctness

Multiple correctness conditions for NVRAM exist. Here, we outline three most important (from the strongest to the weakest):

1. *Nesting Safe Recoverable Linearizability* [3]. It requires each invoked function `F` to be completed even if a crash event occurs while executing `F`. Thus, under that correctness condition, `F.Recover` should finish the execution of `F` either by completing it successfully or by rolling it back.
2. *Durable Linearizability* [17]. It requires that each function `F`, execution of which has finished before a crash, should be completed. If a crash event occurs while executing function `F`, such function may be either completed or not.
3. *Buffered Durable Linearizability* [17]. It is a weaker form of *Durable Linearizability*. Its difference is in that it allows function `F` not to be completed even if its execution finished before a crash. However, that correctness condition requires each object to provide `sync` operation — all functions, finished before a call to `sync` must be completed, even if a crash event occurs.

In this work, we propose an algorithm that can be used to run NVRAM-destined programs under *Nesting Safe Recoverable Linearizability* — the strongest correctness condition.

3 Persistent Stack

3.1 Program stack concept

In order to execute programs for NVRAM, for each thread[6] t we maintain an information about functions, which were executed by t when a crash occurred. Also, to invoke recover functions in the correct order we maintain the order in which these functions were invoked.

We maintain that order by using the notion of program stack: each thread t has its own NVRAM-located stack, and each function executed by t corresponds to a single frame of the stack. When a function is invoked, the corresponding frame is added to the top of the stack. After the end of the execution, the frame is removed from the top. Therefore, when a crash occurs, the stack of thread t contains frames, that correspond to functions that were executed by t at the crash moment. Moreover, such frames are located in the correct order: if execution of `G` was nested into the execution of `F`, a frame of `G` is located closer to the top of the stack, than a frame of `F`.

3.2 Issues of existing implementations

The functionality of the program stack is already implemented by standard execution systems: for example, `x86` program stack. However, we cannot use them as-is, even if we transfer it from the DRAM to the NVRAM.

Here we remind the implementation of the function call via the `x86` stack. Suppose function `F` calls function `G` using `x86` command `CALL G`. To perform such an invocation, we should store a return address on the stack — the address of the instruction in function `F` that follows the instruction `CALL G`. After the execution of `G` is finished, we continue execution of `F` from that instruction. This is exactly how `x86` instruction `RET` works — it simply reads the return address from the stack and performs `JMP` to that address, allowing it to the continue the execution from the desired point.

Note that such a program stack implementation has a number of drawbacks, that makes it impossible for us to use such implementation as a persistent stack:

– After the system restart due to the crash, the code segment may be relocated, i.e., have a different offset in the virtual address space. That will make us incapable of identifying which functions were executed at the crash moment — we simply won't be able to match return address from the stack with an address of some instruction after the code segment relocation.

[6] When talking about practical aspects of concurrent programming, we use the word "thread" in the same context, as the word "process" in the theory of concurrent programming

– We cannot guarantee an atomicity of adding a new frame to the stack or removing a frame from the top of the stack – if a crash occurs during adding or removing stack frame, after the system restarts the stack might be in an inconsistent state.

Thus, instead of using existing program stack implementations, we present our persisted stack structure that overcomes the above drawbacks.

3.3 Persistent stack structure

Each thread has an access to its own persistent stack. For simplicity, in this section we assume that the persistent stack is allocated in the NVRAM as a continuous memory region of constant size. However, we explain how to make a stack of unbounded size in Appendix A of the full version of the paper, available at [2].

Persistent stack consists of consequent persistent stack frames — one frame per function that accesses NVRAM. [7] Each frame ends with a one-byte end marker: it is `0x1` (*stack end marker*) if the frame is the last frame of the stack; otherwise, it is `0x0` (*frame end marker*). Any data located after the *stack end marker* is considered invalid — it should never be read or interpreted in any way (see Fig. 2).

Fig. 2: Persistent stack structure

To finish the description of the data layout, each persistent stack frame consists of: 1) a unique identifier of the invoked function that allows us to call the appropriate recover function during the system recovery; 2) arguments of the function, serialized into a byte array—during the system recovery we pass them to the recover function; 3) a one-byte end marker (either `0x0` or `0x1`).

3.4 Update of the persistent stack

The persistent stack should be updated: 1) when the function is invoked — a new frame should be added to the top of the stack, 2) when the function execution is finished — the top frame of the stack should be removed.

Adding the new frame to the top of the stack. Suppose the stack at the beginning of the operation has two frames in it (see Fig. 3a):

To add a new frame to the top of the stack, we perform the following actions:

[7] Each such function must have a recover version, as described above.

1. After the *stack end marker*, we write a new frame with the *stack end marker* set. Note that the new frame (frame 3) is located after the *stack end marker* of the previous frame (frame 2). Therefore, the new frame is not considered as a stack frame, while the previous frame (frame 2) is still the last stack frame (see Fig. 3b).

2. Change the end marker of the current last stack frame (frame 2) from `0x1` to `0x0`. Thus, the last stack frame (frame 2) becomes the penultimate stack frame and the new frame (frame 3) becomes the last stack frame (see Fig. 3c). We name that one-byte end marker changing operation as *moving the stack end forward*.

(a) Persistent stack before the function invocation

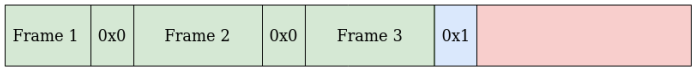
(b) Persistent stack after writing the new frame after the stack end marker

(c) Persistent stack after adding the new frame to the top of the stack

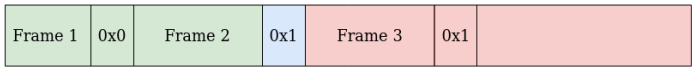
Fig. 3: Adding new frame to the top of the stack

Removing the top frame from the stack. Suppose the stack at the beginning of the operation has three frames in it (see Fig. 4a):

To remove the top frame from the stack, we simply change the end marker of the penultimate stack frame (frame 2) from `0x0` to `0x1`, thus making the penultimate stack frame the last stack frame (see Fig. 4b). We name that one-byte end marker changing operation as *moving the stack end backward*. Note, that frame 3 becomes the part of the invalid data and, therefore, it will not be considered as a stack frame anymore.



(a) Persistent stack before exiting from a function



(b) Persistent stack after exiting from a function

Fig. 4: Removing the top frame from the stack

Dummy frame. Note that both frame removal and frame addition procedures assume the existence of at least one frame in the stack, besides the one that is being removed or added. Particularly, this assumption implies that the bottom stack frame cannot be removed from the stack. We can simply satisfy that assumption by introducing a dummy frame — the stack frame, located at the bottom of the stack (i.e. the first frame, added to the stack). That frame is added to the stack at the initialization of the stack and is never removed. By that, we ensure that there is always at least one frame, thus making it possible for us to use the stack update procedures, described above.

Flushing long frames. Note that sometimes a new stack frame does not fit into a single cache line — for example, that can happen when some function receives arguments list with length greater than the cache line size. In such case, we will not be able to add such frame to the stack atomically (since only single cache line can be persisted atomically). Therefore, we can face a crash event that will force us to write the new frame partially (see Fig. 5).

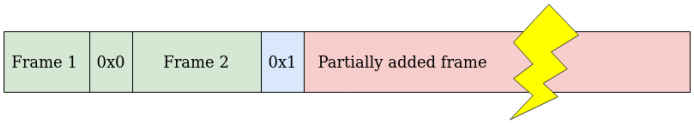


Fig. 5: Persistent stack with partially flushed frame

In our algorithm, we, at first, add a new frame to the stack, and only after the new frame has been written successfully we *move the stack end forward*. Thus, even if the crash event happens, the stack will remain consistent: partially written frame will be located after the *stack end marker* and will not be considered as a stack frame. Therefore, this scenario does not brake *Nesting-safe Recoverable Linearizability*, since the last function invocation was not linearized before the crash event. We can simply think that the crash happened before the function invocation and the function was never invoked.

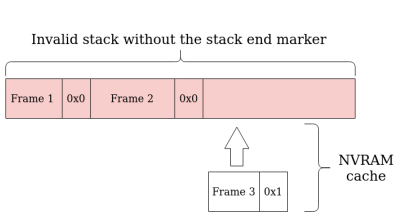
The atomicity of the stack update. We can say that a function invocation linearizes only when we *move the stack end forward*. This requires only the flushing of a single byte to the NVRAM. Since a single byte always resides in a single cache line, this flush always happens atomically.

The same observation can be made for *moving the stack end backward*: an execution of the function is finished when we change the end marker of the penultimate stack frame from `0x0` to `0x1`. As was described above, such action happens atomically.

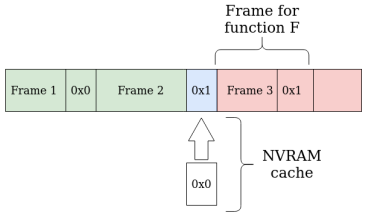
Persistent stack and the NVRAM The procedure of adding and removing a stack frame requires only the ability to flush a single byte atomically and not the entire cache line — this makes us capable of implementing the stack maintenance algorithm on a hardware that does not support atomic flushing of an entire cache line. Thus, the algorithm described above, can be easily emulated without having access to an expensive NVRAM hardware, using almost any existing persistent hardware such as HDD, SSD, etc.

For the above reasoning to remain correct, we should maintain two following invariants:

1. We should flush the new stack frame before *moving the stack end forward*. Suppose we violate that rule. Consider a crash event that happens at some time after the *moving the stack end forward*. Suppose also, that new stack frame (frame 3) has been written to the volatile NVRAM cache and was lost during the crash. After the system restart we will not be able to call the recover function for frame 3, because we have lost that frame (see Fig. 6a).
2. When changing the end marker of some frame (either from `0x0` to `0x1` or vice versa) we should immediately flush it before staring the execution of the invoked function or continuing the execution of the caller function. Suppose we violate that rule. Consider a crash event, happening while executing function `F`, corresponding to frame 3. Also consider that the *frame end marker*, written to frame 2, has been written to the volatile NVRAM cache and thus has been lost (see Fig. 6b). After the system restart, we do not consider frame 3 as a stack frame, and, thus, we do not even invoke `F.Recover`.



(a) New stack frame has been lost due to volatility of the NVRAM cache



(b) End marker has been lost due to volatility of the NVRAM cache

Fig. 6: Results of violating flushing invariants

4 System implementation

4.1 Pointers to the memory in NVRAM

When working with pointers to the NVRAM we face the problems similar to those we faced when working with function addresses (Section 3.2). Suppose we have acquired pointer `ptr` pointing to the NVRAM. We store `ptr` in the NVRAM (for example, in some persistent stack frame, as an argument of some function `F`). After that, we face a crash event. And when we restart the system, the mapping of the NVRAM into the virtual address space can change, thus, making pointer `ptr` invalid, since it does not point to the NVRAM anymore.

The same problem happens when we emulate NVRAM using HDDs, mapped to the virtual address space using `mmap` syscall: on each system restart, HDD is mapped to a different location in the virtual address space.

This problem has a very simple solution: instead of using direct pointers to the NVRAM, we shall use offsets from the beginning of the NVRAM mapping into the virtual address space. Suppose the mapping of the NVRAM begins at address `MAP_ADDR`. Then, instead of storing `ptr` we store `ptr - MAP_ADDR` — an offset of the desired memory location. Note that such an offset does not depend on an exact location of the mapping, thus making it safe for us to store it in the NVRAM and use after the system restart.

4.2 Handling return values

Traditionally, on `x86` architecture, functions return value using the volatile memory — either in `x86` register `EAX`, if the return value is an integer, or in `FPU` register `ST0`, if the return value is floating-point. For example, `cdecl`, one of the most popular `x86` calling conventions, implies the above rules for return value.

However, in our case we cannot use volatile processor registers to store return value. Consider a crash event occurring after the callee function `G` has saved the return value to the `EAX` and finished its execution by *moving the stack end backward*. At that time, the caller function `F` has not persisted the return value from `EAX` to the NVRAM. After the system restart, we will not invoke `G.Recover`, but start from `F.Recover` instead. However, we cannot execute `F.Recover` properly, because we have lost the result of `G`.

That is why functions should store their results directly in the NVRAM. We could come up with two approaches where to store them:

1. on the persistent stack. For example, we can use an especially-allocated place in a persistent stack frame for that purpose.
2. in the NVRAM heap. In such a case, the caller can preallocate a memory location for the answer before invoking the callee, and pass the pointer to that memory location in callee's arguments (note, that as was mentioned in Section 4.1, we should use offsets instead of pointers to the NVRAM). After that, callee can store its answer in that memory location.

In both cases, the callee should flush the answer to the NVRAM before *moving the stack end backward*. Our implementation supports returning of small

values (up to 8 bytes) on the persistent stack, while big values are returned in the NVRAM heap.

4.3 Architecture of the system

The system consists of a single main thread and `N` worker threads.

Main thread can run in either a standard mode or a recovery mode.

When running in the standard mode, the main thread performs the following steps.

1. Initialize the NVRAM heap. This may include the initialization of the memory allocator, the mapping of the NVRAM to the virtual address space and the consequent initialization of the variable `MAP_ADDR`, mentioned in Section 4.1
2. Initialize `N` new persistent stacks.
3. Start `N` worker threads, giving each worker thread pointer to the beginning of its persistent stack.
4. Receive task that should be executed by the system and add them to the producer-consumer queue.

When running in the standard mode, worker threads receive tasks from the producer-consumer queue and execute them.

In case of a crash, the main thread starts in the recovery mode and performs the following steps:

1. Initialize the NVRAM heap.
2. Start `N` recovery threads, giving each recovery thread the pointer to a persistent stack of some worker thread.
3. Wait for all recovery threads to finish.
4. Restart the system in normal mode.

Each recovery thread executes the following algorithm:

1. Traverse its persistent stack from the top to the bottom.
2. Execute the corresponding recover operation for each stack frame.
3. After the recovering of an operation on the top of the stack is finished, pop the top frame.
4. After all the frames (except for the dummy one) are removed from the stack, finish the execution.

System recovery happens in parallel, which allows for a faster recovery than an ordinary single-threaded recovery.

We note that our algorithm deals well with *repeated failures*. If such a failure happens during the recovery, the new recovery continues not from the beginning, but from where the previous recovery was interrupted. More formally, consider a frame, corresponding to a function `F`. If during the recovery we have completed execution of `F.Recover` and removed that frame from the stack, even after the *repeated failure* we will not run a recover function for that frame once more. Thus, we achieve the progress even in the presence of *repeated failures*.

5 Verification

The described algorithm of the persistent stack can be used to implement and verify `CAS` algorithm for NVRAM, described in [3]. That paper assumes the absence of the volatile NVRAM cache, i.e., all writes are performed right into the memory. To emulate this, we should flush each written cache line to the NVRAM immediately after the corresponding write. Also, we should implement the algorithm so that each written value never crosses the border of a cache line to allow atomic flush of each written value.

Multiple correctness conditions for concurrent algorithms exist: the most popular are linearizability [16], sequential consistency and serializability [18]. We want to perform the verification against some of these correctness conditions.

From now on we take `CAS` algorithm for the NVRAM as the running example. Consider the following execution. Multiple threads run a set of `CAS` operations on a single register `Reg`: $\{CAS(Reg, old_i, new_i)\}_{i=1}^{N}$, and the initial value of `Reg` is `init`. And for each operation we know whether it was finished successfully or not.

We present an algorithm, that can be used to check such an execution for serializability in a polynomial time.

5.1 Serializability

To verify the execution for serializability in polynomial time, we build a graph $\langle V, E\rangle, G = \{old_i\}_{i=1}^{N} \cup \{new_i\}_{i=1}^{N} \cup \{init\}$ and construct the set of edges E the following way: $a \to b \in E$ if and only if there exists a successful $CAS(Reg, a, b)$ in the execution. Also, we read the final value of the register. We can read it after all the `CAS` operations are finished.

Since each edge of G corresponds to a successful `CAS`, each successful CAS was executed exactly once, and each successful $CAS(Reg, a, b)$ changed value of Reg from a to b, to verify the execution for serializability we should find some Eulerian circuit that starts in the initial value of the register and ends in the final value of the register — such a circuit corresponds to the sequential execution. Thus, the execution is serializable if and only if such a circuit can be found[8].

5.2 Running examples

We have implemented the algorithm, described above, using HDD-based memory-mapped files to emulate the NVRAM. We used UNIX utility `kill` to interrupt the system at random moments by that emulating system crashes.

We have generated random executions of the algorithm in the following way:

1. Generate an initial integer value of the register;
2. Generate $\{new_i\}_{i=1}^{N}$ and $\{old_i\}_{i=1}^{N}$ as integer values, uniformly sampled from some range: either wide range $[-10^5, 10^5]$), or narrow range ($[-10, 10]$);

[8] Please, note that we can simply serialize unsuccessful operation at the times when the register holds a value different from old_i.

3. Start the system in the normal mode, add descriptors of $\{CAS(Reg, old_i, new_i)\}_{i=1}^{N}$ operations to the producer-consumer queue in the random order;
4. Run 4 working threads that execute these `CAS` operations;
5. At random moment, emulate system failure using the `kill` utility;
6. Restart the system in the recovery mode waiting for all `CAS` operations, that were executing at the crash moment, to complete;
7. Restart the system in the normal mode, add all remaining descriptors to the queue;
8. Run steps 4-7 until all operations are completed;
9. Get answers of all `CAS` operations, get the final value of the register, and, finally, verify the execution for serializability.

We have verified a lot of random executions along with emulated system failures at random moments. All executions of the CAS algorithm presented in [3] were found to be serializable. We also verified the executions of incorrect CAS algorithm with especially-added bugs: we have removed the matrix `R` from the `CAS` algorithm. The executions of such a wrong implementation were reported to be non-serializable.

The implementation is publicly available at https://github.com/KokorinIlya/NVRAM_runner.

6 Future work

We find three interesting directions for the future work: 1) implement and test other NVRAM algorithms; 2) find the polynomial algorithm that verifies executions of `CAS` algorithm for linearizability and sequential consistency, or prove that the problem of such a verification is NP-complete; 3) develop a plugin for one of the modern C++ compilers that can be used to reduce the boilerplate code: e.g., automatically create a new stack frame on each function call, remove the top frame when a function execution finishes, and etc.

7 Conclusion

In this paper we presented an algorithm that can be used to run NVRAM programs. The described algorithm takes into consideration different aspects of real-world NVRAM systems. Moreover, the algorithm can be used to run NVRAM-destined programs on commodity persistent hardware, which can be useful for implementing and testing novel NVRAM algorithms without having an access to an expensive NVRAM hardware. The algorithm was successfully used to implement and verify the CAS algorithm for NVRAM.

References

1. Aguilera, M.K., Frølund, S.: Strict linearizability and the power of aborting. Technical Report HPL-2003-241 (2003)

2. Aksenov, V., Ben-Baruch, O., Hendler, D., Kokorin, I., Rusanovsky, M.: Execution of nvram programs with persistent stack, full version (2021), https://arxiv.org/abs/2105.11932
3. Attiya, H., Ben-Baruch, O., Hendler, D.: Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing. pp. 7–16 (2018)
4. Ben-David, N., Blelloch, G.E., Friedman, M., Wei, Y.: Delay-free concurrency on faulty persistent memory. In: The 31st ACM Symposium on Parallelism in Algorithms and Architectures. pp. 253–264 (2019)
5. Berryhill, R., Golab, W., Tripunitara, M.: Robust shared objects for non-volatile main memory. In: 19th International Conference on Principles of Distributed Systems (OPODIS 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
6. Bhandari, K., Chakrabarti, D.R., Boehm, H.J.: Makalu: Fast recoverable allocation of non-volatile memory. ACM SIGPLAN Notices **51**(10), 677–694 (2016)
7. Blelloch, G.E., Gibbons, P.B., Gu, Y., McGuffey, C., Shun, J.: The parallel persistent memory model. In: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures. pp. 247–258 (2018)
8. Cai, W., Wen, H., Beadle, H.A., Hedayati, M., Scott, M.L.: Understanding and optimizing persistent memory allocation. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 421–422 (2020)
9. Chauhan, H., Calciu, I., Chidambaram, V., Schkufza, E., Mutlu, O., Subrahmanyam, P.: {NVMOVE}: Helping programmers move to byte-based persistence. In: 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads ({INFLOW} 16) (2016)
10. Chen, S., Jin, Q.: Persistent b+-trees in non-volatile main memory. Proceedings of the VLDB Endowment **8**(7), 786–797 (2015)
11. David, T., Dragojevic, A., Guerraoui, R., Zablotchi, I.: Log-free concurrent data structures. In: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). pp. 373–386 (2018)
12. Friedman, M., Ben-David, N., Wei, Y., Blelloch, G.E., Petrank, E.: Nvtraverse: In nvram data structures, the destination is more important than the journey. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 377–392 (2020)
13. Friedman, M., Herlihy, M., Marathe, V., Petrank, E.: A persistent lock-free queue for non-volatile memory. ACM SIGPLAN Notices **53**(1), 28–40 (2018)
14. Guerraoui, R., Levy, R.R.: Robust emulations of shared memory in a crash-recovery model. In: 24th International Conference on Distributed Computing Systems, 2004. Proceedings. pp. 400–407. IEEE (2004)
15. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) **13**(1), 124–149 (1991)
16. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(3), 463–492 (1990)
17. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: International Symposium on Distributed Computing. pp. 313–327. Springer (2016)
18. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM (JACM) **26**(4), 631–653 (1979)