Brief Announcement: Survey of Persistent Memory Correctness Conditions

Naama Ben-David
VMware Research, Palo Alto, CA, USA

Michal Friedman
ETH Zürich, Switzerland

Yuanhao Wei
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

In this brief paper, we survey existing correctness definitions for concurrent persistent programs.

2012 ACM Subject Classification Hardware → Memory and dense storage; Theory of computation

Keywords and phrases Persistence, NVRAM, Correctness, Concurrency

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.41

Related Version *Full Version*: `https://arxiv.org/pdf/2208.11114.pdf`

1 Introduction

Non-Volatile Random Access Memory (NVRAM) is a new type of memory technology that has recently hit the market. Its key feature is that it is *persistent*, like SSDs, but is fast and byte-addressable, much like DRAM. This presents a huge paradigm shift from the way persistence could be achieved in the past; techniques that worked well for sequential block-granularity storage cannot be efficiently used with NVRAMs. Achieving persistence with NVRAM has the potential to speed up applications by orders of magnitude.

However, before designing persistent algorithms for NVRAM, we must first answer a more basic question: What does it mean for an algorithm to be persistent?

Despite algorithms relying on external storage for persistence for decades, the answer to the above question is not clear in the context of faster, byte addressible NVRAM. In particular, it is now realistic to require that virtually *no progress be lost* upon a crash, and that a program be able to continue where it left off upon recovery.

The above requirement, while appealing, is in fact not very precise. Due to registers and caches remaining volatile, individual instructions and memory accesses are applied to volatile memory first, and are then persisted separately. If a system crash occurs between when an instruction is executed and when its effect is applied to NVRAM, progress will inevitably be lost. However, it is possible to define how much progress it is okay to lose, and at what point in the execution we expect each instruction's effect to be persisted. For example, we can ensure no completed operation will be lost upon a crash.

In this brief survey, we discuss definitions of persistence that exist in the literature. As this is an actively and quickly developing field of study, there are many different notations, terminologies, and definitions that often refer to similar notions. We put these definitions into the same terminology, and compare them to each other. Using this point of view, we arrange the definitions into a hierarchy, based on the set of execution histories that satisfies every definition. Interestingly, this hierarchy changes depending on specific model assumptions made. We outline common model assumptions and illustrate their effect on these definitions.

We note that this survey is meant to make sense of the various persistence definitions and to guide researchers and algorithm designers when choosing which model and definition to adopt. However, this survey does not cover the many different algorithms, techniques, and applications that have been developed for NVRAM programming in recent years.

2 Model Assumptions

We consider a system of n asynchronous processes $p_1 \dots p_n$. Processes may access *shared base objects* with atomic *read, write, and read-modify-write* primitives. Each process also has access to *local variables* that are not shared with any other process. Objects (both local and shared) may be *volatile* or *non-volatile*, which affects their behavior upon a crash.

A *history* is a sequence of *events*. There are three types of events: an *invocation* event $obj.inv_i(op, v)$ which invokes operation op on object obj by process p_i with argument v, a *response* event $obj.res_i(op, v)$ in which obj responds to p_i's invocation on op with return value v, and *crash* events. A crash resets all the volatile local variables of the associated process, or all volatile objects if all processes crashed.

A response res is said to *match* an invocation inv in H if obj, op, and i are the same for both, and res is the next event in $H|p$ after inv. An operation is said to be *complete* in H if both its invocation and a matching response appear in H. Otherwise, if an operation was invoked but was not completed, the operation is said to be *pending*.

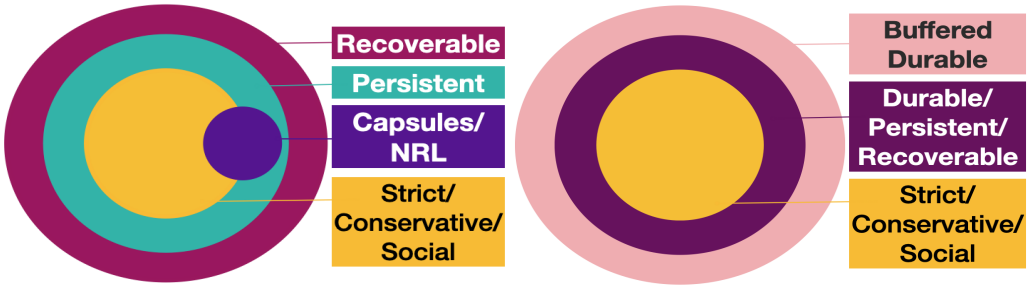
Given a property P and a history H, P is said to be *local* if given a history H in which, for every object O, $H|O$ satisfies P, H also satisfies P.

In the full version of this paper [4], we discuss model variants that appear in the literature and their implications on correctness conditions and implementations.

3 Property Hierarchy

In this section, we present hierarchies relating the existing properties to each other under various model assumptions. A complete list of all formal definitions, including those omitted from this short paper [2, 3, 6, 7, 10], and a more profound comparison among them can be found in the full version [4].

3.1 Same Processes are Invoked



(a) Same Processes (b) New Processes

Figure 1 Hierarchy of definitions when the same processes and new processes are allowed to be invoked after a crash.

In this subsection, we assume that the model allows the same processes to be invoked after a crash. Under this model, the existing definitions can be arranged into the hierarchy that is presented in Figure 1(a). The hierarchy is based on the sets of execution histories that are allowed by each of the definitions; in Figure 1(a), each definition's set of allowed histories is represented by its labelled region.

To understand this hierarchy, it is useful to consider how each correctness condition allows linearizing a given history. The correctness conditions differ by where they allow each pending operation's completion to be placed. Berryhill et al. [5] presented recoverable, persistent, and strict linearizability in this light.

There are several points in a given history with respect to which it may make sense to complete such a pending operation. One point of reference is the crash event that immediately follows inv_{op} in $H|p$. Another is the next invocation by p in the history. Finally, we may also consider the next invocation in H that occurs in the same object as op.

Strict linearizability [1] is the strongest (or *strictest*) condition, in that it allows for the smallest set of histories. It requires every pending operation to be eliminated or completed before the crash. In addition, it is local, meaning that every object that is built from strictly linearizable objects is also strictly linearizable. To achieve this guarantee, one may think of running a recovery operation directly after the crash, and before executing the program. However, it might be too restrictive; in some scenarios, it makes sense to relax this requirement to allow recovery (alternatively; the completion of pending operations) to occur later in the execution.

While strict linearizability requires completions to be placed before the next crash event, persistent atomicity [8] instead completes operations before the next invocation by the same process. Note that, by the definition of legal histories, the next invocation by the same process can never be placed before the current crash, and therefore persistent atomicity is weaker than strict linearizability (i.e. the set of persistent histories *contains* the set of strict histories). Due to this relaxation, it is not local. On the positive side, persistent atomicity may be easier to implement than strict linearizability, since an operation only needs to be recovered (if ever) when the same process invokes another operation.

Berryhill et al. [5] presented the recoverable linearizability definition, which is the most relaxed one. It also requires pending operations to be completed (or removed) before the crash, but practically, to "take effect" before the next invocation of the same process on the same object. Therefore, it allows the most extensive set of histories.

3.2 New processes are invoked

In this subsection, we assume that the model does not allow the same processes to be invoked after a crash, and new processes are spawned instead. This model was first suggested by Izraelevitz et al. [9], as a simplification to previous models. Under this simplification, the definitions that deal with execution continuations do not make sense. The hierarchy in this model is presented in Figure 1(b).

When the same processes are never invoked after a crash, strict linearizability [1] still remains the strongest condition as it requires every pending operation to be eliminated or completed before the crash. Recall that the difference between persistent atomicity [8] and recoverable linearizability [5] is only in recoveries by the same process, and thus these two definitions have the same meaning as durable linearizability [9] which requires getting a linearizable history after removing all crash events from the original history.

By disallowing the executions of the same processes, durable linearizability, persistent atomicity and recoverable linearizability are local under this restriction. Buffered durable linearizability [9] is similar to the others, but additionally allows operations that were completed before the crash to be removed. It therefore is the weakest definition.

References

1. Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.
2. Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 7–16. ACM, 2018.
3. Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264. ACM, 2019.
4. Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions. *arXiv preprint*, 2022. `arXiv:2208.11114`.
5. Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 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. Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM SIGPLAN Notices*, pages 28–40. ACM, 2018.
7. Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. *Distributed Computing*, 32(6):535–564, 2019.
8. Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 400–407. IEEE, 2004.
9. Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.
10. Nan Li and Wojciech Golab. Detectable sequential specifications for recoverable shared objects. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.