The Challenges of Weak Persistency

Viktor Vafeiadis
MPI-SWS, Kaiserslautern, Germany

Abstract

Non-volatile memory (NVM) is a new hardware technology that provides durable storage at performance similar to that of plain volatile RAM. As such, there is a lot of interest in exploiting this technology to improve the performance of existing disk-bound applications and to find new applications for it. Nevertheless, developing correct programs that interact with non-volatile memory is by no means easy, since mainstream architectures provide rather weak persistency semantics and rather low-level and expensive mechanisms in order to avoid weak behaviors. This creates many opportunities for researchers in programming language semantics, logic, and verification to develop techniques to assist programmers writing NVM programs. This short paper and the associated talk outline the challenges caused by NVM and the research opportunities for PL researchers.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Weak Persistency, Non-Volatile Memory

Digital Object Identifier 10.4230/LIPIcs.CALCO.2021.4

Category Invited Talk

Funding My ongoing work on persistency is partly funded from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349, project name PERSIST).

1 Introduction

Until recently, computer storage came in two flavors: (1) volatile storage, such as DRAM, that enables fast byte-sized access but loses its contents upon a power outage, and (2) durable storage, such as hard drives, that preserves its contents upon a power failure but provides only slow block-sized access to data. Due to these characteristics of computer storage, software applications use different mechanisms to access memory and disks.

All this is about to change with the adoption of non-volatile memory (NVM), a recent technology that provides durable storage with performance similar to that of volatile memory. NVM is plugged to the memory bus and can be accessed in byte-sized chunks with latency and throughput slightly worse than those of DRAM (within an order of magnitude). It is thus expected that NVM will soon supplant or even replace volatile memory.

2 Results and Open Questions

Non-volatile memory raises a number of questions relevant for researchers in programming language semantics and verification. Although recent research in this area has scratched the surface of these questions providing some first answers to them, much more research is needed to provide more thorough answers. In the following, I summarize these questions and partial answers in four groups.

First, what *semantics* do programs interacting with non-volatile memory have? Among the multiple ways of defining semantics, to date, only operational and declarative/axiomatic approaches have been applied to model the semantics of persistent programs. Specifically, Raad et al. [8, 9] have developed formal operational and axiomatic models for subsets of the Intel-x86 and the Arm architectures. These models describe the persistency semantics of the core instructions for interacting with non-volatile memory (namely, plain memory accesses and cache-line flush instructions). As can be seen in the cited publications, their semantics is often quite unexpected. Instructions execute asynchronously and even out-of-order, which leads to a number of possible weak persistency behaviors similar in natural to the weakly consistent behaviors observed by concurrent programs running on modern multicore processors. In the future, it would be useful to extend the existing models to cover additional features of the computer architectures, to extensively validate them with respect to hardware implementations, as well as to construct further characterizations of the persistency semantics of NVM programs using, say, denotational semantics with the overall goal at arriving at cleaner definitions.

Second, given that basic memory accesses have weak persistency semantics, how can programmers defend their programs against such weak behaviors, i.e., how can they *avoid weak persistency behaviors*? Currently, the only way to achieve this is to invoke special low-level hardware instructions, such as cache-line flush instructions (on x86 and Arm) or non-temporal store instructions (only on x86). Naively using these instructions, however, does not generate the desired result, because these instructions are weakly ordered with respect to other instructions. One therefore typically has to combine the usage of such instructions with (store) fence instructions to ensure their correct operation. Naturally, it would be desirable to define higher-level architecture-independent mechanisms for persisting stores to non-volatile memory, and to develop correct compilation schemes for these mechanisms to the various different platforms. Beyond higher-level fences and flushes, one can imagine developing even higher-level primitives, such as persistent transactions, persistent data structures.

Third, what are good *correctness criteria* for such persistent data structures and transactions? At a very basic level, one can require some basic algorithm-specific invariants to hold over the persistent state. For example, for a persistent sorted list, one may want to establish that the list is always well-formed in the persistent storage: its links point to valid fully initialized nodes, all such nodes are reachable from the head of the list, and the values of linked nodes are in ascending order. In addition, one may wish to establish an algorithm-independent correctness notion such as persistent serializability or linearizability, roughly stating that the implementation guarantees that the various operations executed in some total order, and that a prefix of that order has persisted.

Finally, given such appropriate correctness notions for persistent algorithms, what *techniques* can we used to establish correctness of such persistent algorithms? The work so far has considered mainly establishing invariants of the persisted state. Two approaches that have been tried are program logics and model checking. Specifically, Raad et al. [6] develop the Persistent Owicki-Gries (POG) program logic, an adaptation of the well-known Owicki-Gries proof system that is able to reason about invariants over the durable state under a subset of the Px86 semantics. In terms of model checking, Gorjiara et al. [4] have developed an approach for verifying assertions (such as invariants) of NVM programs. In a different context, Kokologiannakis et al. [5] developed a model checker for concurrent programs interacting with the ext4 file system, but their technique can easily be re-purposed to verify programs interacting with non-volatile memory. There has also been some works establishing persistent linearizability of simple durable algorithms (e.g., [2, 7]), but clearly much more work is needed to reach the level of complexity and automation that has been achieved for the verification of concurrent algorithms. Obvious next steps are: (1) to consider more advanced program logics, such as GPS [10] and FSL [3], which are extensions of separation logic that handle weak memory consistency, (2) to consider the complexity of basic verification questions following the initial results of Abdulla et al. [1], and (3) to develop automated techniques for checking and/or proving persistent linearizability.

In summary, NVM has created many research opportunities for our community. I intend to explore them in the near future and I hope that other researchers will join me in doing so – whether in collaboration or independently.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. Deciding reachability under persistent x86-tso. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. `doi:10.1145/3434337`.

2 John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Verifying correctness of persistent concurrent data structures: A sound and complete method. *Formal Aspects Comput.*, 33(4):547–573, 2021. `doi:10.1007/s00165-021-00541-8`.

3 Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In *VMCAI 2016*, volume 9583 of *LNCS*, pages 413–430. Springer, 2016. `doi:10.1007/978-3-662-49122-5_20`.

4 Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *ASPLOS 2021*, pages 415–428. ACM, 2021. `doi:10.1145/3445814.3446735`.

5 Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. PerSeVerE: Persistency semantics for verification under ext4. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. `doi:10.1145/3434324`.

6 Azalea Raad, Ori Lahav, and Viktor Vafeiadis. Persistent Owicki-Gries reasoning: A program logic for reasoning about persistent programs on Intel-x86. *Proc. ACM Program. Lang.*, 4(OOPSLA):151:1–151:28, 2020. `doi:10.1145/3428219`.

7 Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model. *Proc. ACM Program. Lang.*, 2(OOPSLA):137:1–137:27, 2018. `doi:10.1145/3276507`.

8 Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. *Proc. ACM Program. Lang.*, 4(POPL):11:1–11:31, 2020. `doi:10.1145/3371079`.

9 Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.*, 3(OOPSLA):135:1–135:27, 2019. `doi:10.1145/3360561`.

10 Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA 2014*, pages 691–707. ACM, 2014. `doi:10.1145/2660193.2660243`.