Object-Oriented Recovery for Non-volatile Memory*

Nachshon Cohen
Amazon
Israel
nachshonc@gmail.com

David T. Aksun
EPFL
Lausanne, Switzerland
david.aksun@epfl.ch

James R. Larus
EPFL
Lausanne, Switzerland
james.larus@epfl.ch

The three-to-five order-of-magnitude performance gap between durable storage (HDDs and SSDs) and volatile memory (DRAM) has resulted in very different interfaces. Persistent data typically is stored in a database or in a file system and accessed through a high-level abstraction such as a query language or API. Due to the high latency of the storage media, these interfaces favor transferring a large quantity of data at each interaction. Programming languages, in contrast, offer direct access to (transient) data in DRAM.

Non-Volatile Memory (NVM) eliminates the performance gap between durable and volatile storage, but using NVM requires changes to programming models. Technologies such as ReRAM [1, 22], PCM [13, 20], and STT-RAM [11] are a new storage media, albeit with an interface and performance characteristics similar to DRAM. These memories retain data after a power shutdown and are likely to be widely deployed in servers in the near future.

NVM requires new programming models to ensure that the persistent storage is left in a recoverable state after an unexpected or abrupt program failure. Durable atomic blocks cause a program to atomically transition between consistent states, each of which can be recorded in NVM and used to restart reliably after a crash [2–6, 8–10, 12, 14, 17, 21, 23, 24].

Capturing a running application's consistent state is, however, only part of recovery. The persistent heap will be used after a restart, which means that it must be put into a state that is consistent in the *new* environment that exists when the application resumes execution. Consider, for example, a persistent key-value store in which each entry contains a pointer to a network connection to a client [16]. The connection is not valid when the program restarts. Recovery should at least discard the old connections to avoid an access to a stale pointer. Sockets, locks, and thread IDs are other, inherently transient fields that are invalid after recovery. These fields are often intermixed with persistent fields in an object, which requires recovery to distinguish the two types of fields and to discard or reinitialize stale, transient values.

Persistent fields containing pointers to other durable objects also become invalid if recovery maps the durable heap to a different location in memory. Currently, operating systems do not guarantee that a persistent heap can be mapped by the `mmap` system call to the same address as before a crash. If the mapping changes, pointers to persistent objects become invalid. Furthermore, program code is also not guaranteed to reside at the same locations as before a crash. Most operating systems, in fact, use address space layout randomization (ASLR) to implement the opposite behavior by placing code at a different virtual address for each execution. Disabling ASLR is possible but reduces security. ASLR affects function pointers (C), virtual table pointers (C++), and read-only data (e.g., string literals).

Fixing these problems after a crash is a significant burden on a programmer. To clear and reinitialize transient fields and update pointers, the programmer must iterate over all live durable objects and update pointers. Failure to find an object (or iterating over an object more than once) is often a subtle error. These operations violate encapsulation as the recovery code must be aware of an object's internal structure (and be updated as an object's fields evolve). Recovery cannot be implemented using an object's methods as the virtual table needs to be updated before these methods are invoked. For these reasons, existing NVM systems generally do not distinguish or reinitialize transient fields and some use self-relative offsets instead of direct pointers [9, 18]. Offsets are more costly and incompatible with standard libraries.[1]

In this paper, we present a C++ language extension for *NVM reconstruction*, the process of reestablishing the consistency of data structures stored in NVM in the environment in which an application is being restarted. `NVMReconstruction` enables a programmer to label a field as transient so that after a restart, the restored instances of this field will be zeroed. More complex recovery is also supported through type-specific recovery methods that can reinitialize transient fields in arbitrary ways. More importantly, programs written with `NVMReconstruction` use conventional data and code pointers, and the system automatically updates these pointers if the location of the persistent heap or the code segment changes.

`NVMReconstruction` also supports upgrading objects. Durable objects are not discarded when an application terminates and they may live for a long time. However, when an application stops and restarts, it can invoke a newer version of the application code. This upgrade can create a mismatch between the new code and the objects in the persistent heap. In particular, the new code may append fields to a class, and an object might expand beyond the space allocated to it. `NVMReconstruction` supports code upgrading by relocating an object to a larger space and redirecting pointers to the object's new location.[2]

To reduce fragmentation of the persistent heap, `NVMReconstruction` implements *offline* NVM compaction. Our compaction algorithm is similar to a (fault-tolerant) copying garbage-collector, but it runs *between* executions of an application, so that when compaction is running, the application is not. Thus, compaction imposes no restrictions on code generation.

`NVMReconstruction` consists of a Clang/LLVM extension and a runtime library. The compiler extension implements our C++ language annotations and emits type information for each class and struct. To track runtime types, our extension modifies the

*Paper appeared in OOPSLA 2018, November 2018. https://doi.org/10.1145/3276523

[1] For C programs, using offsets requires pervasive code modifications at pointer dereferences. For C++, dereferencing can be hidden by operator overloading, but pointer declaration must be modified. In both cases, offsets are slower than direct references.
[2] Other changes, such as deleting or reordering fields, are beyond this work and require application-specific modifications [15].

existing object format by adding an 8-byte header to hold a type identifier. The runtime also records additional execution-specific information in NVM that it uses for reconstruction and compaction. Reconstruction runs concurrently and lazily with an application, which allows an application to restart and respond quickly, without the long latency from updating the entire persistent heap.

1 PRESERVING OBJECT SEMANTICS

In this section, we discuss the challenges in using objects to store durable data. For each of these difficulties, we also describe existing solutions and enumerate their shortcomings.

1.1 Transient Fields

Traditional durable storage interfaces, such as databases, only support durable data and offer no means to distinguish transient data that exists for a single execution of a program. In contrast, when using an NVM heap, durable and transient data will be intermixed.

Consider, for example, locks. A lock is not reusable after the termination of a program. Still, it is a common programming practice to put a lock in an object and to use it to synchronize accesses [7]. Sockets, process (or thread) IDs, and file descriptors are similar transient data that are typically kept with an object.

The presence of transient data in a durable object creates two problems. First, after re-execution, an application might *incorrectly* assume that a transient field is still usable. Second, even if it is possible to detect that a value is invalid, every method must explicitly check if every durable object is well-formed. This introduces runtime overhead and creates unnecessarily complex code.

1.2 Heap and Code Pointer Relocation

Persistent objects containing direct pointers to other persistent objects or to code are vulnerable to the relocation of the heap and of the code segments.

Heap Relocation. The layout of an address space can change when the operating system or libraries are updated or when a program runs under a debugger, profiler, or another runtime tool. The operating system can remap the NVM memory to a different location in virtual memory than the one in the initial execution. If durable data contains direct pointers to persistent data, these pointers are invalidated when NVM memory is mapped to a different location.

Code Relocation. Code segments can also be loaded at a different virtual address than the prior execution when libraries are loaded at different addresses, code is reorganized by address-space layout randomization (ASLR) [19], or a program is modified by a developer.

Direct pointers to methods and functions are common in data structures but forbidden in existing NVM-specific durable storage systems. The most prevalent example of code pointers is the Virtual Table Pointer (VTP) used by C++ to implement virtual methods.

REFERENCES

[1] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830

[2] Joy Arulraj and Andrew Pavlo. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *2015 Int. Conf. Manag. Data*. 707–722. https://doi.org/10.1145/2723372.2749441

[3] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: fast recoverable allocation of non-volatile memory. In *21st Object Oriented Program. Syst. Lang. Appl. - OOPSLA 20116*. ACM, 677–694. https://doi.org/10.1145/2983990.2984019

[4] Dhruva R Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *19st Conf. Object Oriented Program. Syst. Lang. Appl. - OOPSLA 2014*. ACM, 433–452. https://doi.org/10.1145/2660193.2660224 arXiv:2660224

[5] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

[6] Terry Ching, Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *12th Eur. Conf. Comput. Syst. - EuroSys '17*. ACM, 468–482. https://doi.org/10.1145/3064176.3064204

[7] Joel Coburn, Am Caulfield, and A Akel. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *16th Archit. Support Program. Lang. Oper. Syst.- ASPLOS '11*. ACM, 105–118. https://doi.org/10.1145/1961295.1950380

[8] Nachshon Cohen, Michal Friedman, and James R Larus. 2017. Efficient Logging in Non-Volatile Memory by Exploiting Coherency Protocols. *Proc. ACM Program. Lang. - PACMPL* OOPSLA, Article 67 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133891

[9] Krzysztof Czurylo and Andy Rudoff. 2014. NVML: NVM Library. https://github.com/pmem/nvml

[10] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *31st Mass Storage Systems and Technologies - MSST '15*. 1–14. https://doi.org/10.1109/MSST.2015.7208276

[11] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *IEEE Int. Devices Meet.* IEEE, 459–462. https://doi.org/10.1109/IEDM.2005.1609379

[12] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *Annu. Tech. Conf. - ATC 17*. USENIX. http://jinglei.ren.systems/lsnvmm{_}atc17.pdf

[13] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *36th Int. Symp. Comput. Archit. - ISCA '09*. ACM, 2. https://doi.org/10.1145/1555754.1555758

[14] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *22nd Archit. Support Program. Lang. Oper. Syst. - ASPLOS '17*. ACM, 329–343. https://doi.org/10.1145/3037697.3037714

[15] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. 2012. Automating object transformations for dynamic software updating. In *Object Oriented Program. Syst. Lang. Appl. - OOPSLA '12*. ACM Press, New York, New York, USA, 265. https://doi.org/10.1145/2384616.2384636

[16] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *HotStorage '17*. USENIX. https://www.usenix.org/system/files/conference/hotstorage17/hotstorage17-paper-marathe.pdf

[17] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *12th Eur. Conf. Comput. Syst. - EuroSys '17*. ACM, 499–512. https://doi.org/10.1145/3064176.3064215

[18] Oracle Corporation. 2017. NVM Direct Library. http://www.oracle.com/technetwork/oracle-labs/open-nvm-download-2440119.html

[19] Team PaX. 2003. PaX address space layout randomization (ASLR).

[20] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *36st Int. Symp. Comput. Archit. - ISCA '09*. ACM, 24–33. https://doi.org/10.1145/1555754.1555760

[21] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *16th Archit. Support Program. Lang. Oper. Syst. - ASPLOS '11*. ACM, 91–104. https://doi.org/10.1145/1950365.1950379

[22] H. . P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. 2012. Metal-Oxide RRAM. *Proc. IEEE* 100, 6 (June 2012), 1951–1970. https://doi.org/10.1109/JPROC.2012.2190369

[23] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th File and Storage Tech. - FAST '15*. USENIX, 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[24] Jie Zhou, Yanyan Shen, Sumin Li, and Linpeng Huang. 2016. Revisiting Hash Table Design for Phase Change Memory. In *3rd IEEE/ACM Big Data Comput. Appl. Technol. - BDCAT '16*. ACM, 227–236. https://doi.org/10.1145/3006299.3006318