Characterizing non-volatile memory transactional systems

Pradeep Fernando[1], Irina Calciu[2], Jayneel Gandhi[2], Aasheesh Kolli[3], Ada Gavrilovska[1]
[1]Georgia Tech, [2]VMware Research, [3]Penn State

I. Introduction

Emerging Non-Volatile Memory (NVM) technologies, like Intel's DC Optane persistent memory, offer byte-addressability and orders of magnitude faster access to storage than traditional storage technologies. Their key appeal is that they allow applications to access storage directly using processor load and store instructions rather than relying on a software intermediary like the file system or a database [1]. However, ensuring that data stored in NVM is always in a safe and recoverable state is both hard and incurs performance overheads [1]–[3].

To ensure data recoverability, application developers have to carefully orchestrate data movement from the volatile to the persistent components in the memory hierarchy, subject to application-specific constraints. This task is especially complex due to two factors: (1) NVM applications have very diverse crash-consistency requirements [4]; and (2) the *persistence domain* is different across platforms. For example, Intel and Micron guarantee that data becomes persistent only when it reaches the memory controller of the NVM device, i.e., the persistence domain of the system includes the memory controller and the NVM devices [5]. We refer to such systems as having *transient caches*. However, HPE's NVM [6] guarantees that the entire cache hierarchy is persistent, i.e., the persistence domain includes the entire memory hierarchy. We refer to such systems as having *persistent caches*.

In this context, researchers have proposed various transactional systems that provide the well known "ACID" guarantees for NVM applications. These transactional systems significantly simplify NVM application development and leave the complexities of achieving data recoverability on various platforms to the low-level systems software developers. While these systems all provide ACID guarantees, they go about providing these guarantees in different ways: UNDO vs. REDO logging, software vs. hardware transactions. Low-level developers designing ACID transaction systems face a bewildering array of choices, with varied performance characteristics that change with the applications and the platform used. For these developers, we aim to answer the question: **how to quickly explore the design space and arrive at a correct and high-performance implementation of an NVM transactional system?**

Reasoning about implementation details rather than the overall guarantees provided to the user (ACID) helps transaction system developers traverse the design-space more efficiently. To provide ACID guarantees, the underlying transaction system has to correctly ensure three properties: (1) *crash consistency* - individual transactions are failure-atomic, i.e., after a crash, either all or none of the transaction has persisted, (2) *synchronization* - transactions are correctly isolated from other transactions executed on different threads, and (3) *composability* - the crash consistency and synchronization techniques used compose to provide the overall ACID guarantees, by ensuring that dependent transactions are correctly ordered.

This new characterization of transaction systems provides a basis to compare different implementations and to identify the right set of `crash-consistency` and `synchronization` mechanisms for particular applications and hardware platforms. We perform a detailed characterization study of systems with different implementations (hardware transactional memory (`HTM`) [7], software transactional memory (`STM`) [3], and `undo`/`redo` logging with locks [3], [8]) under various persistence domains (transient vs. persistent caches). We perform our study on real hardware using the recently released Intel's DC Optane Persistent Memory [9] and using simulation.

Our empirical study results in several interesting insights for NVM transaction system developers:

1) For all applications, the persistence domain plays the most important role. The overhead of making transactions persistent is considerably lower when caches are persistent.
2) In systems with transient caches, HTM is the best choice, despite its synchronization costs and required architectural changes. This is due to the high overheads caused by flush and fence instructions required by undo/redo logs, which are elided by HTM. The choice between undo and redo logs depends on the application characteristics and the size of the read and write sets of the transactions.
3) In systems with persistent caches, the HTM does not require any architectural changes, but its benefit for supporting persistent transactions is reduced, as software logging mechanisms do not require expensive flush and fence instructions anymore. Here, undo logs are the best choice because redo logs suffer from read-indirection overheads.
4) The overheads of crash-consistency for an HTM are subsumed by synchronization overheads. As applications scale, performance increases despite crash-consistency overheads. When the crash-consistent HTM does not achieve scalability due to aborts, crash-consistent STM ensures this property.

Overall, this presentation will illustrate the following:

- We characterize persistent transactions to quickly and methodically compare different implementations of NVM transactional systems that provide ACID guarantees.
- Using this new characterization, we study the performance of various transaction system implementations on different hardware platforms and for different applications.
- We show that there is no one best way to provide ACID guarantees for NVM applications; the best way changes with hardware platforms and application characteristics.
- Finally, we believe we are the first work to evaluate the range of these different transactional systems on real NVM devices.

II. Key insights

In this work, we focus on applications that use a transactional programming model to get ACID guarantees. For example, updates within each transaction need to provide all or nothing semantics when the data gets to NVM. Providing ACID guarantees requires that the transactional system correctly implement three components: (1) *crash-consistency* (also called failure-atomicity), which ensures all-or-nothing behavior of uncommitted transactions when a failure happens and the validity of the data after the failure (atomicity

and consistency) (2) *synchronization*, which ensures that partial updates are not observable by other concurrently running transactions (isolation), and (3) *persistence* of the committed transactions in the correct order, which ensures that committed transaction are made durable and that the correct dependencies between transactions are maintained (durability). Note that crash-consistency is a property of uncommitted transactions, which guarantees that on a failure, a transaction will either abort, leaving no side-effects, or will commit, finishing its entire execution. In contrast, persistence is a property of committed transactions, guaranteeing their permanence in case of a crash, as well as that dependent transactions' effects are all visible in the correct order. We call a correct implementation of the above three properties that ensures ACID guarantees `crash-sync-safe`.

As illustrated in Table I, developers have a wide variety of choices for `crash-sync-safe` transactions, and choosing between these different options depends on a variety of factors, such as the persistence domain, and the application characteristics. To further complicate matters, some mechanisms offer some of the guarantees, but not all, and developers need to carefully mix and match techniques to ensure correctness. For example, `undo` and `redo` logging can be used to implement crash-consistent transactions for single-thread applications, but do not ensure the correct synchronization of multi-threaded applications, forgoing isolation. Conversely, locking can be used to provide correct synchronization for multi-threaded applications, but cannot ensure persistence for these transactions in case of a failure, forgoing durability, nor crash-consistency, forgoing atomicity and consistency. Transactional memory provides correct synchronization for multi-threaded applications, as well as atomicity and consistency, but cannot ensure persistence for these transactions in case of a failure, forgoing durability.

In this presentation, we provide a comprehensive evaluation of the impact of combining existing crash-consistency and synchronization methods for achieving performant and correct NVM transactional systems. We consider different hardware characteristics, in terms of support for hardware transactional memory (HTM) and the boundaries of the persistence domain (transient or persistent caches). By characterizing persistent transactional systems in terms of their properties, we make it possible to better understand the tradeoffs of different implementations and to arrive at better design choices for providing ACID guarantees. We use both real hardware with Intel Optane DC persistent memory and simulation to evaluate a persistent version of hardware transactional memory, a persistent version of software transactional memory, and undo/redo logging. Figure 1 illustrates a representative results from our investigation. Through our empirical study, we show two major factors that impact the cost of supporting persistence in transactional systems: the persistence domain (transient or persistent caches) and application characteristics, such as transaction size and parallelism.

	ST – CC		MT – Sync.	MT – CSS	
	TC	PC		TC	PC
seq	✗	✗	✗	✗	✗
HTM+seq (+spinlock)	✗	✗	✓	✗	✗
undo/redo (+spinlock)	✓	✓	✓	✓	✓
HTM+undo/redo (+spinlock)	approx.	✓	✓	approx.	✓
ccHTM+undo/redo (+spinlock)	✓	N/A	✓	✓	N/A
STM	✗	✓	✓	✗	✓
ccSTM	✓	N/A	✓	✓	N/A

TABLE I
CRASH-CONSISTENCY AND CRASH-SYNC-SAFETY IMPLEMENTATIONS FOR SINGLE- AND MULTI-THREADED APPLICATIONS. ST: SINGLE-THREADED, MT: MULTI-THREADED, CC: CRASH-CONSISTENT, SYNC: SYNCHRONIZATION, CSS: CRASH-SYNC-SAFETY, TC: TRANSIENT CACHES AND PT: PERSISTENT CACHES. TECHNIQUES EVALUATED FOR SINGLE-THREADED APPLICATIONS NEED TO PROVIDE ONLY CRASH CONSISTENCY. TECHNIQUES EVALUATED FOR MULTI-THREADED APPLICATIONS PROVIDE SYNCHRONIZATION TOO, BY USING A SPINLOCK WHERE NECESSARY. WE NOTE THAT THE HTM+UNDO/REDO IMPLEMENTATIONS FOR TRANSIENT CACHES ARE ONLY APPROXIMATING A CRASH-SYNC-SAFE SOLUTION.

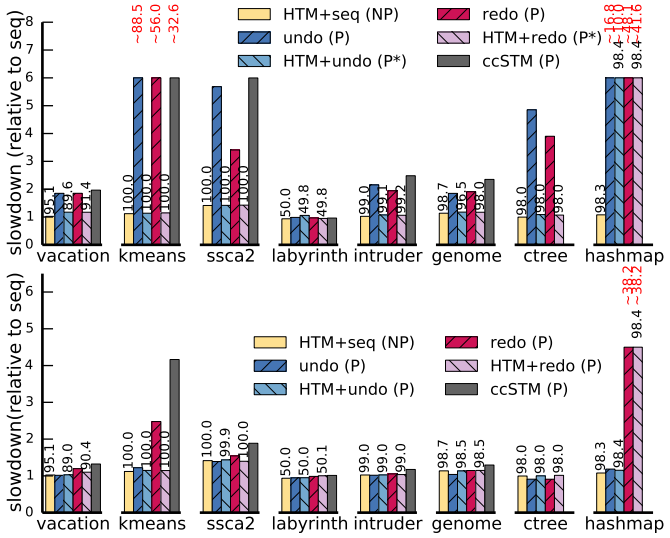

Fig. 1. TSX-enabled hardware with real NVM, with transient (top graph) and emulated persistent (bottom graph) caches. We show transaction success for methods using HTM (values in black). We truncate large bars in hashmap (values in red). (P) crash-consistent; (NP) not crash-consistent.

REFERENCES

[1] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage Management in the NVRAM Era," *PVLDB*, vol. 7, no. 2, pp. 121–132, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol7/p121-pelley.pdf

[2] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

[3] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[4] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 135–148.

[5] A. M. Rudoff, "Deprecating the pcommit instruction," https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016.

[6] "What's in HPE's persistent memory?" retrieved from https://www.pcworld.com/article/3051133/whats-in-hpes-persistent-memory.html, 8 April 2016.

[7] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 452–465.

[8] "Pmdk," "https://pmem.io/pmdk/".

[9] "Big memory breakthrough for your biggest data challenges," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2019.