Failure Recovery from Persistent Memory in Paxos-based State Machine Replication

Jan Kończak
Institute of Computing Science
Poznan University of Technology
Poznań, Poland
Jan.Konczak@cs.put.edu.pl

Paweł T. Wojciechowski
Institute of Computing Science
Poznan University of Technology
Poznań, Poland
Pawel.T.Wojciechowski@cs.put.edu.pl

***Abstract*—Paxos is one of the most popular protocols for state machine replication (a technique used for making services highly available). We are the first to propose a Paxos-based state machine replication framework which is aimed at persistent (non-volatile) memory, pmem in short—a new class of memory offering direct byte-addressable access to memory (e.g., Optane™ DC Persistent Memory). In the paper, we describe two variants of the framework, called mPaxosSM and mPaxos, which support efficient recovery of processes after crash with the use of pmem. In the latter variant, a part of Paxos's state, and in the former also the entire state machine's state that should survive crashes, are stored in the persistent memory. This allows to achieve low failure recovery time. We used a key-value map to compare our frameworks equipped with different memory backends (pmem, DRAM, and emulated pmem), with the classical Paxos that recovers state from snapshots and logs stored in stable storage, and with Paxos equipped with EpochSS—a state-of-the-art protocol ensuring state recovery from peer replicas. Our results show the advantages of pmem and our approach.**

***Index Terms*—state machine replication, Paxos, persistent memory**

I. Introduction

State machine replication (SMR) [1], [2] is a well-known technique for developing distributed services requiring high availability. Paxos [3] is one of the most popular SMR protocols, used in many commercial systems, including data management systems. *Persistent (non-volatile) memories (pmem)*, connected to the system memory bus (e.g. DDR4), such as Intel's Optane™ DC Persistent Memory (DCPMM) [4] based on the 3D XPoint technology [5], offer the durability of disk, latency comparable to DRAM, high throughput, and a better density than DRAM. These properties have spawned a lot of efforts to adopt pmem in computer systems. Persistent memory is exposed by operating systems as memory-mapped files. Ensuring recoverability of replicated state machine from pmem rather than from disks can significantly boost performance. However, our experience has shown that there is much more to developing a pmem-enabled replication engine than simply replacing disk writes by pmem writes.

In this paper we propose two novel Paxos-based SMR tools, which are aimed at computer systems equipped with pmem. To achieve this, we had to rethink the design of Paxos-based state machine replication. In *mPaxosSM*, that part of the state of state machine (application) and of Paxos that should survive crashes is stored in pmem. This allowed us to simplify the design of SMR, as creating *periodic* state snapshots is no more necessary. In *mPaxos*, the whole application state is in DRAM, but snapshots (created by the application) and some part of Paxos state are kept in pmem. Both tools can be used to build replicated systems that after crash can resume operation immediately after a sufficient number of replicas is restarted.

We used mPaxosSM and mPaxos to replicate a key-value map (let us call it *KV-Map*) and compared its performance with the replicated map built using *JPaxos+SS* and *JPaxos+epochs*, described in [6]. The former system implements the original Paxos protocol with the use of *stable storage (SS)* to recover. The latter is Paxos equipped with *EpochSS* [6], a state-of-the-art recovery algorithm that can recover crashed replicas from live peers' snapshots, if only a majority of replicas remains operational. All evaluated systems share a common code base that implements a generic *Paxos framework*, which extends the original Paxos with practical features, e.g., operation batching, snapshots, and the catch-up protocol. For a fair comparison of all systems, as stable storage in JPaxos+SS, we have not only used *solid state drives (SSDs)* but, first of all, persistent memory (pmem), accessed using the standard file API and `fsync()`. On the other hand, mPaxos and mPaxosSM write to persistent memory directly from userspace.

KV-Map using mPaxos is more efficient than the one using JPaxos+SS during the regular (failure free) execution (unless the operations or the snapshots are large, then the two systems are on par), and it also faster recovers after crash. However, in our 10GbE network, the DRAM-only system built using JPaxos+epochs outperforms both. But unlike JPaxos+epochs, all the remaining systems can tolerate *catastrophic failures* (crashes of all nodes), which is a great advantage.

What came as a surprise, the KV-Map with mPaxosSM was the slowest system out of all during the regular (failure free) execution (though the fastest system if pmem was replaced by DRAM, but then it cannot tolerate failures). This result is contributed by the overhead incurred by transactions (required for atomic writes to pmem) and low write throughput of DCPMM. But the recovery performance of mPaxosSM is

This work was supported by the Foundation for Polish Science, within the TEAM programme co-financed by the European Union under the European Regional Development Fund, Grant No. POIR.04.04.00-00-5C5B/17-00.

incomparably higher than in case of JPaxos+SS and mPaxos, as the application state in pmem is immediately ready for use.

Section II defines the system model and Paxos-based SMR. Section III describes the Paxos framework to a level of detail sufficient to understand failure recovery mechanisms. Section IV presents two variants of the framework aimed at persistent memory. Section V discusses failure recovery. Section VI contains experimental results. Section VII considers related work and Section VIII summarizes and concludes the paper.

II. System Model and Paxos-based SMR

We assume a distributed environment in which processes communicate solely by exchanging messages, the network is *asynchronous* (no common clocks, no way to know how long a message will take to get from a to b), the network may drop messages, processes may crash, the crash failures cannot be detected reliably, and there is no single point of failure. We do not consider Byzantine failures.

A *state machine (SM)* executes an operation by changing its state and producing a response, with the operation and machine's current state determining the new state and the response. In state machine replication, all state machines execute the same sequence of deterministic operations[1]. The expected guarantee is *linearizability* [8], which essentially means that all operations return responses that are consistent with some serial execution of these operations that respects the real time order of the operations.

Paxos [3] is a consensus algorithm aimed for SMR, executed by a set of processes, named *replicas*, to reach consensus and *agree* on (*decide*) a single operation by *votings*. An *operation* is a command or a sequence of commands that form an input to the state machine of every replica. If a majority of the replicas run for long enough without crashing and there are no other failures, all running replicas are guaranteed to eventually agree on one of the operations submitted, which is then executed by the state machines. We assume that crashed replicas may subsequently recover under the same ID.

The Paxos algorithm consists of two phases, which may be repeated (because of failures). A very simplistic description follows. In Phase 1, called *leader election*, a process that wants to be a *leader*—a *proposer*, selects a *ballot number*, writes it to stable storage, and broadcasts it using a Prepare message. Other replicas write the ballot number to stable storage and acknowledge it using PrepareOK. In Phase 2, called *voting*, the leader selects an operation, writes it to stable storage, and then broadcasts it using a Propose message. The replicas that receive the proposal (the *followers*) write it to stable storage and then acknowledge the Propose.

In our implementation, replicas do not acknowledge just the leader in Phase 2, but they broadcast an Accept message. Once a replica receives both the Propose and the Accept messages from a majority of all processes, consensus has been reached, and the replica *numbers* the operation and outputs it to its state machine for execution. Operations are then executed by the state machine according to the *operations' numbers*. Obtaining Propose is necessary, as the Accept messages do not carry operations, just instance numbers (defined below).

[1]In Chubby [7], *leases* are used to allow reads from one SM replica only. In our KV-Map, for simplicity, all operations are executed by all SM replicas.

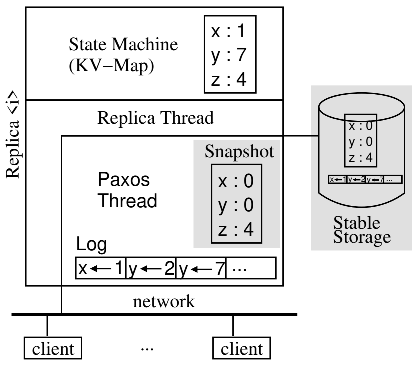

Fig. 1. The architecture of the JPaxos+SS system. mPaxos and mPaxosSM do *not* use stable storage, mPaxosSM does *not* create periodic snapshots, and JPaxos+epochs uses stable storage *only* on system startup.

In SMR, state machines repeatedly execute Paxos to achieve consensus on a *sequence of operations* (aka *Multi-Paxos*). We call each such execution an *instance* of Paxos. There can be a number of instances executing Phase 2 concurrently (see [9]). A leader assigns the lowest *instance number* that is still free to choose for each new Paxos instance. An instance is *decided* when the consensus has been reached. In our implementation, several operations are *batched* by the leader into a single Paxos instance for efficiency (the size of the batch is variable). In some failure or execution scenarios, some operations can be voted in more than one instance, e.g., when a client resubmits an operation to a second replica falsely assuming that the first replica has crashed. We detect such repeated, agreed operations and discard them just before assigning operations' numbers.

III. The Paxos framework

The original papers on Paxos are silent on some important design choices that impact system efficiency, and the existing implementations differ in this respect. With the advent of pmem these choices are even more important, as proved by our results. Below we describe the mechanism of logs, snapshots and the catch-up protocol in our generic *Paxos framework*, which then will be tailored for pmem in the following sections. It consists of two main parts (see Fig. 1): a *Paxos thread*, which is responsible for agreeing operations in consecutive instances, and a *Replica thread* (in fact, a multiple of threads for efficiency), which receives new operations from clients, passes the agreed operations for execution by the state machine, returns the results to the clients, and manages snapshots created by the application.

A. The Log

In our framework, a *log* is a sequence of entries for consecutive Paxos instances, which is updated by the Paxos thread on a regular basis, where each entry contains the following data: an instance state (indicating if the replica got the Propose and if the instance is decided), the number of the last ballot in which the replica cast a vote (in this instance), the last batch of operations seen by the replica (in this instance), and a list of processes L_{Accept} from which an Accept message was delivered (in this instance), together with the ballot number of the Accept messages (we delete records in L_{Accept} with a lower ballot number). The Replica thread passes the operations from instances marked as decided to the state machine for execution, according to the natural order of instance numbers and the position in the batch. Each operation passed to the state machine is assigned a consecutive operation number.

Paxos requires that some essential data of the Paxos protocol which are written to the log must be made *persistent*—such log writes are flushed to stable storage before the system can proceed any further, where *stable storage (SS)* is traditionally a disk, but it can also be pmem. The log written to stable storage is used by a replica to recover its state after crash. In our implementation, followers write to this log prior to sending the Accept message, and the leader writes to it immediately before sending the Propose (it is necessary as the leader does not send itself a separate Accept message). Moreover, a process writes to the persistent log each ballot number which is higher than any other it knew. It is either the one it intends to send in the Prepare message, or which the process learnt from the PrepareOK or other messages described in the paper.

In our experiments, JPaxos+SS uses either disk or pmem as stable storage. mPaxos and mPaxosSM store the log exclusively in pmem. In JPaxos+epochs, the log is entirely maintained in DRAM, and a replica writes to stable storage only once—at startup and at every replica restart after crash. We describe the details of recovery algorithms in Section V.

B. Snapshots

The repeated execution of Paxos leads to an ever growing log. This requires unbounded amount of disk space and may result in unbounded recovery time, since a recovering replica has to replay a potentially long log before it has fully caught up with other replicas. Both space and time overruns are not acceptable, so we periodically create a *snapshot* of the current state of the application, extended with Paxos metadata, and write it to stable storage (JPaxos+SS) or to pmem (mPaxos), or keep in volatile DRAM only (JPaxos+epochs). From this moment any older entries in the log can be erased, and any subsequent snapshot replaces the previous one.

However, creating snapshots is time consuming. Therefore we also designed mPaxosSM. It does not create *periodic* snapshots for failure recovery, as the critical state of the application and of Paxos is stored in persistent memory, hence the state is immediately available after crash. Thus, the log can be truncated at will. But mPaxosSM still needs to create snapshots *on demand*, as required by the catch-up protocol.

As Paxos does not know anything about the application data structures that are replicated, the state machine (application) must be responsible for taking snapshots. The application is free to take a snapshot at any point. But the Paxos framework maintains the log, so it can know better when to take a snapshot. In our implementation, when the size of the log divided by the average size of snapshots is greater than or equal to a configurable threshold, the framework *suggests* the application to take a snapshot. Snapshots are not synchronized across replicas and each state machine independently decides when to create a snapshot, based on the received hint.

The application takes a snapshot of its current state, and informs the Paxos framework that a new snapshot was created which reflects the state of state machine just after executing an operation numbered l. Then the Paxos framework extends the snapshot with the following metadata: l, the number i of the instance in which this operation was decided, the number k of the first operation in the batch of operations in that instance, and the *Responses* map[2]. Thus, our system can take snapshots after executing any operation. After crash, it can recover its state from the snapshot and log, and resume execution from the instance i by omitting first $l-k+1$ operations in i, as executed. To find an instance number for a given operation number, our implementation uses a separate list with instance numbers paired with their first operation number.

Upon each new snapshot is created in DRAM, it is flushed to stable storage (except for JPaxos+epochs, where both the log and the snapshot are in DRAM only, and mPaxosSM, where the application's critical state is stored in pmem). Then log entries containing instances earlier than the *previous snapshot*, including that old snapshot, are erased. We still keep in the log the old operations (with their Paxos-specific metadata) that were agreed after the previous snapshot was taken and before the latest snapshot was taken, as they can help other replicas to catch up efficiently (see Section III-C for details).

C. Catching Up

In Multi-Paxos some slow (or *lagging*) replicas might not have participated in recent Paxos instances. This may occur as consensus is reached once a leader receives messages from at least a majority of the replicas. Some messages can be slow or lost due to network failures. In the original Paxos [3], a replica that did not receive a message for a sufficiently long time can propose itself as a new leader. However, frequent leader changes are inefficient and thus should be avoided. In our implementation, we use a *catch-up* protocol to enable lagging replicas to catch up with *leading* replicas. It uses the log and snapshot of leading replicas to help lagging replicas to catch up, as explained below. It is also used by recovery algorithms, described in Section V.

Any replica should start the catch-up protocol once it notices that it *lags behind*, i.e., it does not know operations for an instance number that is lower (by a configurable offset) than an instance number i received in a *heartbeat* message of a failure detector[3], or a Propose/Accept message, or a RecoveryAck message (defined in Section V-C). To start catch-up, the lagging replica sends a message with all instance

[2] Each replica for every client caches the last response sent to the client. If a client retries sending an operation that was already executed, the operation is not re-executed, but the cached response is returned to the client.

[3] The leader periodically broadcasts a heartbeat message to inform other replicas that it is still alive (aka *failure detector*), to prevent the leader change.

numbers less than or equal to i for which it does not know operations (we call such instances the *missing*), to any replica which is not a leader[4]. Then that replica replies by sending a message with its latest snapshot if there were any missing instances with a number lower than the lowest instance number recorded in its log, and with the current ballot number and all the remaining missing instances (pairs of an instance number with its corresponding operations) that are marked in the log as "decided". Note that the replica does not need to send a snapshot (which can be large), if the lagging replica does not know only those instances which are stored in the log. We therefore truncate the log by the one-but-last snapshot, not the last one.

Once the lagging replica receives the reply it installs the received snapshot (if any and if still necessary), replaces its old snapshot with the new one, updates its current ballot number (if lower then the received one), and then updates its log with the received decided instances (if still necessary). If some operations are still missing, it repeats the protocol, contacting the leader.

IV. Persistent Memory Aware SMR Tools

The mPaxos and mPaxosSM systems use the *Direct Access (DAX)* mechanism to access persistent memory. The DAX feature allows accessing and flushing data to pmem without the need to involve an operating system, as the memory is accessed with no caches in DRAM (unlike traditional block devices). So, a program can execute a CPU store instruction followed by the `sfence` and `clwb` instructions to persist data, rather than executing a `write` syscall followed by a `fsync` syscall. To use DAX, the system administrator creates a file system on a named persistent memory address range (namespace) and mounts that file system into the operating system's file system tree, and applications use `mmap` (a memory map syscall) to attach a region of pmem to its virtual memory.

To implement the systems, we used libpmemobj-cpp, a C++ library for persistent memory programming, which is part of *Persistent Memory Development Kit (PMDK)* [10]. The libpmemobj-cpp library allows persistent memory *objects* to be allocated in a way that is power fail safe, allows referring to them by special pointers (offsets from the beginning of named memory pools), and allows making an arbitrary number of changes *atomic* by encompassing the changes in a *transaction*. If the transaction is interrupted by a power failure or a program crash before it starts committing, any partially done changes are automatically rolled back (on restart), and when it is interrupted during commit, the commit is automatically repeated on restart. We used transactions extensively to ensure the state is always consistent, e.g., data received in a message are written to pmem as a transaction.

Obviously, we also use standard read-write locks to protect objects and prevent the changes made by one thread from becoming *visible* by other threads until the changes are complete.

[4]The leader has the most up-to-date knowledge, but it is also the most busy replica, so we avoid burdening it with additional work.

A. mPaxos: SMR with Paxos in pmem

The design of mPaxos follows closely JPaxos+SS, that is, Paxos equipped with the logs, snapshots, and catch-up as described in Section III, but the log and snapshot reside in persistent memory, not in DRAM, which means that no data are written additionally to other stable storage (e.g., a disk), as it is not necessary. The following data are maintained up to date in persistent memory by the Paxos thread (we omit auxiliary variables):

- snapshots of the application's state, extended with data that the Replica thread maintains (explained below),
- the log, with the content as described in Section III-A,
- a bounded list of log entries that predate the last snapshot (to optimize the catch up protocol),
- the current ballot number, and the state of a proposer (inactive, executing Phase 1, or elected as a new leader).

The Paxos (producer) and Replica (consumer) threads share in pmem a producer-consumer queue (list) of instance numbers which were decided, but not yet executed. Decided instances are added to this queue on a regular basis (irrespective of snapshots' creation). When a new snapshot is created, all instances having all operations executed before the snapshot was created are removed from the queue. On every snapshot creation, the Replica thread updates the following variables that are also stored in pmem: the number of the next instance and the next operation to be executed by the state machine, and the Responses map.

Persistent memory speeds up system recovery after crash. There is no need to read snapshots and logs from a disk, and there is no need to recreate the state of Paxos, as this state is already in the memory ready for use. But pmem is slower than DRAM, so we chose which data to store in pmem considering what is necessary to ensure data consistency in case of failures, and high system performance during failure free runs.

B. mPaxosSM: SMR with Paxos and SM in pmem

We took a different approach when designing mPaxosSM. Since pmem has matured into a fairly complete programming model, and libraries have been built on that basic model to provide application developers with the benefits of persistent memory, so we propose that the developers design their applications in such a way that they maintain the state necessary for failure recovery in pmem. In case of a key-value map, such as our KV-Map, this state includes the content of the map.

In mPaxosSM, a state machine (application) does not create periodically snapshots of its state, as the state machine is able to restart after a crash and *resume* the execution based on the state stored in pmem that survives the crash (in Section V-B, we give more details). The thread executing the Paxos protocol stores the same data in pmem as it was in mPaxos, except that there are no snapshots. The log is truncated at will, when the Paxos framework decides to do it.

However, a replica is still obliged to create a snapshot *on demand*, when it is asked for it by a lagging replica executing the catch up protocol. For this, the Paxos framework simply requests the application to return the name of the file storing critical data in pmem and to lock computation, so that it can create a consistent snapshot. Once the snapshot is created, the computation is released. The lagging replica executes the reversed procedure to install the *catch-up snapshot* back to pmem. To support the mechanism we extended SMR's API.

The Replica thread modifies on a regular basis in pmem: the number of the next instance and of the next operation to be executed by the state machine, a queue with the numbers of instances decided, but not yet executed, the Responses map, a list of files created from the snapshot received during catch up, and a flag signaling if the catch-up snapshot is being installed.

All evaluated systems were implemented in Java, but the pmem library is in C++. To avoid costly calls to *Java Native Interface (JNI)* in mPaxosSM and mPaxos, the following data are cached: ballot number, the lowest undecided instance number, the proposer state, the number of the highest instance in the log, and the numbers of the next instance and operation to be executed by the state machine.

mPaxosSM facilitates a simpler design of applications (the programmers do not need to create snapshots), and the instant time of system recovery after crash.

V. Recovering from Failures

The recovering replica either rebuilds its state from a snapshot and log (JPaxos+SS), that are persisted in SS (typically disks, but it can be pmem, as in our case), or it simply *resumes* the execution from pmem (mPaxosSM and mPaxos), or from other replicas (JPaxos+epochs), as explained below.

A. Recovery from State Read from Stable Storage

In Lamport's Paxos [3], when a replica crashes and subsequently *recovers*, it replays the persistent log read from stable storage to reconstruct its state prior to crashing. After this, the process does not differ from lagging processes, so it can reply to Paxos messages and catch up current state as usual.

The above mechanism can be optimized using snapshots, as mentioned in [3]. Should the replica fail, during subsequent recovery it will install the latest snapshot and then replay the (truncated) log—both read from stable storage—to rebuild its state. From now on, the replica is ready to reply to Paxos messages. After reading a snapshot, Paxos-specific information from the snapshot is returned to the Paxos framework, which in turn will use the information to coordinate the snapshot with the log. In particular, the operation numbers allow to recognize the moment when the snapshot was created.

In our implementation, to catch up with other replicas, the recovered replica launches the catch-up protocol. As in case of non-crashed, lagging replicas, this occurs when the replica realizes that it lags behind, by comparing its last instance number (the recovered replica read it from its local disk) with the instance number i, carried by either the Propose, Accept, or the heartbeat message of the failure detector that it first receives. If i is higher than expected it starts catch-up.

In [6], the above recovery algorithm is called *FullSS*, as it fully depends on data read from stable storage (a disk or pmem). The algorithm is used by JPaxos+SS.

B. Resumption from Persistent Memory

When designing mPaxos and mPaxosSM, we took a different approach, as these systems are aimed to take the best of byte-addressable pmem. So there is no notion of *state recovery* from stable storage or from other replica, since all critical data of Paxos, and in mPaxosSM also the critical state of the application, are maintained in pmem only. Thus, a replica restarted after crash can simply resume execution from the moment in which the system halted. This has a tremendous impact on Paxos behavior. While JPaxos+SS and JPaxos+epochs after crash and restart must first recover their state from stable storage or peers, before they can reply to Paxos messages, mPaxos and mPaxosSM can immediately participate in the Paxos protocol. The main difference between the last two systems is that mPaxos (the Paxos thread, to be precise) has to recreate the state of application in DRAM from a last snapshot stored in pmem, while mPaxosSM does not do it, as the application itself keeps its own critical state in persistent memory, so it can survive crashes.

Note that in the original Paxos, after a catastrophic failure, restarted processes must repeat all undecided instances, while in mPaxos and mPaxosSM, processes store in pmem the full information about every Paxos instance, including its state and the list of the received Accepts, and also whether the process has already received the PrepareOKs from a majority. Thus, a process can immediately continue voting or receiving Accepts or sending the Propose messages (if the process was a leader at crash).

However, there is one pitfall. If a replica crashed while executing some operation, mPaxosSM does not know after recovery if the last operation was executed completely or not. Therefore it has to again issue the operation for execution with the associated number. The application knows the number of the last executed operation, so it can recognize that the operation is repeated and has to decide whether to execute it or not. Note that the application programmer is responsible to get it right, as the decision depends on the semantics of operations. For example, in case of our KV-Map, reexecuting the same write (or the same read) right after the first execution is not harmful, but an operation "increment a counter" cannot be repeated.

C. Recovery from State Received from Other Replicas

Let us consider *peer-based recovery algorithms* for Paxos, that do not require that replicas write the state of state machine and Paxos to storage that survives replica crashes. The crashed process can recover its state from failure by reading the missing state from another replica, if only a majority of processes is always up and running. Such recovery algorithms are not trivial, as they must prevent a situation when two groups of processes decide differently, because the recovered process did not learn all of its state lost at crash. Below we show an example in Phase 2 (voting) of the Paxos protocol, but analogous scenarios may occur in Phase 1 (leader election).

Consider three replicas (see Fig. 2). A follower b broadcasts Accept in an instance i and crashes. After restart, it does

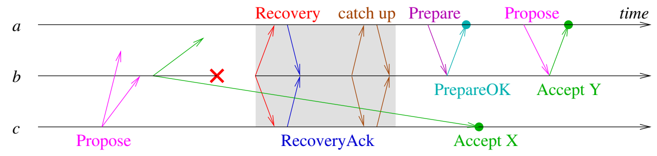

Fig. 2. An example of incorrect process recovery leading to inconsistent decisions caused by a stray message. Messages not shown are lost or in transit.

not remember that it has already accepted some operation X in this instance before the crash, because it did not write to stable storage before sending Accept X. Moreover, it will not learn about this decision by asking the other processes for the missing state—the other follower a did not receive Propose and Accept yet, and the leader c did not receive Accept yet (the messages are still in transit). If in the meantime a becomes a leader, it may form a quorum with the recovered process b and decide a different operation Y in i than the old leader c which will eventually receive the Accept X sent before the crash and decide X in i, which causes inconsistency. Therefore messages that were sent but not delivered before a crash, called *stray*, must be handled properly.

In [6], we show two peer-based recovery algorithms, called ViewSS and EpochSS. In our experimental evaluation, we used the JPaxos+epochs framework, which uses the latter that is more efficient. EpochSS requires that each process maintains in stable storage a counter, which is incremented only when the process starts or restarts (after crash). Values of the counter are called *epochs*. Epochs are piggybacked in Paxos messages. Each process maintains a local map of epoch numbers for all processes. JPaxos+epochs can recover a system from failure only if a majority of processes is always up and running.

VI. Experimental Evaluation

A. Environment

In the evaluation we used six machines, on each we used one of the NUMA nodes with Intel® Xeon® Gold 6252N and six interleaved Intel® Optane™ DCPMM modules, 128GB each. The machines were equipped with 1TB Intel® DC P4510 NVMe SSDs, and were connected to the 10Gbps Ethernet switch. They were running OpenSUSE Tumbleweed with Linux 5.8, libmemobj-cpp 1.10, and openjdk 14. Each replica was run on a separate machine and the clients sending requests were run on other machines.

B. Memory Backends Used in Evaluation

We built our systems on JPaxos [6], a Java implementation of Paxos equipped with efficient state recovery support, and extended it with our new proposals, so that we can easily switch between memory backends. By a *memory backend* (or a *backend*, in short), we mean a set of routines that access data in memory (pmem, DRAM, and pmem emulated in DRAM, in our tests). We implemented mPaxos as JPaxos modified to use pmem (as in Section IV-A). Based on it we developed mPaxosSM, in which also the state machine writes (a critical state of the application) to pmem (see Section IV-B), so requires no periodic snapshots creation. The source code is available [11].

TABLE I
Systems and Memory Backends

System@backend	Snapshots	Paxos	State Machine	Crash-Recovery
JPaxos+SS	periodic	Java, data *flushed* to stable storage (e.g. disks)		catastrophic failures supported
JPaxos+epochs	periodic	Java	Java	catastrophic failures not supported
mPaxos@RAM	periodic	JNI / STL	Java	crash-stop, no recovery
mPaxos@pmem	periodic	JNI / PMDK	Java	catastrophic failures supported
mPaxos@emulp	same as mPaxos@pmem, but pmem is emulated in DRAM			
mPaxosSM@RAM	on demand	JNI / STL	JNI / STL	crash-stop, no recovery
mPaxosSM@pmem	on demand	JNI / PMDK	JNI / PMDK	catastrophic failures supported
mPaxosSM@emulp	same as mPaxosSM@pmem, but pmem is emulated in DRAM			

Table I presents all evaluated systems (with all backends). JPaxos+SS and JPaxos+epochs are the original JPaxos implementations with FullSS and EpochSS recovery algorithms, respectively. JPaxos+epochs stores all data in DRAM using Java Collections. JPaxos+SS writes synchronously to files. For a fair comparison with other systems, in our experiments JPaxos+SS writes to files stored in pmem, although we also give the results for SSDs.

The current support of Java in PMDK [12] is rudimentary, so we implemented memory backends for the pmem-enabled systems in C++ and compiled them to shared libraries, which are then invoked using *Java Native Interface (JNI)*. To estimate the overhead of JNI, which requires extra memory copying and prevents JIT optimisations, we compared JPaxos+epochs with mPaxos@RAM, which (like mPaxosSM@RAM) uses JNI and C++ standard data structures (aka STL). To estimate the overhead of pmem and PMDK, we compared mPaxos and mPaxosSM for three backends: @RAM, @pmem, and @emulp. They store data to, respectively, DRAM (using STL), real pmem (using libpmemobj-cpp, a C++ library of PMDK), and DRAM (using the same PMDK routines that are needed to write to pmem, where DRAM's address range is manually marked as persistent, thus treated by the operating system as pmem).

C. System Throughput

To measure the system throughput we run the KV-Map state machine on three replicas. The clients were sending either a 8kB `put` (50% requests), or a 9B `get` (50% requests). So the average request size was 4kB. The result for a `get` was the current value of the key, and for a `put`—the replaced one. We tested the systems with two snapshot sizes: 10MB and 100MB, which corresponds, respectively, to 1k and 12.5k unique keys in the KV-Map.

We use two metrics: *system throughput* (or *throughput*, in short) in *requests per second (RPS)* and *leader uplink use*, i.e., the use of leader outgoing network bandwidth, in *bits per second (bps)*. The leader's outgoing network link is the most busy one in the system. When the network is the bottleneck, then leader uplink use approaches 10Gbps. We calculate the link use using network interface statistics and express it in bits

TABLE II
SYSTEM THROUGHPUT WITH 4KB REQUESTS

System and backend or SS (SSD, pmem)	Snapshot size	Leader uplink [Mbps]	Throughput [RPS]
JPaxos+epochs	10MB	9 685.44	106 278.43
mPaxos@RAM		9 338.53	100 653.97
mPaxos@emulp		9 221.89	99 721.62
mPaxos@pmem		8 928.79	96 038.44
JPaxos+SS (pmem)		8 977.74	97 904.78
JPaxos+SS (SSD)		8 072.67	83 265.09
JPaxos+epochs	100MB	8 851.13	93 294.19
mPaxos@RAM		7 254.97	72 936.65
mPaxos@emulp		6 936.74	68 666.71
mPaxos@pmem		6 357.79	64 492.50
JPaxos+SS (pmem)		7 440.99	76 019.19
JPaxos+SS (SSD)		6 231.96	62 100.08
mPaxosSM@RAM	—	9 765.33	106 306.62
mPaxosSM@emulp		5 296.43	56 618.73
mPaxosSM@pmem		3 937.50	42 020.00

per second, divided by 10^6 to get Mbps. The experimental evaluation results are summarized in Table II and discussed below.

1) Throughput with Periodic Snapshots: First, we discuss the systems run with small snapshots (10MB). JPaxos+epochs uses 9.6Gbps of leader uplink, executes 106k RPS, and the network bandwidth limits the performance. mPaxos@RAM uses 9.3Gbps of leader uplink and executes 100k RPS, so is 5% slower than JPaxos+epochs. This is the cost of using JNI[5], which can be eliminated if the systems were implemented in a language with efficient native pmem support (e.g., C++). Handling each new application snapshot in mPaxos@RAM momentarily halts the system, and for the rest of the time the network limits throughput. In mPaxos@pmem the use of leader uplink is 8.9Gbps and 96k requests are processed per second. This is 10% less RPS than JPaxos+epochs and 5% less than mPaxos@RAM. mPaxos@pmem, similarly to mPaxos@RAM, halts the system while a snapshot is processed. The 5% drop in performance can be attributed to the latency of pmem. In this test, the network, not the performance of pmem, was the main cause of RPS limit.

With larger snapshots (100MB) the system throughput decreases for all systems/backends. JPaxos+epochs executes 12% less requests, while mPaxos@RAM and mPaxos@pmem, respectively, 28% and 33%. The magnitude of the throughput drop is high for the systems using JNI (mPaxos@RAM and mPaxos@pmem), and moderate for JPaxos+epochs. Hence, JNI is a major cause for the RPS drop, but the impact of limited pmem performance is also visible.

The results show that mPaxos@pmem with small snapshots is slower (by 10%) than JPaxos+epochs, but the latter system restricts the number of simultaneous failures, so it is less practical. mPaxos@pmem is also slightly slower (by 2%) than JPaxos+SS (pmem) that uses persistent memory as stable storage. However, when we deduct the cost caused by JNI (which is around 5%, if we compare mPaxos@RAM with JPaxos+epochs), we estimate that mPaxos@pmem will be slightly faster than JPaxos+SS (pmem), both of which tolerate catastrophic failures, and about 20% faster than JPaxos+SS (SSD) that uses SSDs as stable storage. On the other hand, mPaxos@pmem with larger snapshots is 31% slower than JPaxos+epochs, and is also slower (by 15%) than JPaxos+SS (pmem). As before, this cost includes JNI overhead, which for large snapshots is around 22%, when comparing mPaxos@RAM with JPaxos+epochs. So, we expect that mPaxos@pmem will be no worse than JPaxos+SS (pmem), if we get rid of JNI. On the other hand, the current implementation of mPaxos@pmem (with JNI) outperforms JPaxos+SS that writes to SSDs by 4%.

2) Throughput with On-demand Snapshots: As a baseline to evaluate mPaxosSM's throughput we took mPaxosSM@RAM. mPaxosSM@RAM uses 9.7Gbps of the leader uplink and processes 106k client requests per second. This saturates the leader uplink, and is also slightly better than JPaxos+epochs, as the latter must spend some of its resources for creating snapshots on a regular basis. So, redesigning the system to work only on the on-demand snapshots did not incur any performance problems.

With mPaxosSM@emulp the leader uses only 5.3Gbps of its outgoing link and the throughput drops to 56k RPS—47% less than mPaxosSM@RAM, even though mPaxosSM@emulp still writes to DRAM (emulated pmem) rather than real pmem. The only difference between mPaxosSM@emulp and mPaxosSM@RAM is that the former uses the libpmemobj-cpp library of PMDK and writes to memory using PMDK transactions, while the latter uses standard C++ data structures. On the other hand, mPaxosSM@pmem uses 3.9Gbps of leader uplink and processes 42k RPS, so it is 60% slower than mPaxosSM@RAM and 26% slower than mPaxosSM@emulp. Note that mPaxosSM@emulp and mPaxosSM@pmem share the same code; the only difference is that the former writes to pmem emulated in DRAM, while the latter writes to real pmem. Apparently the use of PMDK has a significantly higher impact (47%) on the system throughput than the difference in performance between DRAM and pmem (26%).

3) Profiling mPaxosSM: We used a profiler to pinpoint the cause of the slowdown. It turned out that the performance of the backends using PMDK is limited by a thread, which constantly uses 100% CPU, responsible for unbatching the decided instances and passing operations one by one to the state machine for execution. As the operations must be executed sequentially, this cannot be parallelized. We profiled the thread alone (in mPaxosSM@pmem) with the `perf` tool. In 67% of the samples, the profiler hit either PMDK's transaction upkeep, or PMDK's memory allocations. Most of the time was spent in copying data accessed from within a transaction into the undo log (`pmemobj_tx_xadd_range_direct`, 26.7% of CPU time), and in committing the transaction (`pmemobj_tx_commit`, 21.1% of CPU time). Allocating memory (`pmem::obj::make_persistent<>`) took 8.8%, and freeing it (`pmem::obj::delete_persistent<>`) took 3.9% of

[5]The only difference in failure-free operation between JPaxos+epochs and mPaxos@RAM is that the former stores the data items described in Section IV-A using standard Java data structures, while the latter uses equivalent standard data structures in C++. The performance of the data structures in C++ is no worse than in Java. However, JNI (the mechanism responsible for calling C++ code from Java) brings the performance penalty that we observe.

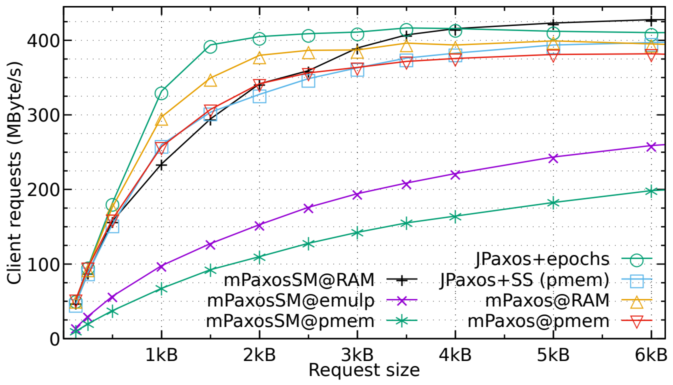

Fig. 3. Throughput for different request sizes.

CPU time.

The results show a huge cost of atomic transactions. The transactions are there to guarantee that a crash at any point of time leaves the persistent memory in a state that can be recovered to a consistent one, so to some extent this cost is unavoidable. However, we believe that writing the transactions manually instead of using the ones provided by PMDK could reduce the overhead. We leave this for future work.

4) Throughput for Different Request Sizes: We also evaluated the throughput of KV-Map on three replicas with different request sizes, in range from 128B to 10kB. To compare the throughput, we present it in MBytes/s, calculated as RPS multiplied by the request size. For each request size, through tests, we selected the numbers of parallel Paxos instances, clients, and operations in a batch that maximize the system throughput. We tested all systems for a wide range of the numbers of parallel instances (3÷10) and the sizes of operations in a batch (32k÷320k). The size of a snapshot is in order of 10MB. Fig. 3 presents the vital part of the evaluation results (up to 6kB).

For small requests (< 1kB) all systems with all backends are limited by the CPU, and apart from mPaxosSM@emulp and mPaxosSM@pmem have a similar throughput. The latter two use more CPU for PMDK transactions, so they perform worse. For moderate requests (1kB÷3.5kB) mPaxos (with any backend) outperforms JPaxos+SS (pmem), and up to 2kB also mPaxosSM@RAM. With requests up to 4kB, JPaxos+epochs is on top, and beyond it stays second best, with mPaxosSM@RAM taking the lead. With sufficiently large requests the throughput of all systems apart from mPaxosSM@emulp and mPaxosSM@pmem is limited by the available network bandwidth and (except for mPaxosSM@RAM) by the momentarily halts when periodic snapshots are created. When the requests are 2.5kB or larger, mPaxos@RAM, mPaxos@emulp and mPaxos@pmem are, respectively, 5%, 7% and 10% slower than JPaxos+epochs. With requests beyond 8kB JPaxos+SS approaches the performance of JPaxos+epochs by 1% when it uses pmem, but it is 20% slower if it uses SSDs. However, JPaxos+SS (with SSDs or pmem as SS) requires four times as many operations to be batched in each instance to reach top performance when compared to any other system.

D. Restarting after Crash

1) Restarting after Crash in mPaxos: Recovering a crashed replica in JPaxos+SS and JPaxos+epochs is thoroughly evaluated in [6], including catching up with other replicas. Restarting a replica in mPaxos@pmem differs from JPaxos+SS only in the time required by the latter to read the logs from stable storage. Therefore, below we present only a summary of the results for these backends on our hardware.

We evaluated how long it takes to recover a replica in JPaxos+epochs, JPaxos+SS and mPaxos@pmem. To this end, we run three replicas and 1k clients that were continuously sending requests (3kB on average) to KV-Map. The periodically created snapshot had 200MB. We crashed one replica (let denote it R_r), and restarted it after 20s. With JPaxos+epochs, R_r starts in 0.04s and needs another 0.04s to exchange the recovery messages. Then it executes the catch-up protocol, which takes 1.6s. In JPaxos+epochs, R_r can send any Paxos messages only once the catch-up succeeds, that is 1.7s from the restart. In JPaxos+SS, R_r can reply to the Paxos messages once it restores state from SS, which takes less time (1.14s). However, the recovered state is outdated, so JPaxos+SS starts catch-up once it learns that other replicas advanced while R_r was down. When R_r runs with mPaxos@pmem, it can send any Paxos messages immediately after restart. Starting R_r, which includes recovering the application from the snapshot (read from pmem), takes only 0.42s. Obviously, R_r must also catch up with other replicas to update its state.

2) Restarting after Crash in mPaxosSM: We evaluated how long it takes for a replica to restart and be fully functional again, and what impact had the restart on the entire system. For this, we run three and five replicas, and 1k clients which were sending non-stop either `get` (50%) or 6kB `put` (50%) requests to KV-Map. In the 30th second we crashed one of the replicas (R_r), and in the 50th second we restarted it. The size of the state machine's pmem was set to 512MB[6]. The restarted replica R_r has to catch up with the execution, and it does so by first querying a follower R_f for the commands.

Typically a client sends requests repeatedly to the same replica as long as it receives replies on time. Otherwise, the client connects with another replica. Therefore to make the restarted R_r reply to the clients, we force the clients to change replicas (selected randomly) every 1k requests.

Fig. 4 shows how the outgoing link usage changes for all replicas when one follower crashes and restarts in a system with three replicas. It takes about 0.3s to start a process, that is, start Java VM, call the code initializing our system, set up pmem memory mappings, establish network connections, etc. More precisely, it takes 0.23s from the start of R_r to establish connections to other replicas (from now on the replica can reply to Paxos messages), and after another 0.07s a catch-up query is sent. In response, R_f generates a snapshot in 0.38s. For this time the processing on R_f stops. Transmitting

[6]One reason to use a larger size than the snapshot's size used in the previous subsection, is that the results are more pronounced, and the other is that the size of pmem is the memory capacity of a state machine, while a snapshot prepared by it contains only the vital, and possibly compacted, data.

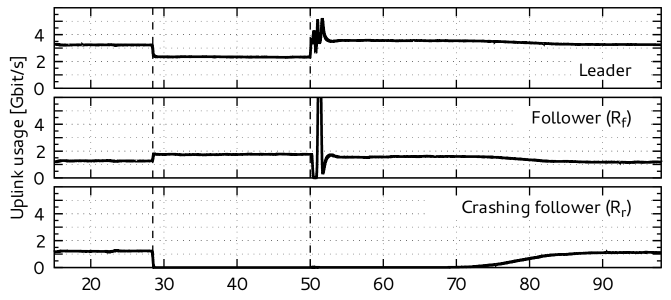

Fig. 4. Outgoing link usage while a follower crashes and later restarts in mPaxosSM@pmem with three replicas.

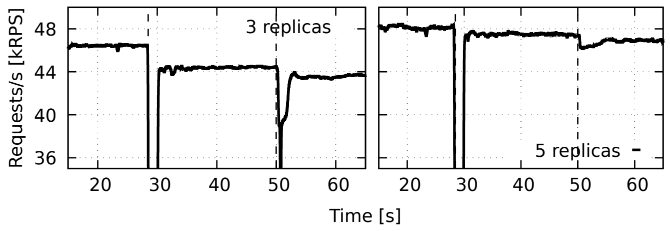

Fig. 5. Throughput upon leader crash and restart in mPaxosSM@pmem.

the snapshot (518MB) from R_f to R_r takes 0.85s. Then, R_r spends 0.55s on writing the snapshot to pmem, and another 0.23s restarting the SM on the newly written pmem. This concludes the catch-up protocol. Altogether, it takes 2.3s to start R_r and apply a snapshot received from a peer. The total number of replicas does not impact the results.

In Fig. 5 we present how the system throughput changes for a system with three and five replicas when the leader crashes and restarts. The system throughput slightly drops at crash, as with R_r down each replica must handle more clients. Once the recovering R_r executes the catch-up, the system with three replicas suffers a severe throughput drop, as R_r is catching up and R_f is preparing a snapshot, so only one replica is operational. With five replicas the impact of the catch-up protocol on the overall throughput is moderate—R_r is catching up and R_f creates a snapshot, and three remaining replicas are replying to the clients non-stop, and no point in time decisions are suspended. Once the catch-up concludes, the throughput is not immediately restored, as while R_r knows all decided operations, it still needs to execute them.

To tell when the throughput is fully restored, we check when R_r responds to clients in a timely manner. This can be seen in the use of R_r's uplink usage in Fig. 4. At R_r's crash (28s) the clients it handled move to other replicas. When R_r has updated its state from R_f's snapshot, the updated state is 2s outdated (this is the time it takes to create, transfer and apply a snapshot). So, R_r must not only execute newly decided operations, but also operations decided for these 2 secs. Once it completes catching up with execution, its outgoing link usage rises by the data sent back to the clients (70÷85s). So, it takes about 35 seconds from the start of R_r until the throughput from before the crash is restored in a system with three replicas. With five replicas restoring throughput takes 90 seconds. It takes longer because the system throughput drops less at crash in a system with five replicas, as handling the clients previously connected to R_r spreads over four rather two replicas, so R_r has less CPU time left to catch up with execution, as the CPU is more occupied by the current requests.

VII. Related work

There have been numerous articles related to consensus algorithms. Lamport's original description of Paxos in [3], was followed by attempts to explain it more clearly (e.g., [13]–[15]), or to fill in any missing details and extend the algorithm to provide a better foundation for implementation (e.g., [16], [17]) or to optimize its performance (e.g., [18], [19], [20], [21], [22], [23]). In addition, some other consensus algorithms were proposed (e.g. [24], [25]), discussed below.

However, the above papers do not explain how to develop an efficient replicated state machine system. The rise of demands for automated solutions to cluster management, failover, and sharding finally led to the adoption of consensus in practical systems (such as Chubby [7], ZooKeeper [26], Spinnaker [27], and Spanner [28]). In [7], the authors describe the design of Google's Chubby lock service, which uses Paxos. Paxos, as originally stated, is a page of pseudocode, while inside of Chubby the implementation grew up to several thousand lines of C++. In [29], the authors document the evolution of the Paxos algorithm from theory into practice while developing Chubby. A mechanism of durable logs and snapshots serialized on disk is briefly described, which is similar to ours. As we can see from these papers, making a centralized consensus system production-ready can come at the cost of adding optimizations and *recovery mechanisms*.

In [6], we discussed the existing work on failure recovery for replicated state machines, built with Paxos and other consensus protocols. The cited works include recovery from a persistent log (e.g, [17]), as in the original Paxos, or a log and snapshot written to stable storage (e.g., [29] [22]), as in JPaxos+SS. In [17], the authors evaluated the performance of Paxos with and without synchronous disk writes and show a high cost to provide crash resiliency using disks. In [6], we proposed an alternative approach for Paxos: ViewSS and EpochSS, in which a replica recovers state from peers. The latter recovery protocol is more efficient than the former and JPaxos+SS, hence we used it for comparison with our pmem-enabled Paxos frameworks in this paper.

In [27], the authors describe Spinnaker's replication protocol, which bears some similarity with Paxos. They explain how a leader and followers are recovered after a node failure. The recovery of a follower proceeds in two phases: local recovery (from a persistent log) and catch up. If the follower has lost all its data because of a disk failure, then it moves directly to the catch up phase, which is similar to the one used in our Paxos framework, but the leader is contacted. The leader responds by sending the missing committed writes, and at the end of the catch up phase, it momentarily blocks new writes to ensure that that the follower is fully caught up. In our framework, the busy leader is not disturbed at first place and the system is not blocked during catch up.

In [30], the authors proposed failure detectors aimed for the crash-recovery model, and determined under what conditions stable storage is necessary in order to solve consensus. They proposed two consensus algorithms, one requires stable storage and the other does not. They showed that stable storage is not needed to recover iff always-up replicas outnumber unstable or eventually-down replicas. They also showed that if there are no replicas which are always-up, then recovery without frequent accesses to stable storage is not possible. For a comparison, the correctness of Paxos equipped with ViewSS and EpochSS [6] is based on a more practical assumption about the number of simultaneous failures.

Oki and Liskov's Viewstamped Replication (VR) [24] (see also [31], where the core consensus protocol was described) proposed an alternative approach to consensus (see also [32], presenting a practical view on VR). The VR algorithm requires few data to be stored permanently, with sporadic writes. It also gives a clear description of how the state of lagging replicas is updated. In [31], Liskov and Cowling describe how the group reorganizes when a replica fails, and how a failed replica is able to rejoin the group in the VR (see [33] and [34] for analogous solutions aimed for Paxos). The method does not depend on stable storage, but strengths assumptions on system asynchrony and clock behaviour. The authors do not explain how a replica joining the system learns whether it begins execution or recovers after crash. ViewSS and EpochSS use, respectively, view and epoch numbers for this, which are recorded in stable storage.

Raft [25] is another consensus algorithm which shares some similarities to VR, written for managing a replicated log but designed with the goal of making the algorithm itself more understandable than Paxos. It allows proposers to be elected only if they have the most up-to-date logs. This prevents the need for transferring data from follower to leader upon election. Raft uses a catch-up method that is similar to ours, but in order to support recovery all key data of the algorithm must be written to stable storage.

Periodic state snapshots as well as the log/state transfers can severely impact the performance of some applications. In [35], the authors propose sequential checkpointing and collaborative state transfer to optimize this process. Both methods can be applied to many systems, including ours. In P-SMR [36], the recovering replica can execute new commands before the state of the replica gets fully updated, which is possible as long as the new commands are independent of the missing ones.

Since the first persistent memory products made it to market, one can observe an endless stream of new, pmem-optimized (non-replicated) key-value stores that leverage pmem's byte addressability and low latency, yielding systems that greatly outperform traditional block-based solutions (see, e.g., [37], [38], [39], [40], [41]). In [37], the authors proposed a key-value store that leverages both the byte-addressability of pmem and the lower latency of DRAM, using a technique called cross-referencing logs to keep most pmem updates off the critical path. In [41], the authors proposed a persistent hash index residing in pmem for fast searching and a B+-Tree index residing in DRAM for fast updating and supporting range scan. On top of the hybrid index, they built a key value store. We also split data in DRAM and persistent memory in the replicated state machine for efficiency. Other optimization methods described in these papers are orthogonal to the work presented in this paper, and also apply to KV-Map.

Little work has been done so far on the use of persistent memory for consensus protocols. In [42], the authors proposed an approach to providing fault-tolerance for pmem that treats memory as a replicated storage system, with a programmable network switch (which cannot fail), which implements a simple consensus protocol for keeping replicas consistent through failures. The implementation uses a generalisation of a protocol in [43] which ensures linearizable read-write access to memory, while tolerating failures. Contrary to Paxos that allows for arbitrary operations (e.g., increment), the protocol can only support reads/writes. We are not aware of other work than ours on the use of persistent memory to optimize state recovery in Paxos-based state machine replication.

VIII. Conclusions

We showed that persistent memory can boost performance of Paxos based replicated state machines, as it provides access latencies less than those of SSDs, while data persists in memory after power interruption like in SSDs and HDDs. However, the existing Paxos frameworks are unable to take full advantage of pmem because their internal architectures are predicated on the assumption that memory is volatile. We proposed two novel designs, mPaxos and mPaxosSM, which are tailor-made for pmem. The latter allows some components of the framework which degrade the system performance to be removed (e.g. periodic snapshots).

We examined the implications of pmem for recovery in Paxos. When state snapshots are small, the throughput of the mPaxos-based replicated hashmap is on par with the one using JPaxos+epochs that does not depend on stable storage (as the system recovers after failure from peers). But JPaxos+epochs requires quorum all the time, while mPaxos and mPaxosSM can tolerate any number of crashed replicas. Moreover, the mPaxosSM-based hashmap enjoys an instant recovery time, as entire critical state survives the crash in pmem.

Although, the mPaxosSM-based hashmap does not create periodic snapshots, the system throughput is comparable with the JPaxos+SS-based one that uses cutting-edge SSDs. This result shows that the overhead of using transactions and memory allocators in Intel's PMDK (a library for safe pmem programming) is high and ought to be reduced to fully gain the benefits of the new hardware.

Acknowledgment

We thank Intel Corporation and Intel Technology Poland for providing us with computing resources equipped with the Intel® Optane™ DC persistent memory technology.

References

[1] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM TOPLAS*, vol. 6, no. 2, pp. 254–280, Apr. 1984.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[3] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[4] Intel Corporation, "Intel® optane™ persistent memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[5] R. Crooke and M. Durcan, "A revolutionary breakthrough in memory technology," 3D XPoint Launch Keynote, 2015. [Online]. Available: https://www.youtube.com/watch?v=VsioS35D-HY

[6] J. Kończak, P. T. Wojciechowski, N. Santos, T. Żurkowski, and A. Schiper, "Recovery algorithms for Paxos-based state machine replication," *IEEE TDSC*, vol. 18, no. 2, pp. 623–640, March-April 2021.

[7] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. of OSDI '06*, Nov. 2006.

[8] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[9] N. Santos and A. Schiper, "Tuning Paxos for high-throughput with batching and pipelining," in *Proc. of ICDCN '12*, Jan. 2012.

[10] *Persistent Memory Development Kit (PMDK)*, Intel Corporation, 2020. [Online]. Available: https://pmem.io

[11] "mPaxos, mPaxosSM – State machine replication tools based on Paxos, with support of failure recovery form persistent memory (Java/C++)," 2021. [Online]. Available: https://github.com/JPaxos/

[12] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020.

[13] L. Lamport, "Paxos made simple," *SIGACT News*, vol. 32, no. 4, pp. 51–58, 2001.

[14] B. W. Lampson, "How to build a highly available system using consensus," in *Proc. of WDAG '96: the 10th International Workshop on Distributed Algorithms*, ser. LNCS, vol. 1151, Oct. 1996.

[15] ——, "The ABCD's of Paxos," in *Proc. of PODC '01*, Aug. 2001.

[16] R. van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys*, vol. 47, no. 3, pp. 42:1–42:36, 2015.

[17] J. Kirsch and Y. Amir, "Paxos for system builders: An overview," in *Proc. of LADIS '08: the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, Sep. 2008.

[18] L. Lamport, "Generalized consensus and Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, 2005.

[19] ——, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, 2006.

[20] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machine for wans," in *Proc. of OSDI '08*, Dec. 2008.

[21] L. J. Camargos, R. M. Schmidt, and F. Pedone, "Multicoordinated agreement protocols for higher availabilty," in *Proc. of NCA '08*, 2008.

[22] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proc. of NSDI '11*, March-April 2011.

[23] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of SOSP '13*, Nov. 2013.

[24] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A new primary copy method to support highly-available distributed systems," in *Proc. of PODC '88*, Aug. 1988.

[25] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of USENIX ATC '14*, Jun. 2014.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proc. of USENIX ATC '10*, Jun. 2010.

[27] J. Rao, E. J. Shekita, and S. Tata, "Using Paxos to build a scalable, consistent, and highly available datastore," in *Proc. of VLDB '11*, 2011.

[28] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proc. of OSDI '12*, Oct. 2012.

[29] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proc. of PODC '07*, Aug. 2007.

[30] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," in *Proc. of DISC '98*, Sep. 1998.

[31] B. Liskov and J. Cowling, "Viewstamped Replication revisited," Computer Science and Artificial Intelligence Lab., MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.

[32] D. Mazières, "Paxos made practical," Jan. 2007, Unpublished manuscript. [Online]. Available: http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf

[33] L. Lamport, D. Malkhi, and L. Zhou, "Vertical Paxos and primary-backup replication," in *Proc. of PODC '09*, Aug. 2009.

[34] L. Jehl, T. E. Lea, and H. Meling, "Replacement: Decentralized failure handling for replicated state machines," in *Proc. of SRDS '15*, 2015.

[35] A. N. Bessani, M. Santos, J. Felix, N. F. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proc. of USENIX ATC '13*, Jun. 2013.

[36] O. M. Mendizabal, F. L. Dotti, and F. Pedone, "High performance recovery for parallel state machine replication," in *Proc. of ICDCS '17*, Jun. 2017.

[37] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan, "Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs," in *Proc. of USENIX ATC '18*, 2018.

[38] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proc. of INFLOW '13: the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, Nov. 2013.

[39] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," in *Proc. of VLDB '15*, Aug-Sept 2015.

[40] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 18–26, 2015.

[41] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for DRAM-NVM memory systems," in *Proc. of USENIX ACT '17*, Jul. 2017.

[42] H. T. Dang, J. Hofmann, Y. Liu, M. Radi, D. Vucinic, R. Soulé, and F. Pedone, "Consensus for non-volatile main memory," in *Proc. of ICNP '18*, Sep. 2018.

[43] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM*, vol. 42, no. 1, pp. 124–142, 1995.