Improving the Concurrency Performance of Persistent Memory Transactions on Multicores

Qing Wang†, Youyou Lu†, Zhongjie Wu‡, Fan Yang†, Jiwu Shu†*
†*Tsinghua University,* ‡*Alibaba Group Holding Limited*
{q-wang18@mails., luyouyou@, yangf17@mails., shujw@}tsinghua.edu.cn, alanwu.wzj@alibaba-inc.com

***Abstract*—Persistent memory provides data persistence to in-memory transaction systems, enabling full ACID properties. However, high data persistence worsens the concurrency performance due to delayed execution of conflicted transactions on multicores. In this paper, we propose SP^3 (SPeculative Parallel Persistence) to improve the concurrency performance of persistent memory transactions. SP^3 keeps the dependencies between different transactions in a DAG (direct acyclic graph) by detecting conflicts in the read/write sets, and speculatively executes conflicted transactions without waiting for the completeness of data persistence. Evaluation shows that SP^3 significantly improves concurrency performance and achieves almost linear scalability in most evaluated workloads.**

I. Introduction

Emerging persistent memory (e.g., Intel's 3D XPoint), which attaches memory directly to the memory bus, enables byte-addressable access to persistent data. Since persistent memory brings data durability to in-memory systems, it is possible to ensure all the ACID (*A*tomicity, *C*onsistency, *I*solation, and *D*urability) properties of a transaction in the memory level [1]–[7].

In persistent memory transactions, both concurrency control and crash consistency are required to achieve full ACID properties. Concurrency control isolates the conflicted I/O operations between transactions, and ensures correctness for execution of multiple transactions. Crash consistency requires data versions are persisted in order, to provide consistent state change in persistent memory [1], [2], [8]. In crash consistency, strict ordering, which is required between data persistence, incurs frequent memory flush and ordering operations like `clflush` and `mfence`. When these commands are explicitly performed, the CPU is stalled. As persistent memory has relatively higher memory write latency (i.e., *persistence latency*), these flush and ordering operations lead to dramatic performance degradation [8], [9].

To reduce the transaction overhead due to high persistence latency, a number of approaches have been proposed [1]–[3], [8], which can be broadly categorized into two ways. One is to decouple the program execution from the data persistence, so as to **relax the execution ordering**. *Epoch* [8] is of such designs. For an epoch, any write after the epoch should wait until all writes before the epoch have been persisted. Epoch is supposed to be supported in the CPU cache hardware, so that the program tells the hardware about the ordering and continues execution without stalling. The other is to allow transactions to be persisted speculatively, so as to **relax the persistence ordering**. *LOC* [1] keeps versions and dependencies of transactions and speculatively persists transaction data. Transactions can be resolved when a predecessor transaction (i.e., a transaction happens before the current transaction and has conflicts with the current transaction) fails. LOC improves the efficiency of data persistence by write coalescing.

In addition to the above-mentioned techniques to reduce transaction overhead locally, *LB++* [3] also attempts to improve the concurrency performance of transactions on multicores. It is designed based on the observation that there is unnecessary ordering between inter-thread transactions. Therefore, it removes such ordering to improve persistence efficiency.

However, the worsened concurrency performance comes from not only the unnecessary ordering between inter-thread transactions, but also the delayed execution of conflicted transactions. For conflicted transactions, a transaction has to wait until the completeness (of both execution and persistence) of its predecessor transaction. For example, transaction `Txn_1` writes data `A`, and transaction `Txn_2` (`Txn_1` < `Txn_2`) reads `A`. `Txn_2` can not commit until all transaction data in `Txn_1` is committed and persisted. Due to high persistence latency in persistent memory, the commit and persist phase of `Txn_1` consumes some time. This leads to long waiting time of `Txn_2`, either stalling its CPU core or wasting its CPU core resource with aborts and retries. As such, in addition to the well-known problem of ordering and persistence overhead in persistent memory, *the concurrency execution overhead is not negligible for transactions on multicores*, which unfortunately is under-exploited.

Our goal in this paper is to improve concurrency execution and allow parallel persistence even for transactions with conflicts on multicores. To achieve this, we propose the SP^3 (SPeculative Parallel Persistence) design. The **key idea** is to speculatively execute the transaction in each core, without waiting for the completeness of the persistence of its predecessor transactions. In case of persistence failures or transaction aborts, dependencies between transactions among different cores are kept to abort the dependent transactions.

*Jiwu Shu is the corresponding author. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61772300, 61832011), and Alibaba Group through Alibaba Innovative Research (AIR) Program.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

In order to make dependencies scalable, SP^3 assigns transactional ID locally, instead of using a global ID distributor, and keeps the dependencies in a direct acyclic graph (DAG). With such design, SP^3 improves CPU core utilization of persistent memory transactions, and thereby improves the concurrency performance. Major contributions of this paper are as follows:

- We observe that transaction execution encounters dramatic performance degradation on multicores in persistent memory, and identify the cause that the high persistence latency delays the execution of conflicted transactions on multicores and wastes multicore resources.
- We propose the SP^3 (SPeculative Parallel Persistence) technique to speculatively execute transactions on different cores, by introducing the dependency DAG for dependency tracking in case of speculative failures.
- Evaluations with different workloads show that SP^3 achieves 17.2× and 6.1× higher throughput than two well-known existing persistent memory systems, PMDK and Mnemosyne, and achieves better scalability.

II. Background and Motivation

A. Persistent Memory Transactions

With persistent memory, the persistence feature can add the `D (Durability)` property to transactional memory, to make the full `ACID` possible in memory. Meanwhile, adding the durability property incurs high overhead in persistent memory transactions. It is because not only the strict ordering stalls CPU, but also the high persistence latency slows down transaction execution. There are a few categories of techniques that attempt to reduce this overhead.

Lazy Persistence (i.e., LB++) [3] is based on Epoch Persistence [8]. For a persistent memory transaction, the program issues the epoch barrier command to the CPU cache, so that the hardware ensures the ordering of data that is persisted. With the hardware epoch, the software program continues execution without explicit waiting. As shown in Figure 1(a), the execution is exempted from the ordering for persistence. It keeps the intra- and inter-thread dependencies between transactions. If there is no dependency, transactions can be executed concurrently.

Speculative Persistence further relaxes the persistence ordering [1]. It allows data from different transactions to be coalesced and persisted in parallel, by keeping their versions and dependencies. As shown in Figure 1(b), data blocks from different transactions can be reordered, coalesced and persisted. However, speculative persistence is not able to keep dependencies between different cores.

DudeTM [5] uses shadow DRAM to avoid the inefficiencies of traditional undo logging and redo logging based techniques. However, the *persist threads* in DudeTM over-restrict the persistence ordering of transactions (i.e., a transaction is considered to have been persisted only when all previous transactions in all cores have been persisted). What's more, there is only one *reproduce thread* in DudeTM to replay and clean logs, harming the system scalability [7].

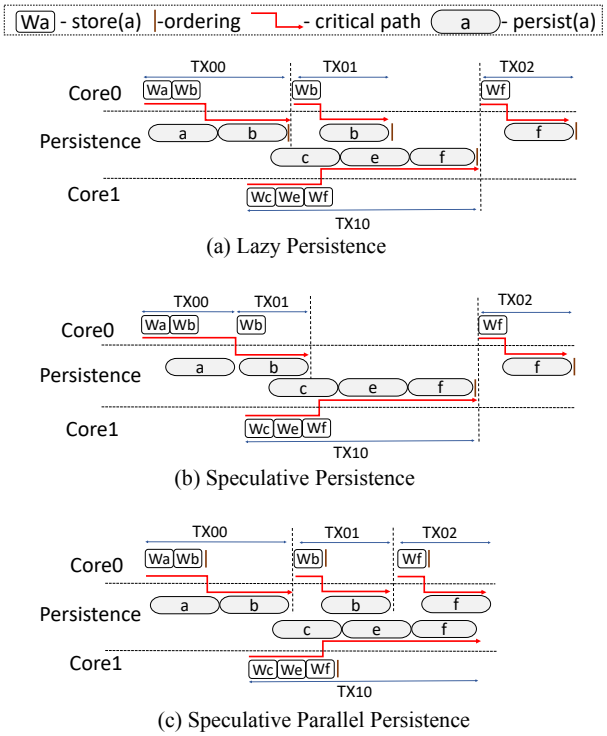

Fig. 1. Different Optimizations in Persistent Memory Transactions.

Pisces [7] optimizes read operations in persistent memory transactions via a dual-version concurrency control protocol. However, it lowers the isolation level (i.e., snapshot isolation).

B. Motivation

In persistent memory transactions, the causes of performance degradation on multicores come from two aspects: higher memory write latency and concurrency conflicts. To estimate each of the two parts in persistent memory transactions, we evaluate three systems: DRAM-TX, NVM-TX, and NVM-Ideal. DRAM-TX is an in-memory transaction system running on DRAM. For NVM-TX, we run the same transaction system, but add the memory write latency to 500ns as in [9]. NVM-Ideal removes the concurrency control from NVM-TX, without considering the correctness. In the experiment, we run red-black tree update and search operations to evaluate them.

Figure 2 compares the transaction throughput of the three systems. Intuitively, the performance degrades when the memory write latency is increased, as comparing the DRAM-TX and NVM-TX. But, worth to be noticed, the gap between NVM-TX and NVM-Ideal becomes wider when the number of cores increases. This indicates that the concurrency is a major force in degrading the transaction performance on multicores. To ensure ACID properties, all data blocks in a transaction are visible to others only after they are committed and persisted. A transaction in one core that has conflicts with another transaction in another core has to wait until the completeness of conflicted transaction.

In this paper, we aim to mitigate the concurrency problem caused by high persistence latency and improve multicore effi-

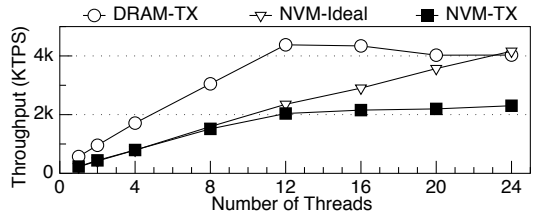

Fig. 2. Estimation of Concurrency and Persistence Overhead Breakdown.

ciency. Our proposed SP^3 is to speculatively allow transaction execution on multicores, even when they have conflicts, and achieves parallel persistence in different cores, as illustrated in Figure 1(c).

III. Design

A. Overview of SP^3

Due to persistence latency, transaction execution time is stretched, which delays the execution of conflicted transactions. Take the example shown in Figure 3(a), transaction TX_a and TX_b are executed on different CPU cores. TX_a starts to commit, after checking the versions of its read set. If there is no version change in the read set, i.e., there is no conflict, TX_a persists its data items then releases locks. At the same time, TX_b tries to access object $item_0$ which is locked by TX_a and has to wait for TX_a to finish its persistence.

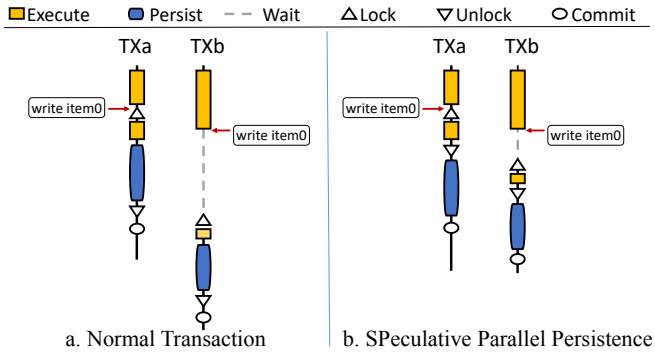

Fig. 3. Conflicts Between Transactions.

In contrast, SP^3 wants to speculatively execute transactions without waiting for conflicted transactions' persistence (Figure 3(b)). Before TX_a starts to persist log, it unlocks $item_0$. Thus TX_b can access $item_0$ without being blocked or aborted by TX_a's time-consuming persistence, which enables parallel persistence between conflicted transactions.

The challenge is to guarantee crash consistency and serializability in SP^3. In above example, TX_a is ordered before TX_b in the serial order. Therefore, TX_b can not return to upper applications until the persistence completeness of its own and predecessor transaction (TX_a). Otherwise, its durability guarantee will be violated, when the system crashes. Moreover, TX_b has to read the changes which TX_a makes to $item_0$, that is, TX_a should make modified $item_0$ visible to TX_b. To overcome the challenge, we design a commit protocol (§III-B) and a recovery protocol (§III-D) to track and replay dependencies between conflicted transactions.

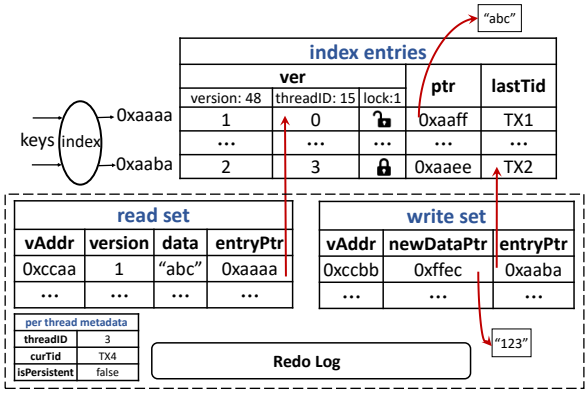

Fig. 4. Internal State In Transaction Execution

B. Commit Protocol

SP^3 uses optimistic concurrency control (OCC), which provides scalable performance in in-memory transaction systems on multicores [10]. Like other transaction systems that use OCC, SP^3 has an underlying index (shown in Figure 4), which maps keys (i.e., virtual addresses of objects) to $index\ entries$. Each $index\ entry$ consists of a pointer (i.e., ptr) to the actual data location of the object. There is a 64-bit ver word in each entry, which encodes $version$, $threadID$, and $lock$. The $version$ increases when the object is modified. The $lock$ protects the object from concurrent updates. The $threadID$ records which thread locks and writes the object at the moment. To track dependencies between transactions, we add a $lastTid$ word to each index entry, which records the ID of the transaction that last modified this object.

The execution of a transaction is divided into two phases: read phase and commit phase. During the read phase, a transaction reads objects into its read set without acquiring locks, and buffers its modifications into its write set and acquires corresponding locks. A redo log entry is also generated during read phase, but is not persisted at this time. Each entry in the write set stores the new address of the modified object in $newDataPtr$. There is an $entryPtr$ stored in each entry of write/read set, which points to the corresponding $index\ entry$, to avoid looking up the index in the commit phase.

When finishing the read phase, the thread executes commit protocol. SP^3 introduces *speculative persistence* to cross-core transaction parallelism. To achieve this, SP^3 allows data to be visible to other transactions before they are persisted. With such relaxing, transactions in the other core can read the latest data to continue execution. As such, the *execution order* between transactions on multicores is serialized, if there are dependencies, but without waiting for the persistence of the predecessor transaction. The *persistent order* can be relaxed, i.e., transactions persist their data to the log in parallel.

For the speculative failure, i.e., a predecessor transaction fails in data persistence, the transaction aborts and retries. The transaction can not return the commit success to users until its predecessor transactions complete persistence. Note that, this order is only the visibility order to the user, but not hurts the execution or persistence of transactions.

Algorithm 1: Commit Protocol

```
Input: read set RS, write set WS, log Log
for e in RS do                          /* Step 1: Validation */
    changed = e.version ≠ e.entryPtr.ver.version;
    locked = e.entryPtr.lock == 0x1
             ∧e.entryPtr.ver.threadID ≠ threadID;
    if changed ∨ locked then
        return false;
depSet = {};
for e in RS ∪ WS do                     /* Step 2: Dep Track */
    depSet = depSet ∪ {e.entryPtr.lastTid};
for e in WS do                          /* Step 3: Unlock */
    e.entryPtr.ptr = e.newDataPtr;
    e.entryPtr.lastTid = tid;
    increase_version(e.entryPtr.ver);
    unlock(e.entryPtr);
Log.append(depSet);
Log.flush()                             /* Step 4: Persist */
for depTid in depSet do                 /* Step 5: Pending */
    Wait TX_depTid is persisted
return true;
```

Algorithm 1 gives the detailed commit protocol in SP^3. In **Step 1**, the thread validates the read set. If some objects either have different versions from those stored in the read set, or are locked by other transactions, the current transaction is aborted and the thread releases locks.

On passing validation, the thread tracks dependencies (**Step 2**). For each entry in the read/write set, $lastTid$ stored in its corresponding $index\ entry$ is added to the $depSet$. The $depSet$ is the IDs of all predecessor transactions that current transaction depends on. In Figure 4, it is easy to find that $TX4$ depends on $TX1$ and $TX2$ by scanning $lastTid$ in $index\ entries$.

In **Step 3**, the thread makes the transaction updates visible to other transactions. It updates the ptr in the $index\ entries$ to point to the latest version and stores the current transaction ID in the $lastTid$ field. After that, it releases locks and other conflicted transactions can be executed forward.

Step 4 is to persist log. The thread appends the $depSet$ to the log tail, and flushes the log to persistent memory. After that, the transaction is considered to have been successfully persisted. However, the transaction can not be returned to upper applications at this point, because it is not clear whether the transactions in its $depSet$ have been persisted.

Finally, in **Step 5**, the thread waits for all transactions in its $depSet$ finishing persistence (i.e., **Step 4**). To detect the transaction persistence state of transactions in the $depSet$, we record $\langle curTid, isPersistent\rangle$ pair for each thread. $curTid$ is the ID of transaction currently running in this thread. The persistence of transactions inside each thread is in strict order. If the transaction ID in the $depSet$ is smaller, it is persistent. If the transaction ID equals $curTid$, the $isPersistent$ indicates the state. The $isPersistent$ is set to $true$ after **Step 5** in the commit phase. The transaction is considered to have been committed successfully at this point, and ACID properties are provided to the upper applications.

C. Logical Transaction ID

For scalability, SP^3 uses per-thread log and global logical ID without a centralized global ID distributor. The global logical ID is designed based on the dependency tracking in SP^3. Each thread in SP^3 holds a local logic clock, which is encoded by the pair $\langle threadID, logicID\rangle$. When a new thread is registered into SP^3 system, it is allocated with a unique $threadID$. In each thread, the $logicID$ is used for the local transaction ID and is incremented right before a new transaction starts. As such, $\langle threadID, logicID\rangle$ is used as a global logical transaction ID to identify the transaction itself.

The ordering between different transactions can be represented by a dependency DAG (direct acyclic graph). In the dependency DAG, each node represents a transaction and each edge directed from $node_i$ to $node_j$ represents TX_i is ordered before TX_j. TX_i is ordered before TX_j (i.e., $TX_i < TX_j$) *if and only if* one of the following three conditions is met:

(1). $threadID_i == threadID_j \wedge logicID_i < logicID_j$

(2). $threadID_i \neq threadID_j \wedge \langle threadID_i, logicID_i\rangle \in depSet_j$

(3). $\exists TX_k, TX_i < TX_k \wedge TX_k < TX_j$

D. Recovery

When the system crashes, SP^3 needs to first analyze the dependencies across different cores, and then replay the logs in a logical order globally. For the dependency, the recovery protocol exploits the dependency DAG described in §III-C. With this dependency, the recovery process can replay logs accordingly.

First, SP^3 constructs the DAG dependency graph. The recovery thread gathers all valid logs generated by different threads, then parses them. It inserts a new node into dependency DAG when reading a *complete* log entry, and adds the edges representing corresponding dependencies. Then, the recovery thread deletes *uncommitted* transactions from the DAG. A transaction is uncommitted if a dependent transaction is not in the DAG or is uncommitted. When the DAG is constructed, the recovery thread performs a topological sort for all DAG nodes, as all nodes in the DAG are in a partial order. After the sorting, the recovery thread is able to recover data and index by executing log entries in the result order.

The recovery can be performed in parallel using multi-threads. For example, the results of toposort can be partitioned, and tasks can be dispatched to different threads, to make all threads running in parallel. SP^3 also performs checkpoint periodically, to truncate logs and reduce recovery overhead.

IV. Evaluation

A. Experimental Setup

We evaluate SP^3 on a SuperMicro server. The server is installed with CentOS 7.6 at kernel 3.10.0. It is equipped Intel Xeon 2.2GHz CPU (2 sockets $\times$ 12 cores) and 128GB DDR4 memory. For explicit persistence operations after `clflush` commands, we add an extra write latency to emulate the

persistent latency. This latency is emulated using the CPU timestamp counter (TSC) as in previous studies [9], [11].

Evaluated Systems. We compare SP^3 to two well-known existing persistent memory systems, Mnemosyne [9] and PMDK [12]. In addition to SP^3, we also evaluate NVM-TX and NVM-Ideal for comparison. NVM-TX is the baseline system without optimizations in SP^3. NVM-Ideal is a configuration that removes the concurrency control from NVM-TX, without correctness guarantee, so as to estimate the ideal performance for concurrency optimization.

Workloads. In the evaluation, we use different workloads as shown in Table I, similar to the evaluation in previous studies [1], [3], [9], [13]. For these workloads, the entry size is set to 128-byte. For ordering and persistence operations, we issue `clflush` and `mfence` commands. The default extra latency is set to 500 ns. To evaluate the contention, the default zipfian parameter is set to 0.99 for workload skews, which is used in existing benchmarks like the YCSB workload [14]. The ratio of insert operations is 50% by default.

Workload	Description
SPS	Random swaps of array entries
Queue	Insert/delete entries in a queue
HashTable	Insert/search entries in a hash table
RBTree	Insert/search nodes in a red-black tree
B+ Tree	Insert/search nodes in a B+ tree

TABLE I
WORKLOADS.

B. Comparison with Existing Systems

We first compare our proposed SP^3 system with Mnemosyne [9] and PMDK [12], respectively, with a single core or multiple cores (16 cores). Figure 5 shows the transaction throughput of each system for different workloads. From the figure, we have two observations:

(1) All the three systems show comparable performance in the single-core evaluation, while they show significant differences in the 16-core evaluation. In the single-core evaluation, the throughput in SP^3 is 34.27% to 301.65% of that in PMDK, and 87.52% to 320.26% of that in Mnemosyne. The average performance of SP^3 outperforms PMDK and Mnemosyne by 127.40% to 195.94%, respectively. In the 16-core evaluation, the throughput in SP^3 is 78.83% to 5149.64% of that in PMDK, and 187.63% to 1647.53% of that in Mnemosyne. The average performance of SP^3 outperforms PMDK and Mnemosyne by 1723.30% and 613.99%, respectively.

(2) SP^3 gains higher performance improvement than the other two systems on multicores. The throughput of SP^3 with 16 cores is $0.61\times$ to $13.23\times$ of that with a single core for different workloads, with an average improvement of $10.11\times$. PMDK has $0.35\times$ to $3.61\times$ throughput improvement from a single core to 16 cores, with an average improvement of $1.13\times$. Mnemosyne has $0.70\times$ to $15.68\times$ throughput improvement from a single core to 16 cores, with an average improvement of $5.59\times$.

C. Concurrency Performance

We then evaluate the scalability of SP^3 by varying the number of cores from 1 to 24. To better understand SP^3's scalability, we also give the scalability of Mnemosyne, NVM-TX, and NVM-Ideal. In NVM-Ideal, because concurrency control is removed, there is a correctness issue in memory access when inserts occur. Therefore, only update operations are performed in NVM-Ideal.

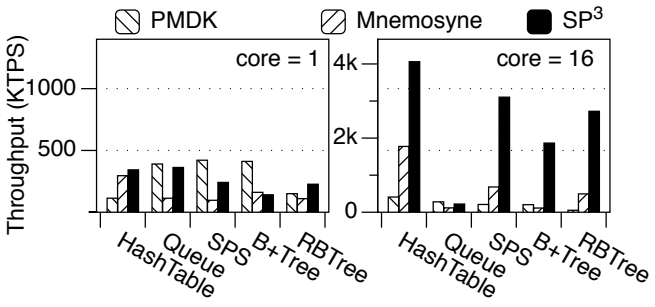

Fig. 5. Comparison with Existing Systems in Single/Multi Core.

Transaction Throughput. Figure 6 shows the concurrency performance of Mnemosyne, NVM-TX, SP^3 , and NVM-Ideal. For all evaluated workloads, SP^3 shows relatively better scalability than the other two persistent memory transaction systems Mnemosyne and NVM-TX. SP^3 achieves almost linearly growing performance when the number of cores increases for the evaluated workloads, except for the SPS and Queue workloads. In comparison, Mnemosyne does not scale well when the number of cores increases. NVM-TX, which uses optimistic concurrency control, still has performance degradation when the number of cores is large. Comparing NVM-TX and SP^3, SP^3 allows transactions in different cores to continue persistence, even when they have conflicts. This design reduces the wasted CPU stalling in different CPU cores, and thereby improves the multicore efficiency. As such, the SP^3 design is effective in improving concurrency performance in persistent memory transactions.

In the evaluated workloads, the SPS and Queue workloads show performance degradation in SP^3. This is because of high contention in the two workloads. The array and queue data structures are simple, and all elements are linearly organized, so the skewed workload setting (with a zipfian parameter of 0.99) leads to contention to a few data elements.

Abort Ratio. Figure 7 shows the corresponding abort ratio respectively for NVM-TX and SP^3. From the figure, we can see that SP^3 has a significantly lower abort ratio than NVM-TX. This also explains the reason why SP^3 gains performance improvement on multicores shown in Figure 6. In the evaluated workloads, the SPS and Queue have high abort ratio, which is discussed above. In the other workloads, SP^3 achieves a much lower abort ratio than NVM-TX. This is because transactions are speculatively executed by recording the dependencies across cores, and this speculative technique reduces transaction aborts and retries.

D. Recovery Performance

We also evaluate the recovery performance of SP^3. Two parameters have high impact on recovery performance. One is the number of threads that execute concurrent transactions.

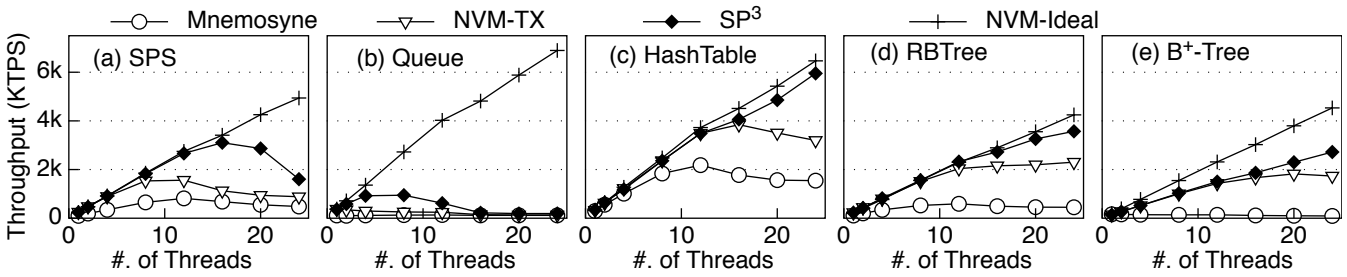

Fig. 6. Throughput.

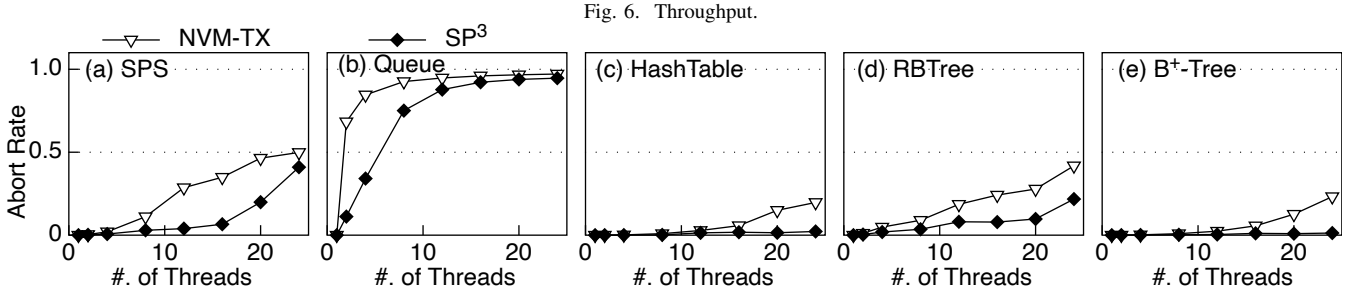

Fig. 7. Abort Ratio.

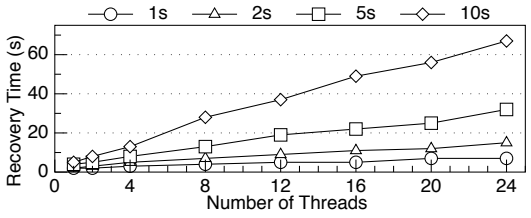

Fig. 8. Recovery Performance.

During recovery, DAG construction requires dependencies over different threads' logs, which increases recovery latency. The other is the checkpoint interval. When the interval is large, the persistent frequency of indexing metadata is reduced, but the amount of data that needs to be scanned to construct DAG is increased. Therefore, in this experiment, we vary the values of both them. The number of threads is varied from 1 to 24. The checkpoint interval is set to 1s, 2s, 5s, and 10s. We first run the SP^3 transaction system until they generate 10GB logs, and then perform the recovery test. Note that, the scanned data amount is less than 10GB during recovery, because the only transactions from the last checkpoint need to be processed.

Figure 8 shows the recovery time for different number of threads and different checkpoint interval. For each checkpoint interval, the recovery time increases when the number of threads is increased. This is because dependencies have to be scanned from different logs to construct the DAG dependency graph, as stated before. For different checkpoint intervals, the recovery time increases when the checkpoint time is increased. This is because that more logs have been persisted but not checkpointed when the checkpoint interval is large. These logs have to be scanned to check the dependency. In all, the recovery can complete within tens of seconds. For a typical 5s checkpoint interval, the recovery takes around 30 seconds even for 24 cores, which is acceptable.

V. Conclusion

In persistent memory transactions, high persistence latency leads to a high property of conflicts to transactions in multicores. To address this issue, we propose SP^3 (SPeculative Parallel Persistence), a persistent memory transaction system to improve concurrency performance on multicores. SP^3 speculatively executes transactions in different cores, to improve the core efficiency. It also keeps the logical dependencies of transactions across cores, without using a global ID distributor. Evaluation shows that SP^3 achieves almost linear performance scalability in persistent memory transactions.

References

[1] Y. Lu et al, "Loose-ordering consistency for persistent memory," in *ICCD*, 2014.
[2] S. Pelley et al, "Memory persistency," in *ISCA*, 2014.
[3] A. Joshi et al, "Efficient persist barriers for multicores," in *MICRO*, 2015.
[4] A. Kolli et al, "High-performance transactions for persistent memories," in *ASPLOS*, 2016.
[5] M. Liu et al, "DudeTM: Building durable transactions with decoupling for persistent memory," in *ASPLOS*, 2017.
[6] A. Joshi et al, "DHTM: Durable hardware transactional memory," in *ISCA*, 2018.
[7] J. Gu et al, "Pisces: A scalable and efficient persistent transactional memory," in *USENIX ATC*, 2019.
[8] J. Condit et al, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.
[9] H. Volos et al, "Mnemosyne: Lightweight persistent memory," in *ASPLOS*, 2011.
[10] S. Tu et al, "Speedy transactions in multicore in-memory databases," in *SOSP*, 2013.
[11] Y. Lu et al, "Blurred persistence in transactional persistent memory," in *MSST*, 2015.
[12] "Intel corporation. persistent memory programming," "http://pmem.io/", 2019.
[13] J. Zhao et al, "Kiln: closing the performance gap between systems with and without persistence support," in *MICRO*, 2013.
[14] B. F. Cooper et al, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.