Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx

Amirsaman Memaripour[†] Anirudh Badam[‡] Amar Phanishayee[‡]
Yanqi Zhou[⋆] Ramnatthan Alagappan[+] Karin Strauss[‡] Steven Swanson[†]

UC San Diego[†] *Microsoft*[‡] *Princeton University*[⋆] *Univ. of Wisconsin-Madison*[+]

Abstract

Data structures for non-volatile memories have to be designed such that they can be atomically modified using transactions. Existing atomicity methods require data to be copied in the critical path which significantly increases the latency of transactions. These overheads are further amplified for transactions on byte-addressable persistent memories where often the byte ranges modified for data structure updates are significantly smaller compared to the granularity at which data can be efficiently copied and logged. We propose Kamino-Tx that provides a new way to perform transactional updates on non-volatile byte-addressable memories (NVM) without requiring any copying of data in the critical path. Kamino-Tx maintains an additional copy of data off the critical path to achieve atomicity. But in doing so Kamino-Tx has to overcome two important challenges of safety and minimizing NVM storage overhead. We propose a more dynamic approach to maintaining the additional copy of data to reduce storage overheads. To further mitigate the storage overhead of using Kamino-Tx in a replicated setting, we develop Kamino-Tx-Chain, a variant of Chain Replication where replicas perform in-place updates and do not maintain data copies locally; replicas in Kamino-Tx-Chain leverage other replicas as copies to roll back or forward for atomicity. Our results show that using Kamino-Tx increases throughput by up to 9.5x for unreplicated systems and up to 2.2x for replicated settings.

1. Introduction

Designers of persistent data structures, storage systems, and databases have consistently used fast volatile memories, like processor caches and main memory, to mask the access latencies of durable media like SSDs and hard disks. The need to ensure that all changes are atomic and durable, even in the face of failures, has dominated their design. Traditionally, these systems have used either write-ahead logging [15, 23] or copy-on-write [18] to achieve atomicity and durability.

EuroSys '17, April 23-26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064215

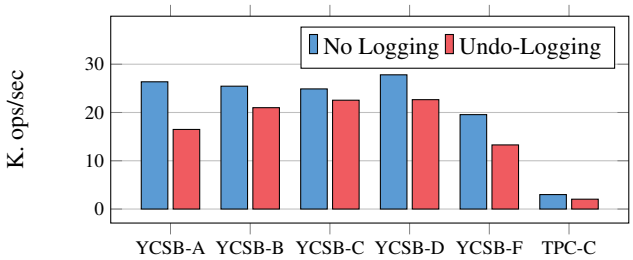

Figure 1: Avg. throughput for running YCSB workloads (A-F) and TPCC benchmark suite against MySQL. YCSB workloads B-D are over 90% reads and hence have lower logging overheads overall.

With Non-Volatile Memory (NVM) the dream of persistent storage, addressed at a byte granularity directly by the CPU, and accessed roughly at the latency of main memory, is on the horizon. It can be used to obtain durability at speeds that are two to three orders of magnitude faster than SSDs. Battery-Backed DRAM (BBRAM), Intel/Micron's 3D-Xpoint (3DXP) and HP's memristors are a few examples of such a technology. Since these memory technologies are durable, data must be modified atomically when moving from one consistent state to another. However, existing atomicity mechanisms have a high overhead as they require old data to first be copied in the critical path before it can be modified. Such copying increases the latency of transactions and decreases throughput.

When atomicity is obtained using undo logging, old data is copied to a different location on NVM called the undo-log before the transaction can edit data in place. If the transaction aborts then the transaction coordinator restores the old state using the copy in the undo-log. Likewise, in copy-on-write (CoW) based schemes, all modifications are made to a copy of data in NVM. If the transaction has to abort then simply deleting the copy is enough.

In either case, however, a copy of the data has to be created in the critical path which increases memory bandwidth usage and more importantly increases the latency of transactions due to instruction overhead in allocating space for the copies, indexing the copies, performing the actual copying of the data, and ultimately for deallocating them. An example is the logging scheme used by MySQL to offer atomicity for SQL transactions [26] on an NVMM (non-volatile main memory) based disk. Figure 1 shows how logging can impact MySQL throughput for a client with four threads running YCSB workloads, and for transaction throughput using TPCC benchmark suite, showing typical overheads of 50–250%.

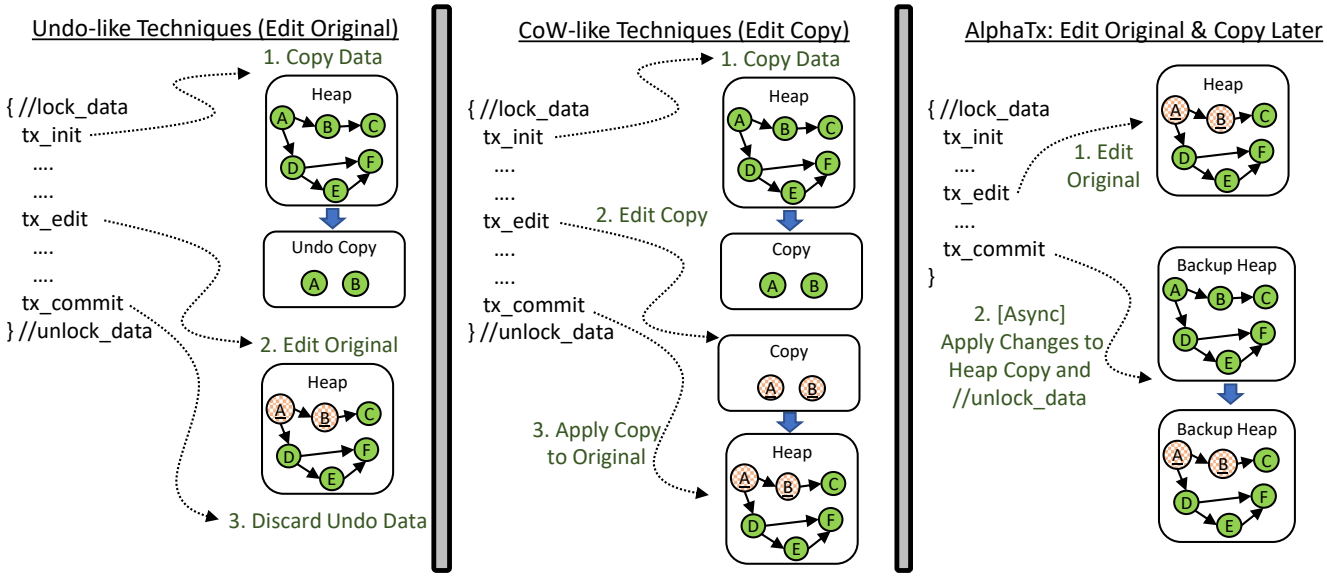

Figure 2: Comparison of the three atomicity mechanisms when a transaction modifies three objects A and B that are part of a larger persistent heap: First approach creates a copy of the objects and edits the original heap in place. In case of an abort, the copies are used to restore the heap. The copies are deleted and locks released after committing. In the second approach, the copies are edited and are later applied to the original heap after a successful commit; locks are also released at this point. In case of an abort, the copies are simply deleted. In Kamino-Tx, two copies of the entire heap are maintained. Transactions edit original data without any copying. In case of an abort, data is restored from the copy. After a commit, edited data is first applied to the copy and then the locks are released.

These overheads are especially magnified if the granularity at which data is logged is larger than the actual byte-ranges that the transaction modifies. For example, in a document store such as MongoDB [24], an entire document is typically logged though each operation might only change a few byte-ranges within the document. Likewise, in Intel's NVML, an entire `C` structure is typically logged in transactional updates to data structures even though only a few fields are typically modified as part of the transaction. While memory bandwidth can be reduced by fine-granularity logging, the overhead of CPU instructions for allocating, indexing and deallocating the copies remains high.

Motivated by these problems, we propose Kamino-Tx to address the following challenge for NVM-based systems: *In the common case, can we achieve atomicity in transactions without requiring any copying of data in the critical path?* We start with the observation that we can eliminate copying in the critical path by maintaining an additional copy (*backup*) of all the data and using that copy to recover from failures and aborts. All transactions are processed in-place on the main copy – no copying is performed in the critical path of the transaction. All successful transactions are also asynchronously applied in-place to the backup once they are committed to the main copy. If the transaction on the main copy has to abort then the state in the backup is used to undo the changes back to a consistent state. Thus, Kamino-Tx also removes the overhead of log and buffer management which is necessary for traditional logging-based techniques.

Kamino-Tx addresses two important challenges for in-place updates to work efficiently: *Safety* and *high storage requirement*. First, Kamino-Tx should be safe for all workloads and not just the common case. In the case of dependent transactions, where the write-set of a transaction intersects with the read- or write-set of a subsequent transaction, the latter transaction has to wait for the main and backup copies to be replicas of each other with respect to the write-set. Second, Kamino-Tx minimizes storage requirement by intelligently tracking the most frequently written objects and maintains copies only for those objects to strike a balance between latency and storage requirement.

Furthermore, we address the challenge of minimizing the storage footprint when using Kamino-Tx in a replicated setup. If used naively, where each replica maintains a backup copy, Kamino-Tx can double the cluster-wide NVM storage requirement. Driven by the observation that replicas can be used as copies for rolling back or forward for atomicity, we present Kamino-Tx-Chain, a new way to perform chain replication where replicas avoid maintaining individual backup copies and perform in-place updates.

This paper makes the following contributions:

- A new mechanism to obtain atomicity for transactions on NVMM is proposed by using an additional copy of data that is updated off the critical path to reduce transactional latency.
- A mechanism to maintain copies of only the most frequently modified objects is proposed to reduce the cost of such a copy.
- A new replication mechanism is proposed that leverages replicas as copies for both availability and to provide transactional atomicity.

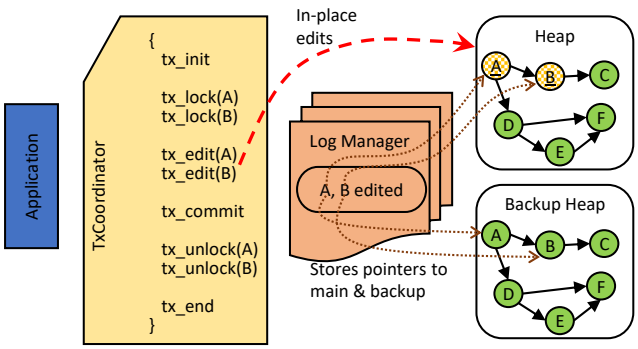

Figure 3: Architecture of the Kamino-Tx transactional system for managing a persistent object heap in non-volatile memory. The Transaction Coordinator uses the Log Manager to efficiently track the changes being made by the application. These changes are tracked only in terms of addresses of the persistent objects being modified. Transaction Coordinator uses this information to abort changes made to the main version by copying data from the backup version when necessary.

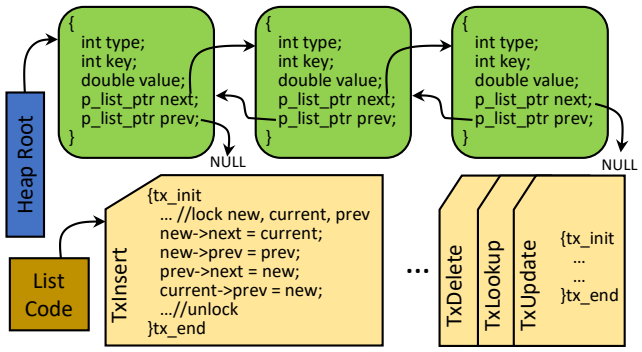

Figure 4: Structure of the heap and the format of transactions in Kamino-Tx. Kamino-Tx's heap is a collection of persistent objects in non-volatile memory. They have native types and pointers to other objects. Transactions in Kamino-Tx atomically modify several objects at a time.

The rest of this paper is organized as follows: We first discuss related work to improve atomic updates on NVMs in Section 2. We then describe the architecture of Kamino-Tx in Section 3. Section 4 presents the optimization to reduce the storage requirement of the backup and Section 5 extends these optimizations to chain replication. The implementation details of Kamino-Tx are presented in Section 6. Finally, we present the evaluation results from experimenting with Kamino-Tx in Section 7 before concluding in Section 8.

2. Related Work

Persistent heap management systems [6, 8, 20, 31, 32] allow safe, transactional and user-level access to non-volatile memories. Such systems enable applications to allocate and free objects from persistent memory, to have persistent pointers between such objects to help build rich data structures, and to perform transactions spanning multiple persistent objects in an atomic and consistent fashion.

Some such systems use a undo-based approach [6, 8, 20] while others have proposed using a copy-on-write approach [31, 32]. Both these categories of systems still require creating copies of modified objects/data in the critical path of the transaction before it is committed to the application. In this paper, we present a new way to implement transactions for persistent heap managers on non-volatile memories that can provide atomicity without having to perform any copying in the critical path of the transactions. Figure 2 illustrates how the commit operation is delayed because of the copying of the data in the critical path for undo and CoW approaches. In comparison, Kamino-Tx commits faster because the copying is performed off the critical path.

Several mechanisms have been proposed to optimize the implementation of the atomicity mechanisms for transactions on NVMs [4, 33]. For instance, Arulraj et al [4] compare three different optimized implementations of DBMS transactions for NVM that they refer to as NVM-InP, NVM-CoW and NVM-Log. The three approaches differ in the order in which copies are created, whether the original or the copy is modified and finally on how the database is updated with the new data. In all these three implementations, however, data is first copied in the critical path before it is modified which we aim to eliminate in Kamino-Tx.

NVM-Aware Storage. A wide range of applications use file systems and databases to access and manage persistent storage. Since block-based systems cannot fully exploit the byte-addressable potential of NVMs, NVM-aware file systems [1, 9, 12, 22, 34], databases [4, 14, 19] and replicated storage systems [11, 21, 35] are being proposed. There are also techniques which simply add failure consistency to the *mmap()* interface [27]. However, like traditional file systems and database systems, they provide atomicity by creating copies of modified data in the critical path. With Kamino-Tx, we propose improving the performance for these systems by providing atomicity more efficiently.

Hardware Support. Persistent processor caches [17, 36] and system-level persistence [25] would guarantee pending writes to NVM can be recovered after failures. It also eliminates the overhead of flushing caches for persistence [5]. However, atomicity is still necessary to protect such systems against bugs, deadlocks or live-locks in the OS which can leave the data in an irrecoverable state. Kamino-Tx does not require but can reap the same benefits from such novel hardware support.

3. Kamino-Tx

Kamino-Tx introduces a new atomicity mechanism that allows an application to modify multiple persistent objects inside a transaction without having to create copies of them in the critical path. In this section, we begin by presenting Kamino-Tx-Simple, a simpler version of the new atomicity mechanism that highlights the technique, and the safety challenges it has to overcome. We use a running example application of a transactional heap manager for non-volatile

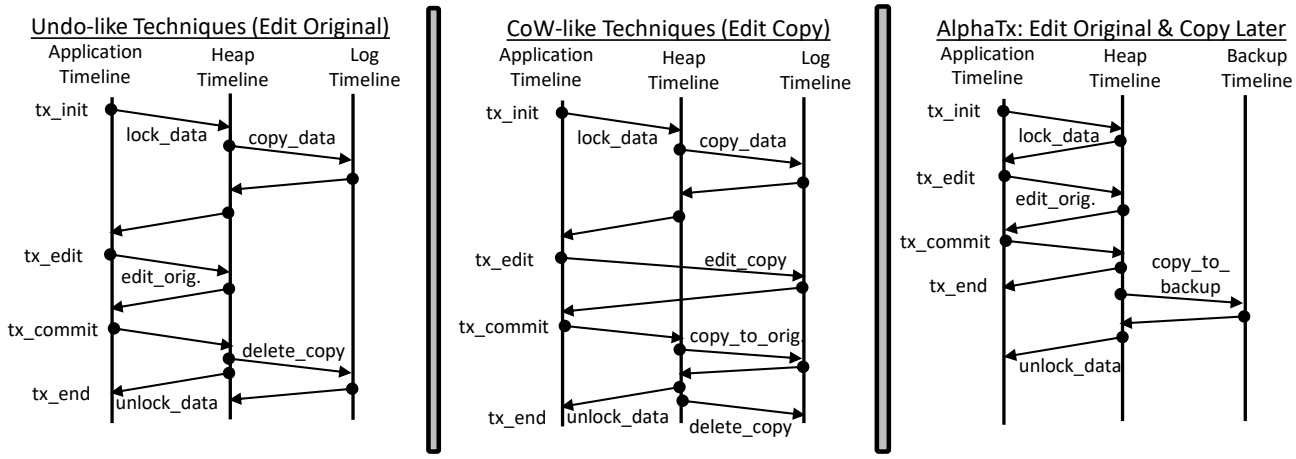

Figure 5: Existing methods spend significant amount of time in the critical path for creating copies of data. Kamino-Tx on the contrary moves this latency off the critical path.

memory. In the next section we present optimizations to reduce some of the storage overheads of this simple scheme.

Figure 3 shows the four high-level components in Kamino-Tx: (1) A persistent heap manager that allows applications to allocate and deallocate persistent objects, and allows such objects to store data and pointers to other objects, (2) a Transaction Coordinator interface that allows applications to atomically modify multiple persistent objects at a time within a transaction such that the heap always remains consistent, (3) a Log Manager that helps the Transaction Coordinator track the persistent objects modified by a transaction, and (4) a backup version of the entire heap manager that the Transaction Coordinator and the Log Manager use for enforcing atomicity on the main version of the heap.

Data in Kamino-Tx's heap is organized as a collection of persistent objects in non-volatile memory. These objects store native types such as integers, floats, doubles, strings and also persistent pointers to other persistent objects. Transactions in Kamino-Tx involve reading and editing a few such persistent objects atomically. Object edits can take the form of changing a field such as a native type or a persistent pointer as well as a whole object edit without reading it apriori. To enforce isolation, Kamino-Tx transactions declare write intent at an object granularity when the Transaction Coordinator grabs an object level lock. This model of managing persistent memory is more or less identical to proposed systems in the past [6, 8, 20, 31, 32]. Figure 4 shows how an operation in a persistent doubly linked list is implemented using Kamino-Tx.

Kamino-Tx-Simple: The main idea of Kamino-Tx is to provide transactional updates without the overhead of creating copies of modified objects or fields in the critical path – transactions are allowed to directly modify persistent objects without having to make any copies first. However, a consistent version of the heap is still needed to recover from failures or transaction aborts. Kamino-Tx-Simple always maintains a backup version of the entire heap at a separate location within the same machine, but in most cases avoids copying of data in the critical path of transaction execution. Copying of data to and from the backup is performed asynchronously.

Transactions in Kamino-Tx-Simple work as follows: First, the transaction is applied to the main version of the data in place without creating any copies of the objects involved. Second, if the transaction has aborted then the unmodified objects in the backup version are used to roll the main version back to a consistent state before the transaction started. If the transaction commits then the modified persistent objects are copied to the backup version of the data in-place.

Figure 5 contrasts existing methods of providing atomicity for transactions in non-volatile memory against Kamino-Tx. Kamino-Tx has the potential to complete transactions faster because it copies data off the critical path rather than during the execution of the transaction.

Kamino-Tx has to ensure the following important properties for safety, even in the presence of failures and reboots:
Safety 1: When a transaction commits successfully, its changes need to be propagated to the backup before a dependent transaction executes.
Safety 2: When a transaction aborts, its changes need to be rolled-back from the backup before a dependent transaction executes.
We define a *dependent* transaction as one whose read- or write-set intersects with any prior transaction's write-set. We call this intersection the set of *pending objects*. Dependent transactions have to wait for the main and backup copies to be consistent with each other (only for pending objects), and are the only transactions in Kamino-Tx for which data is copied in the critical path if not already copied asynchronously before they start.

To ensure the safety properties listed above, Kamino-Tx's Transaction Coordinator first acquires read-write locks for the objects in the working set of the transaction it is executing. The Transaction Coordinator releases these locks only after the main and backup are consistent with respect

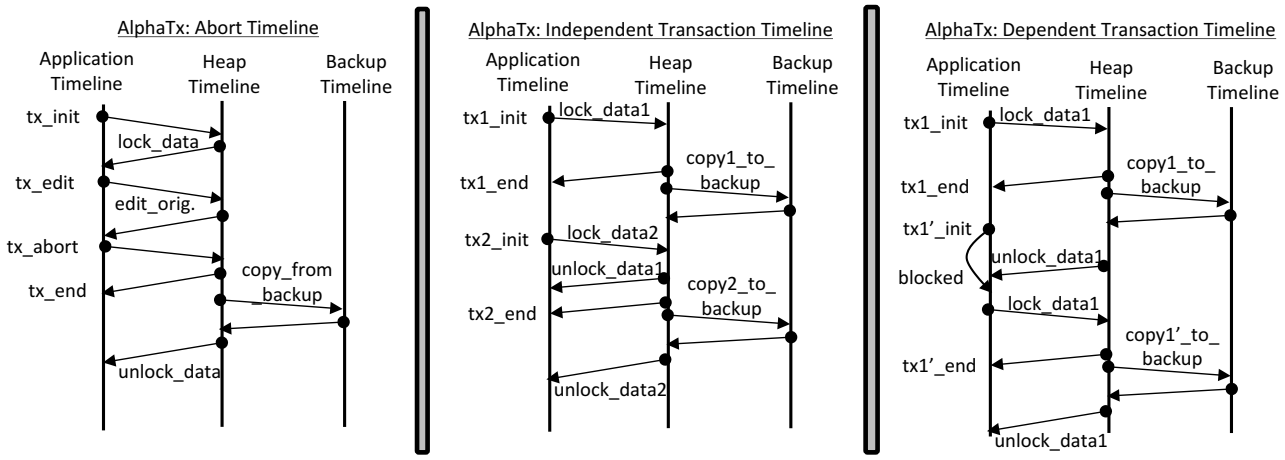

Figure 6: Kamino-Tx aborts by fetching the original data back from backup version and updating the main version back to a consistent state. In Kamino-Tx, a transaction that does read or write objects locked by other transactions proceeds immediately without having to wait for the backup version to catch-up. However, transactions that read or write objects locked by other transactions need to wait until the backup version catches up and the locks are released.

to the pending objects. Holding onto locks, however, delay dependent transactions as shown in Figure 6. This implies that the worst case performance of Kamino-Tx would be that of traditional transactional systems. However, we find in our evaluation with several real world workloads that only a small percentage of transactions are dependent and therefore the overall throughput benefits are significant.

Kamino-Tx ensures existence of a consistent copy of each persistent object before allowing a program to modify it. If the backup copy of an object is outdated, Kamino-Tx would not allow any transaction to acquire its lock until the backup is up-to-date. Moreover, Kamino-Tx replicates updates of an object to its backup copy only at transaction commit and once its main copy is durably stored on NVMM. Thus, Kamino-Tx provides the same semantics as undo-logging by maintaining a consistent durable copy of objects.

For the atomicity mechanism to work efficiently, Kamino-Tx needs to know the differences between the main version of the data and the backup version at all points of time. This information is needed for rolling back the changes to the main version on aborts and also for rolling forward the backup version for committed transactions. We design a simple mechanism called Log Manager to maintain this difference in an efficient manner by remembering only the addresses of the objects that are modified in each transaction.

Log Manager: The Kamino-Tx Log Manager maintains a persistent and temporally ordered log of transaction write intents and transaction outcomes of commit or abort. Before the user program starts modifying a persistent memory region, the Log Manager creates a reliable record of the write intent, comprising of the persistent object's address. This record should be accessible by the Transaction Coordinator regardless of a node reboot. The Log Manager leverages Intel's NVML interfaces to detect write intents which are recorded through space-efficient NVM-optimized logging. Furthermore, the Log Manager also records whether a commit or abort was called by the application.

The information stored by the Log Manager is used by the Transaction Coordinatorfor three reasons: (1) Once the transaction commits on the main version of data, the Transaction Coordinator uses the write intents to copy the corresponding persistent objects from the main version to the backup version, (2) If the transaction is aborted for any reason then the Transaction Coordinator uses the write intents to copy the corresponding persistent objects from the backup version to the main version, and (3) After a crash, during the recovery process, the Transaction Coordinator uses the Log Manager's information by performing the appropriate action ((1) or (2)) depending on whether each transaction was committed, aborted, or left incomplete due the crash (incomplete transactions are treated the same as aborted transactions). Figure 6 shows how aborts roll back to a previous consistent state by fetching objects from the backup of the heap.

As alluded to earlier, for each write intent, the Transaction Coordinator holds a lock on the corresponding data item and does not release the lock until: (a) the transaction is either committed or aborted, and (b) both the main and backup versions of data are identical. Such locks are maintained in volatile memory to speed up the system: write intents are enough to recover the lock information needed as we show in Section 6.2.

Once the Transaction Coordinator is done with committing the transaction on the main version and copying the modified objects to the backup version, Log Manager is free to remove all corresponding write intents. Although Kamino-Tx removes copying from the critical path for non-dependent transactions, it still needs to make Log Manager records durable before allowing transactions to proceed with writes. However, our experiments show that our implementation of Log Manager, which is based on fine-grained logging of fixed-size write intents with minimum number of

cache flushes, provides orders of magnitude better performance than copying data in the critical path.

Despite getting rid of data copies from the critical path of transactions in the common case, Kamino-Tx-Simple suffers from one disadvantage: having a full copy of the data can be expensive for some applications. In the next section, we present an optimization to reduce this storage overhead while retaining most of the benefits of Kamino-Tx-Simple.

4. Reducing Cost of Kamino-Tx

Since Kamino-Tx maintains an additional copy of the entire data in the form of a backup, the cost of maintaining a transactional heap would increase by the amount needed for the additional NVM. In this section, we propose techniques to reduce the storage overhead of Kamino-Tx-Simple.

Kamino-Tx-Simple has a storage requirement of $2 \times$ *dataSize* where *dataSize* is the size of a single application heap. We present an optimization to Kamino-Tx such that the storage requirement reduces to $(1+\alpha) \times dataSize$ where α is a tunable parameter (positive real number less than 1).

For single server systems, we make the observation that most application working set sizes are skewed and contain a small percentage of the entire data set. For such applications, we propose using a dynamic backup region to reduce storage requirement by only maintaining copies of frequently modified objects in the backup. Although some applications might experience higher write latency, applications with skewed access patterns can achieve access latencies close to that of a full backup with smaller storage requirement.

Applications with high levels of spatial locality for writes benefit the most from using a dynamic backup region. For example, our experiments show that, simply maintaining the internal nodes (nodes closer to the root) of a persistent B+Tree in the backup region significantly improves the latency of insert and delete operations on the B+Tree.

Kamino-Tx-Dynamic extends Kamino-Tx-Simple by using a partial backup. Its architecture is shown in Figure 7. There are two ways in which Kamino-Tx-Dynamic differs from Kamino-Tx-Simple. First, the backup version of the heap now contains only a few objects (configurable size). More specifically, it contains the most frequently modified objects. Second, it contains a new look-up table that helps the log manager identify objects whose copies exist.

The look-up table is a persistent concurrent hash-table [13] and an LRU queue in volatile memory. The hash-table is a mapping between persistent objects and their offsets in the partial backup region. Since the partial backup cannot hold all persistent objects, we use an LRU queue to replace least recently updated objects with those which are about to get updated and do not have a copy. When the Log Manager encounters an object whose copy does not exist in the dynamic backup region, it creates an entry for it by booting out the least recently used object as reported by the look-up table.

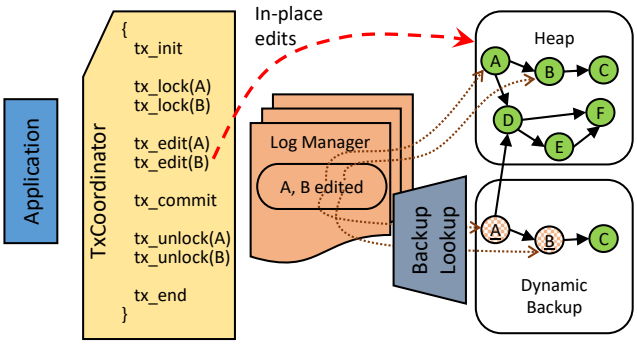

Figure 7: Architecture of Kamino-Tx-Dynamic. Only a partial backup is maintained where copies of only the most frequently modified objects are stored. A backup look-up table helps the log manager look up objects whose copies exist. If not, the log manager creates the copies on demand and inserts them in the dynamic backup region which is maintained in an LRU manner.

This enables keeping frequently modified objects inside the partial backup.

The storage requirement of the dynamic backup regions can be tuned by the application. For instance, if the application expects a write working set size to be 20% of the entire data set then setting the α parameter to 0.2 is adequate. The storage requirement of Kamino-Tx-Dynamic is therefore $(1+\alpha) \times dataSize$.

Note that the skew in data accesses at this level is not to be confused with dependency between transactions. These are orthogonal features of an application. Applications with skewed access patterns may not have any dependent transactions if the inter-arrival time of transactions working with the same set of objects is sufficiently large.

5. Replicated Kamino-Tx

Primary-Backup replication across multiple machines is a common technique used for fault-tolerance and high-performance. For example, Chain Replication [30] and its variants are used extensively in research and large-scale industrial systems [3, 7, 16, 28, 29].

Such systems need $f+1$ replicas to tolerate f failures. In Chain Replication, replicas are organized on a chain where the head of the chain receives all the write requests while the tail of the chain receives all reads. Writes propagate through the chain via each replica all the way to the tail where it is committed to the client.

Each replica performs an operation and then forwards it downstream. Such operations are received by the replicas in persistent operation queues. Forwarded operations are stored in an in-flight queue. Garbage collection messages propagate up the chain from the tail once they have been committed and are used to cleanup operations buffered by in-flight queues on the replicas. The tail also directly notifies the client when an operation completes. Furthermore, the first replica which is the head of the chain determines the global ordering of operations for all the replicas downstream. Finally, read op-

Replication Method	#Servers	Storage Requirement	Dependent Transaction Latency	Independent Transaction Latency
Traditional Chain	$f+1$	$(f+1)\times dataSize$	$(f+1)\times(l_c+l_n+l_t)$	$(f+1)\times(l_c+l_n+l_t)$
Kamino-Tx-Simple Chain	$f+1$	$2\times(f+1)\times dataSize$	$(f+1)\times(l_n+l_t)$	$(f+1)\times(l_n+l_t)$
Kamino-Tx-Dynamic Chain	$f+1$	$(1+\alpha)\times(f+1)\times dataSize$	$(f+1)\times(l_n+l_t)$	$(f+1)\times(l_n+l_t)$
Kamino-Tx-Amortized Chain	$f+2$	$(f+2+\alpha)\times dataSize$	$2\times(f+1)\times(l_n+l_t)$	$(f+1)\times(l_n+l_t)$

Table 1: Comparison between different Kamino-Tx schemes and traditional chain replication for transactions. f = the number of failures to tolerate. α = the proportion of the total heap that the dynamic backup can hold. *dataSize* is the size of the heap. l_t, l_c and l_n are the transaction execution, copying and network hop latencies respectively.

erations are always performed only on the tail. Due to this strict ordering, Chain Replication provides applications with the strong guarantee of Linearizability.

Replicating an NVM based persistent object store for fault-tolerance will only increase the latency of transactions, with copies created on the critical path on each node only adding to this latency. Furthermore, the undo/CoW copies created for each transaction can only be deleted by the garbage collection messages that are propagated upstream.

While naively applying Kamino-Tx-Simple or Kamino-Tx-Dynamic at each replica can avoid copying data on the critical path of transactions, they will unfortunately result in large storage requirement of $2\times(f+1)\times dataSize$ or $(f+1)\times(1+\alpha)\times dataSize$ respectively. Table 1 compares the number of servers needed and the storage requirement for various chain replication mechanisms.

This motivates our question: *How can we minimize storage requirement in a Chain Replicated NVM-based object store while avoiding the runtime overhead of copying data in the critical path at each replica for transactional operations?*

We start with the observation that the head node in a chain determines the ordering of transactions. Hence, the head node can decide if a transaction should commit or abort. In the case of an aborted transaction, maintaining a local backup copy at the head will enable the head node to roll back locally, respond to an aborted transaction immediately, and admit only committed transactions to the chain. Given this advantage of maintaining a backup at the head, we ask if we can get away with no backup copies at the other replicas?

The backup copy in Kamino-Tx-Dynamic is used to roll back in-place updates for aborted and incomplete transactions. Given that transactions that abort at the head are not admitted downstream in the chain, a backup copy at any non-head replica may only help with rolling back an incomplete transaction. A local ongoing transaction is considered incomplete when a replica reboots and recovers before its failure is detected thereby potentially causing torn or incomplete writes to leave data in an inconsistent state. We observe that replicas are backup copies after all. Downstream nodes in a chain (i.e., nodes closer to the tail) have older data compared to upstream nodes in the chain. So we can potentially use the immediate downstream node in the chain as a backup copy. Likewise, we can potentially use the immediate upstream node in a chain to roll forward.

The approach above mandates that replicas in a chain should never process dependent transactions concurrently, a property easily enforceable at the head of the chain; for every transaction processed, the head node holds appropriate locks until the tail commits and sends an unlock message to head, thus never allowing a dependent transaction to be admitted into the chain until the locks are released.

Unfortunately, safely tolerating f failures with $f+1$ nodes is impossible when we avoid maintaining a backup at non-head nodes. Consider the scenario where f replicas fail and stop, while one non-head replica reboots and recovers before its failure is detected. This replica might have an incomplete transaction and it has no way of rolling back to a consistent state. To use replicas in a chain as backup copies to roll back incomplete transactions, we always need at least two nodes in a chain. We propose Kamino-Tx-Chain, an extension to Chain Replication, that uses $f+2$ replicas that perform in-place transactional updates to tolerate f failures. Kamino-Tx-Chain has a modest total storage requirement of $(f+2+\alpha)\times dataSize$ ($\alpha\varepsilon(0,1]$) (details shown in Table 1).

5.1 The Kamino-Tx-Chain Protocol

The goals of Kamino-Tx-Chain protocol are the following: (1) Tolerate f replica failures, including transient reboots that recover before failures are detected, (2) Maintain the strong consistency guarantees that Chain Replication provides and (3) Avoid copying of data in the critical path. At a high level, Kamino-Tx-Chain contains $f+2$ replicas, a Transaction Coordinator and a backup copy at the head, and a Log Manager at every replica node as shown in Figure 8.

Specifically, the head replica uses either a Kamino-Tx-Simple technique ($\alpha=1$) or a Kamino-Tx-Dynamic technique with smaller α. All the other replicas modify the objects in place without creating any copies of data or maintaining backup versions of data. At a high level, Kamino-Tx-Chain works similar to the basic Chain Replication protocol except some modifications that we discus next.

Transactions in Kamino-Tx-Chain are executed as follows: All transactions arrive at the head node. Head performs admission control of dependent transactions. Locks for admitted transactions are acquired and held until the head receives an acknowledgment for it from the tail. The Transaction Coordinator starts executing the transaction on the head. If the transaction commits then the Transaction Coordinator forwards the transaction down the chain. *That is, only those*

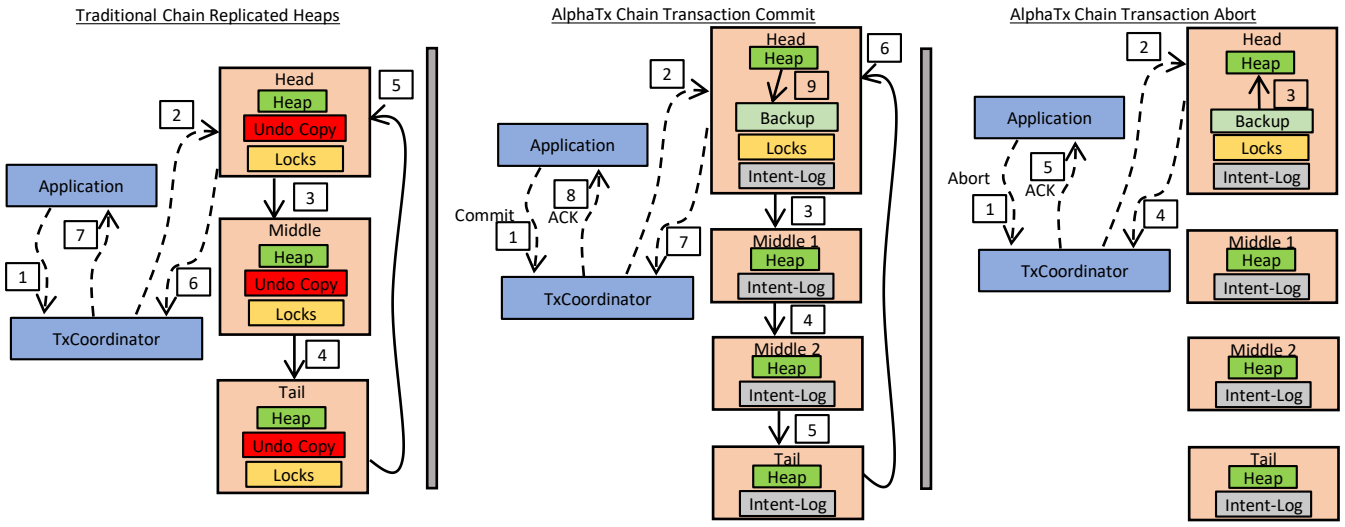

Figure 8: Kamino-Tx-Chain uses one additional replica to eliminate copying in the critical path for tolerating f failures. Traditional chain replicas lock and copy data in the critical path where are Kamino-Tx chain replicas other than the head do not need to copy data at all. Also, the head copies data off the critical path for committed transactions while aborts are performed without the rest of the chain's involvement (dashed arrows are local function calls on the head node while the solid arrows are remote calls).

transactions that have been prepared for commit by the application are sent down the chain. Each replica receives the transaction in the form of a remote procedure call with a named function and the arguments to the function. This is same local function call the client makes on the head replica.

The replicas buffer such calls in an input queue in non-volatile memory before the receipt is acknowledged upstream. Each replica executes transactions in its input queue by modifying objects in place without creating any copies. It then forwards the transaction downstream and moves the transaction from its input queue to a buffered queue of in-flight transactions. The transaction completes when the tail replica commits and notifies the Transaction Coordinator.

In Chain Replication, the tail sends the client a final acknowledgment for a write. In Kamino-Tx-Chain, the tail sends the final acknowledgment to the head instead of the client, before sending clean-up acknowledgments up the chain as before. The client in Kamino-Tx resides on the head, so the final call to the client is a local up-call on the head. The clean-up acknowledgments travel upstream in the chain and serve to remove corresponding entries from the buffered queue of in-flight transactions.

The head node releases locks of a transaction when two conditions are met: (1) On receiving a transaction completion acknowledgment from the tail, and (2) the Transaction Coordinator copies modified data to the backup region. As aborted transaction are never admitted downstream in the chain, they can be performed by the Transaction Coordinator similar to the un-replicated case (Figure 8 abort case).

All replicas maintain a Log Manager with write intents. These intent logs are needed for recovering from failures; they are used to determine the set of items corresponding to incomplete transactions that need to be rolled back or forward as the case may be. The intent-logs are also deleted when clean-up acknowledgments show up.

5.2 Handling Fail-Stop Failures

Fail-stop failures in Kamino-Tx-Chain are handled similar to traditional Chain Replication. The chain is repaired, the membership view of the chain changes, and any new replica joins as the tail of the chain (with appropriate state transfer from its predecessor).

On a tail failure, similar to Chain Replication, the new tail sends the head completion acknowledgments for all in-flight transactions, i.e., transactions the new tail forwarded but did not receive a clean-up acknowledgment for.

Head node failures require special care since the backup and the locks need to be recovered. The new head goes through its Log Manager's intent logs, creates a local backup, and constructs a lock set corresponding to all in-flight transactions. This is a conservative set, as the tail might have already committed some of these transactions and alerted the old head. Instead of waiting for the clean-up acknowledgments to update the lock set, the new head queries the tail to determine the last transaction t acknowledged by the tail. The head can unlock data for all in-flight transactions that came before t, but still hold locks for all the objects in the write-set of every other in-flight transaction.

5.3 Handling Quick Reboots

In Chain Replication, if a replica reboots and recovers, but has been offline for longer than the failure detection period, then it is considered a fail stop failure. But if a replica reboots and recovers before it is considered failed, it has to follow a protocol to safely rejoin the chain. Kamino-Tx-Chain needs to guard against incomplete transactions for such quick reboots.

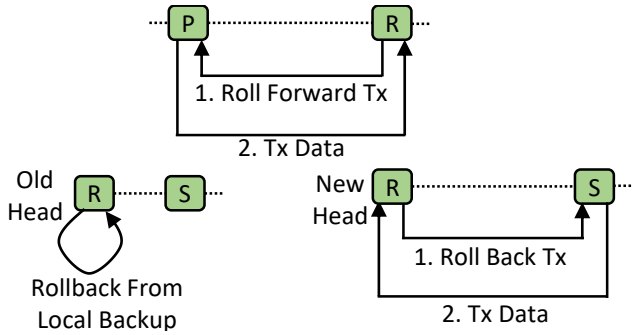

Figure 9: Data integrity for incomplete transactions on reboots. (R is the rebooting replica, P is predecessor and S is successor.)

The rebooted replica first contacts the membership manager (Zookeeper instance) requesting to rejoin the chain with the *viewID* it believes is current. *viewID* is maintained by the membership manager and represents a concrete instance of a chain. All messages carry a *viewID* and replicas reject messages with an older *viewID*. The membership manger replies to the replica with its predecessor and successor information. The membership view might have changed and more importantly the replica's previous successor or predecessor might have changed, causing the chain to follow a chain repair protocol for fail-stop failures described earlier, but this is done only after fixing incomplete transactions first.

A recovering replica first identifies data items in the write set of incomplete transactions using its intent logs. For these data items, the replica does one of three things: (1) If it is a non-head node, it rolls forward changes from its assigned predecessor, (2) If it is still the head of the chain, it rolls back from its local backup, or (3) If it is a new head of the chain, it rolls back changes from its assigned successor (Figure 9). As the head of a chain never admits dependent transactions it is always safe to roll forward from a predecessor or roll back from a successor in the recovery protocol described above.

As an aside, the data integrity protocol, used when all nodes of a chain are offline long enough to be marked as failed and some of them reboot, is identical to the recovery protocol described above and needs at least two replicas from the last know version of the chain to work.

6. Implementation

In this section, we describe the implementation of Kamino-Tx's components. We start with describing the programming interface and continue this section with implementation details of Kamino-Tx-Simple and Kamino-Tx-Dynamic.

6.1 Programming Interface

Kamino-Tx does not mandate using specific APIs and can be integrated with any programming interface which provides information about transaction boundaries and write intents. However, it is most suited for transactional heaps for NVM where each transaction modifies small byte ranges in several objects at a time. In this paper, we take Intel's non-volatile memory library [20] as an example and replace its atomicity

```
struct ObjectType1 { char attr[255]; };
struct ObjectType2 { int attr; };
TX_BEGIN(pop) {
  // declare write intents
  TX_ADD(obj1);
  TX_ADD(obj2);
  // cast & get virtual memory pointers
  ObjectType1 *obj1_p = D_RW(obj1);
  ObjectType2 *obj2_p = D_RW(obj2);
  // modify objects as needed
  strcpy(obj1_p->attr, "NewValue");
  obj2_p->attr = strlen(obj1_p->attr);
}
TX_END { } //flush changes to NVM for commits
//or roll back objects for aborts
```

Figure 10: Using Intel's NVML to write a sample transaction. In the unmodified library, TX_ADD calls would add the objects to an undo log by creating copies. In Kamino-Tx's version of the library, data is not copied in the critical path.

scheme with Kamino-Tx's. We use Intel's NVML library which provides low level persistent memory support as well as transactional object store semantics. Figure 10 shows a sample transaction using NVML.

Kamino-Tx is implemented as a user-level library implemented in C and can be compiled with any program which uses Intel's NVML. It basically redefines the functionality of a set of interfaces defined by NVML to extract transaction boundaries and write intents. Table 2 shows the list of functions that we redefine to feed Kamino-Tx with the necessary information. This implies that any application that works with NVML just needs to be re-linked to work with Kamino-Tx. The API for the replicated system is also exactly the same. The library abstracts the replicas from the application that runs on the head node of the chain.

The programming interface interacts with the Log Manager by requesting a new transaction to be created, declaring an intent to write, demanding a flush operation on persistent memory regions modified by a transaction, requesting an object to be transactionally freed, issuing a transaction abort or initiating a transaction commit. Based on these interactions, Log Manager maintains necessary information for each transaction which the Transaction Coordinator will use to transfer data between the main and the backup versions.

Memory allocator is implemented in Kamino-Tx as a part of the persistent heap using persistent memory objects. Therefore, allocations and deallocations are simply treated as modifications to persistent metadata objects that the application atomically modifies indirectly via the object allocation and deallocation calls made within transactions.

6.2 Implementation of the Log Manager

We implement the Log Manager as a software module in our modification to NVML. Log Manager maintains a global lock-free persistent intent log along with per thread private NVM region. During the first execution of the program, a region of the persistent memory called intent-log is allocated

Function signature	NVML	Kamino-Tx
TX_ZALLOC(s)	Allocates an object of size *s*.	Allocates the object & reports the allocation to Log Manager.
TX_ADD(obj)	Creates an undo-log entry for *obj*.	Notifies the Log Manager about an intent to write to *obj* with a lock.
TX_FREE(obj)	Transactionally deallocates an object	Reports the deallocation to Log Manager.
TX_COMMIT()	Undo log deleted.	Backup rolled forward using main version and Log Manager. Locks released.
TX_ABORT()	Aborted using the undo log.	Main version rolled back using backup and Log Manager. Locks released.

Table 2: Semantics of *NVML* interfaces before and after integration with Kamino-Tx

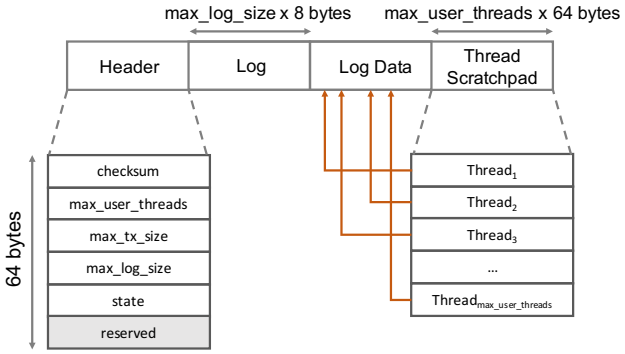

Figure 11: The storage layout of Log Manager

to Log Manager which is split across the threads and reused for transactions.

Figure 11 shows the layout of this region. The header section contains the following information: maximum number of supported threads, the maximum object size, maximum number of log entries, the state of the region and a checksum field which is used for ensuring consistency. There are three different values for the state field: Running, Committed and Aborted. Based on the state, Log Manager detects failures and starts the recovery process.

At the beginning of each transaction, Log Manager initializes an intent-log and gives it to the thread executing the transaction. Once the transaction is either committed or aborted, this intent-log is appended to a global intent-chain and stays there until consumed by Transaction Coordinator. Transaction Coordinator can read from this log and decide on the proper action based on the state of each transaction.

Log Manager must know in advance three types of operations a program performs: allocation of a new persistent memory region, deallocation of an existing region and modification of a persistent object. We modify the TX_ZALLOC, TX_FREE and TX_ADD primitives in NVML to detect such intents without any additional programming overhead for the developer designing transactions. Each intent-log entry is an object address that fits within a cache line such that they can be persisted without being torn. The addresses are generated by the unmodified library and we use the same addressing mechanism for storing in the intent-logs. The Log Manager calls one flush instruction after all the write intents are declared while the Transaction Coordinator calls one flush instruction for persisting the changes.

6.3 Transaction Coordinator

Transaction Coordinator is implemented as a combination of stub code in commit and abort calls, and also a background thread which utilizes the information maintained by Log Manager to keep backup version consistent with the main version. During normal operation, Transaction Coordinator waits for the first active transaction to commit or abort. Based on the state of the transaction, Transaction Coordinator will either re-execute the transaction on the backup region or undo it by copying the data regions back to the main copy. All the information required for this process, including the list of modified memory regions and the state of transactions, is provided by the intent-logs.

Once Transaction Coordinator ensures durability of the backup, it releases locks held on modified objects. This allows other transactions trying to modify those objects to proceed. Finally, Transaction Coordinator will remove the intent-log corresponding to the processed transaction from Log Manager and reads the record for next transaction.

6.4 Dynamic Backup

We extend the Kamino-Tx implementation with a LRU queue and a persistent hash table to enable dynamic backup regions. For every attempt to update a persistent object, Transaction Coordinator first checks if the object is present in the LRU queue. If it exists, then Log Manager is used to enable atomic in-place updates. The coordinator uses the persistent hash table later to synchronize the copy with the actual object. Otherwise, the least recently used object in the dynamic backup region is replaced with a copy of the object which is going to be modified. However, locked objects are never evicted to ensure safety, that is pending objects are never candidates for eviction.

The hash table is also updated accordingly to record the mapping between the persistent data object and the location of its copy. All the objects currently part of transactions are locked in the hash table so they are not replaced.

7. Evaluation

We evaluate Kamino-Tx to demonstrate that the latency of transactions decreases and transactional throughput increases. In order to evaluate Kamino-Tx, we have designed and implemented a key-value store that uses a NVML based persistent B+Tree that we implement. YCSB [10] workloads are used to evaluate latency and throughput of various con-

YCSB Workload	A	B	C	D	F
Read	50	95	100	95	50
Update	50	5	-	-	-
Insert	-	-	-	5	-
Read & Update	-	-	-	-	50

Table 3: The % of different operations in YCSB workloads.

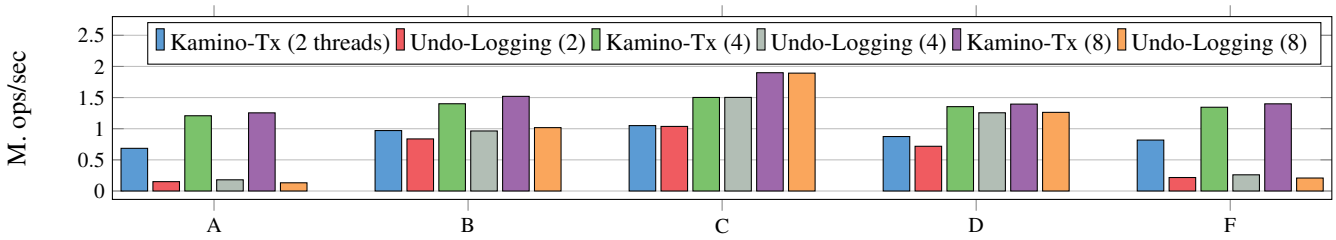

Figure 12: YCSB throughput with Kamino-Tx-Simple and undo-logging (Intel's NVML) as the number of threads vary from two to eight.

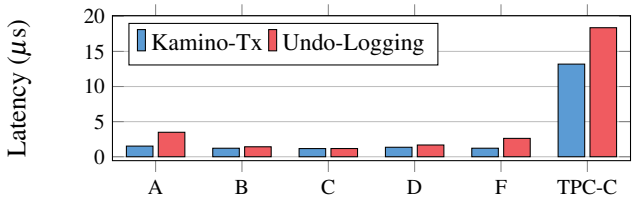

Figure 13: YCSB latency with Kamino-Tx-Simple and undo-logging (Intel's NVML).

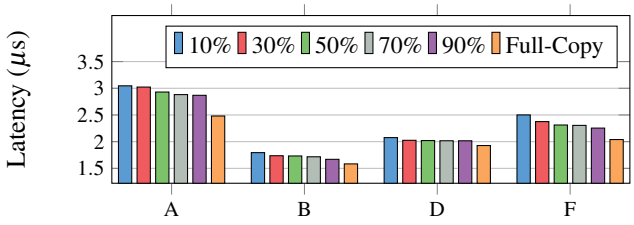

Figure 14: YCSB latency with full and partial backups.

figurations of the key-value store. Table 3 describes each workload in terms of the set of operations performed on 10M key-value pairs, each of size 1KB.

Our baseline implementation utilizes unmodified Intel's NVML library, which provides atomicity through undo-logging [20]. We compare this undo-logging against the Kamino-Tx atomicity implementation in our modified version of NVML library.

For our experiments, we use five different deployments of the key-value store to measure latency and throughput:

- Undo-Logging: baseline version with Intel's NVML.
- Kamino-Tx-Simple: in-place update using a full copy.
- Kamino-Tx-Dynamic: in-place update using only $(100 \times \alpha)\%$ more storage for backup.
- Traditional Chain and Kamino-Tx-Chain: each tolerating two failures.

All experiments are performed on Microsoft Azure A9 instances each with 16 cores, 112 GB of memory and Microsoft Azure RDMA networking with a 32Gbps Infiniband network. We use DRAM to emulate NVM since NVDIMM is the only available NVM technology today. Thus, our results will hold for NVDIMM which is the fastest NVM available today. For other slower NVMs, the benefits of Kamino-Tx would only be larger since the copying would take longer.

7.1 Kamino-Tx's Performance

The main goal of Kamino-Tx is removing logging overhead from the critical path. Figure 13 shows the performance benefit of using Kamino-Tx-Simple over our baseline implementation using Intel's NVML. For write-intensive workloads, Kamino-Tx performs up to 2.33x faster than the baseline system. Cache flushes, transactional allocation and software needed for maintaining undo-logs comprises most of the overhead for the baseline system. Both systems provide similar latency for workload C which is 100% read.

Figure 12 shows throughput numbers for Kamino-Tx-Simple and the baseline system. Except for workload C, which only includes read operations, Kamino-Tx offers higher throughput compared to the undo-logging counterpart for up to 9.5x. Moreover, Kamino-Tx-Simple offers up to 40% better throughput for the TPC-C workload.

Dependent transactions. To measure the impact of locking on dependent transactions we performed specialized key-value transactions where we synthetically controlled the keys in requests. We perform the experiment with 80% look-up operations and 20% insert operations. We compare two settings where all the insert operations are performed on the same key. In the first setting the insert operations are spaced out uniformly between look-up operations while in the second setting insert operations are performed in bursts on the same key. We find that for undo-logging the average latency of the operations was unaffected (within the margin of error). However, for Kamino-Tx we find that the average latency increases by 8% with substantial increase in average latency for the insert operations of over 30%.

Worst-case performance. The benefit of Kamino-Tx is greatest when the distance between dependant transactions is big enough to allow copying to the backup off the critical path. Therefore, the worst-case scenario for Kamino-Tx is continuously executing a transaction that updates the same object. We compare the worst-case performance of Kamino-Tx-Simple against NVML's undo-logging using a benchmark that creates 1 to 8 threads, each creating an object (varying in size between 64 and 4,096 bytes) and transactionally updating it for 100 K times. For objects smaller than 1 KB, Kamino-Tx-Simple offers lower latency compared to undo-logging by obviating log allocation. Both techniques offer similar latency for larger objects since the majority of the transaction time is spent on copying. Also, they offer similar throughput as they hit the memory bandwidth limit.

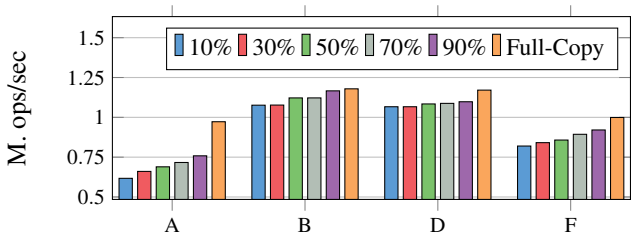

Figure 15: YCSB throughput with full and partial backups.

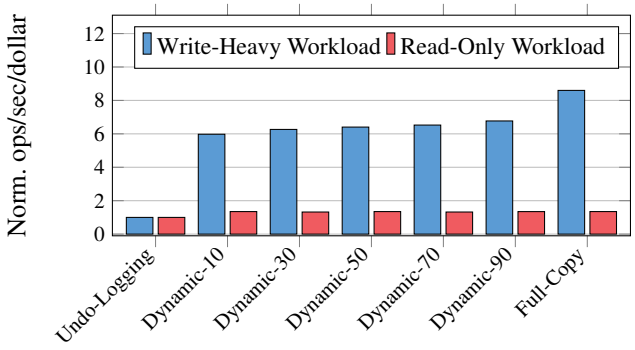

Figure 16: Normalized performance per dollar for different backup configurations and undo-logging.

7.2 Kamino-Tx-Dynamic's Performance

Figures 14 and 15 compare Kamino-Tx-Dynamic to Kamino-Tx-Simple in terms of latency and throughput, respectively. We use five different storage budgets for Kamino-Tx-Dynamic, ranging from 10% to 90% of the size of original data. Although Kamino-Tx-Simple outperforms Kamino-Tx-Dynamic by up to 1.5x for write-intensive workloads, Kamino-Tx-Dynamic can reduce storage cost by 50% for a 5% decrease in average throughput for read-heavy workloads. The latency values are higher than we expected because of a coarse-granularity lock on the hash table needed for the backup look-up. We expect the latencies to fall further with a more fine granularity locking approach.

We compare Kamino-Tx against undo-logging in terms of attainable throughput per dollars spent. Here, we use average throughput numbers for YCSB workloads A and B as well as estimated TCO numbers from AWS Total Cost of Ownership (TCO) Calculator [2]. TCO numbers are calculated based on Microsoft Azure A9 VMs with 16 cores and 112 GB of memory.

Figure 16 shows normalized values for our B+Tree implementation with undo-logging, Kamino-Tx-Simple and Kamino-Tx-Dynamic. The blue and red bars show numbers for write-intensive and mostly read workloads, respectively. Kamino-Tx-Simple offers up to 8.6x more throughput for each dollar spent in presence of write-intensive workloads. Although Kamino-Tx-Simple is more cost-efficient for write-intensive workloads, Kamino-Tx-Dynamic could be a better choice for read-heavy workloads, specially for those with high levels of data locality for writes.

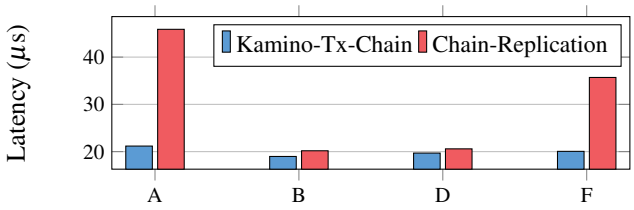

Figure 17: YCSB latency for Kamino-Tx-Chain and traditional chain replication each tolerating two failures.

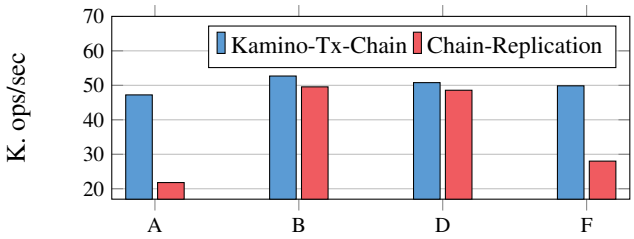

Figure 18: YCSB throughput for Kamino-Tx-Chain and traditional chain replication configured to survive two failures.

7.3 Kamino-Tx-Chain's Performance

Figure 17 shows the latency numbers for replicated schemes. For write-intensive workloads, Kamino-Tx-Chain is up to 2.2X faster than the traditional chain. Kamino-Tx obtains these benefits because it does not create any copies of the data (including keys and values) in the critical path.

Figure 18 shows throughput numbers for Kamino-Tx-Chain and our implementation of traditional chain replication. By requiring 33% extra storage space compared to the vanilla version, Kamino-Tx-Chain can offer up to 2.2x better throughput for write-intensive workloads.

8. Conclusion

We present Kamino-Tx which provides lightweight transaction support for NVMM. Kamino-Tx offers atomic in-place updates and obviates creating copies of data items in the critical path while providing crash consistency using a copy of data items maintained asynchronously. We study an alternate approach to reduce the cost of Kamino-Tx by only maintaining copies of only hot data items. Evaluation results show this provides close-to-optimal performance with small storage overheads. Furthermore, we also present a new variant of chain replication that exploits the copies of data at replicas for providing crash consistency. Kamino-Tx increases throughput by up to 9.5x for unreplicated systems and up to 2.2x for replicated settings with an additional copy of data.

Acknowledgments

We would like to thank to our shepherd, Pascal Felber, as well as the anonymous Eurosys reviewers for their helpful comments and feedback.

References

[1] System Support for NVMs in Linux. http://nvdimm.wiki.kernel.org.

[2] Amazon Web Services, Inc. AWS Total Cost of Ownership (TCO) Calculator, May 2016. Available at https://awstcocalculator.com.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 1–14. ACM, 2009.

[4] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's Talk About Storage and Recovery Methods for Non-volatile Memory Database Systems. In *Proceedings of ACM SIGMOD 2015*, 2015.

[5] K. Bhandari, D. R. Chakrabarti, and H. Boehm. Implications of CPU Caching on Byte-addressable Non-volatile Memory Programming, 2012.

[6] B. Bridge. NVM Support for C Applications, 2015. Available at http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP'11, pages 143–157, New York, NY, USA, 2011. ACM.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of ACM SOSP 2009*, 2009.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154, New York, NY, USA, 2010. ACM.

[11] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[12] S. R. Duloor, S. Kumar, A. Keshavamurthy, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference of Computer Systems*, 2014.

[13] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 371–384, Lombard, IL, 2013. USENIX.

[14] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High Performance Database Logging Using Storage Class Memory. In *Proc. 27th IEEE ICDE'11*, Hanover, Germany, 2011.

[15] M. J. Franklin. Concurrency Control and Recovery. *The Computer Science and Engineering Handbook*, pages 1058–1077, 1997.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[17] E. Giles, K. Doshi, and P. Varman. Bridging the Programming Gap Between Persistent and Volatile Memory Using WrAP. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF'13, pages 30:1–30:10, New York, NY, USA, 2013. ACM.

[18] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Comput. Surv.*, 13 (2):223–242, June 1981.

[19] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware Logging in Transaction Systems. In *Proceedings of the Forty First International Conference on Very Large Data Bases*, Aug. 2015.

[20] Intel Corporation. Persistent Memory Programming, 2015. Available at http://pmem.io/nvml/.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proc. USENIX OSDI'16*, Savannah, GA, 2016.

[22] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proc. FAST'13*, San Jose, CA, Feb. 2013.

[23] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[24] MongoDB. http://mongodb.com.

[25] D. Narayanan and O. Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[26] Oracle Corporation. The InnoDB Recovery Process, 2016. Available at https://dev.mysql.com/doc/refman/5.1/en/innodb-recovery.html.

[27] S. Park, T. Kelly, and K. Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European*

Conference on Computer Systems, EuroSys'13, pages 225–238, New York, NY, USA, 2013. ACM.

[28] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini. Flex-KV: Enabling High-performance and Flexible KV Systems. In *Proceedings of the 2012 Workshop on Management of Big Data Systems*, MBDS'12, pages 19–24, New York, NY, USA, 2012. ACM.

[29] J. Terrace and M. J. Freedman. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, Berkeley, CA, USA, 2009. USENIX Association.

[30] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[31] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies*, 2011.

[32] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[33] T. Wang and R. Johnson. Scalable Logging Through Emerging Non-volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, June 2014.

[34] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 323–338, Santa Clara, CA, Feb. 2016. USENIX Association.

[35] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A Reliable and Highly-available Non-volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 3–18, New York, NY, USA, 2015. ACM.

[36] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the Performance Gap Between Systems with and without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.