Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration

Sudipto Das[‡] Shoji Nishimura[§*] Divyakant Agrawal[‡] Amr El Abbadi[‡]

[‡]University of California, Santa Barbara
Santa Barbara, CA 93106-5110, USA
{sudipto, agrawal, amr}@cs.ucsb.edu

[§]NEC Corporation
Kawasaki, Kanagawa 211-8666, Japan
s-nishimura@bk.jp.nec.com

ABSTRACT

Database systems serving cloud platforms must serve large numbers of applications (or *tenants*). In addition to managing tenants with small data footprints, different schemas, and variable load patterns, such multitenant data platforms must minimize their operating costs by efficient resource sharing. When deployed over a pay-per-use infrastructure, elastic scaling and load balancing, enabled by low cost *live migration* of tenant databases, is critical to tolerate load variations while minimizing operating cost. However, existing databases—relational databases and Key-Value stores alike—lack low cost live migration techniques, thus resulting in heavy performance impact during elastic scaling. We present *Albatross*, a technique for live migration in a multitenant database serving OLTP style workloads where the persistent database image is stored in a network attached storage. Albatross migrates the database cache and the state of active transactions to ensure minimal impact on transaction execution while allowing transactions active during migration to continue execution. It also guarantees serializability while ensuring correctness during failures. Our evaluation using two OLTP benchmarks shows that Albatross can migrate a *live* tenant database with no aborted transactions, negligible impact on transaction latency and throughput both during and after migration, and an unavailability window as low as 300 ms.

1. INTRODUCTION

Cloud platforms hosting hundreds of thousands of applications pose novel challenges for the database management systems (DBMS) serving these platforms. A large fraction of these applications (called **tenants**) are characterized by small data footprints with unpredictable and erratic load patterns [14, 23]. **Multitenancy** allows effective resource sharing amongst these tenants, thus minimizing operating cost. For a database built on a pay-per-use cloud infrastructure, *elastic scaling* and *load balancing*—i.e. scaling up and down the size of a live system based on the load—are critical to ensure good performance while minimizing operating cost.

A multitenant database consolidates multiple tenant databases to a single node to improve resource utilization [1, 8]. Tenants are often consolidated to nodes (or servers) with capacity less than the combined peak resource requirements of the tenants co-located at a node. Such consolidation relies on the fact that peak resource usage is not frequent and peaks for different tenants are often temporally separated. However, to deal with unpredicted load patterns, migrating tenant databases is critical to ensure that the tenants' service level agreements (**SLA**) are met. When the resource requirements of a tenant changes, migration allows allocating more resources to the heavily loaded tenant while isolating other tenants from being impacted by the sudden increase in load of a co-located tenant.

To be effectively used for elasticity, database migration must be lightweight, i.e. with minimal service interruption and negligible performance impact. This feature is called **Live Migration** in the virtualization literature [5]. Due to static provisioning in enterprise infrastructures, elasticity and live migration were not critical for traditional relational database (RDBMSs). Even though most Key-Value stores [4, 6] support data migration for fault-tolerance or load balancing, they use heavyweight techniques such as stopping part of the database, migrating it to a new node, and restarting it at the destination. Such *stop and migrate* techniques (or simple optimizations) have a high performance penalty resulting from aborted transactions (due to the source node stopping the tenant) and high impact on transaction latency and throughput (due to the destination node starting with a cold cache). Existing DBMSs are therefore not amenable to elastic scaling.

We focus on designing an *efficient and low cost technique for live migration of a tenant database in a multitenant DBMS*. Live migration enables elasticity as a first class notion and eases database administration by decoupling a tenant's database from the node hosting it, thus allowing virtualization in the database tier. We propose **Albatross**,[1] the first end-to-end technique for live migration in shared storage database architectures executing OLTP workloads. In a shared storage DBMS, the persistent database image is stored in network attached storage (**NAS**) servers. This decoupled storage abstraction allows independent scaling and fault-tolerance of the storage layer [4, 9, 11, 18]. We assume a multitenancy model where multiple tenants share the same database process (shown to result in more effective resource sharing [8]) where live virtual machine (**VM**) migration techniques [5] cannot be used to migrate the individual tenant databases.

Albatross leverages the semantics of database systems to migrate the database cache and the state of transactions active during migration. In shared storage architectures, the persistent data of a tenant's database is stored in the NAS and therefore does not need migration. Migrating the database cache allows the tenant to start "warm" at the destination, thus minimizing the impact on transac-

[*]The author conducted this work as a visiting researcher at UCSB.

[1]The name Albatross is symbolic of the lightweight nature and efficiency of the technique that is typically attributed to Albatrosses.

Proceedings of the VLDB Endowment, Vol. 4, No. 8
Copyright 2011 VLDB Endowment 2150-8097/11/05... $ 10.00.

tion latency. To minimize the unavailability window, this copying of the state is performed iteratively while the source continues to serve transactions on the tenant database being migrated. Copying the state of transactions active during migration allows them to resume execution at the destination. Albatross, therefore, results in negligible impact from the tenants' perspective, thus allowing the system to effectively use migration while guaranteeing that (i) the tenants' SLAs are not violated, (ii) transaction execution is serializable, and (iii) migration is safe in spite of failures. Moreover, in Albatross, the destination node performs most of the work of copying the state, thus effectively relieving load on the overloaded source node.

We implemented Albatross in ElasTraS [9, 10], a scalable multitenant DBMS for the cloud. Our evaluation of Albatross using two OLTP benchmarks, YCSB [7] and TPC-C [20], shows that a live tenant database can be migrated with only $5-15\%$ average transaction latency increase immediately after migration, no transactions aborted in most cases, and an unavailability window as low as 300 ms. Moreover, Albatross has negligible impact on other tenants colocated at the source and destination of migration. Albatross's effectiveness is evident when compared to the tens of seconds to minutes of disruption when migrating a tenant in a traditional RDBMS such as MySQL and about $3-5$ second unavailability window for the *stop and migrate* technique implemented in ElasTraS. In addition, such heavy weight techniques result in a $300-400\%$ increase in latency of transactions executing immediately after migration.

The major contributions of this paper are as follows:

- We propose Albatross, the first lightweight live database migration technique for shared storage architectures.
- We propose four different metrics to evaluate the cost of live database migration techniques.
- We provide arguments justifying correctness of Albatross and characterize its behavior under different failure scenarios.
- We provide a thorough evaluation using a wide variety of transactional workloads and four different migration cost metrics.

Organization. Section 2 surveys different database multitenancy models, proposes migration cost measures, and reviews some known database migration techniques. Section 3 explains Albatross and Section 4 argues its correctness. Section 5 describes a prototype implementation that is evaluated in Section 6. Section 7 surveys related work and Section 8 concludes the paper.

2. PRELIMINARIES

2.1 Multitenancy Models

Multitenancy, i.e. resource sharing amongst different tenants, is important to serve applications that have small but varying resource requirements [14, 21, 23]. SaaS providers like Salesforce.com [21] are the most common examples of multitenancy in both the application as well as the database tier. A *tenant* is an application's database instance with its own set of clients and data. Different multitenancy models arise from resource sharing at different levels of abstraction; the *shared machine*, *shared process*, and *shared table* models are well known [14]. Salesforce.com [21] uses the *shared table* model where tenants share the database tables. ElasTraS [9], SQL Azure [1], and RelationalCloud [8] use the *shared process* model where the tenants share the database process. Xiong et al. [22] uses the *shared machine* model where tenants only share the physical hardware but have their independent VMs and database processes. Since different models store the tenants' data in different forms, a common logical notion of a **Tenant Cell** (or **Cell** for brevity) is used to represent a self-contained granule of application data, meta data, and state representing a tenant.

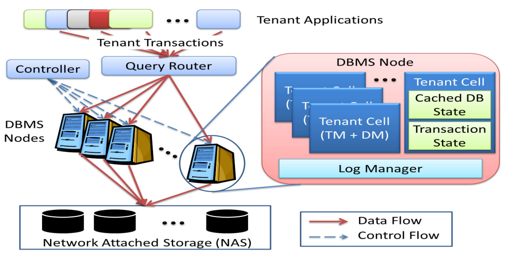


Figure 1: Reference database system model.

The multitenancy model also has implications on the system's performance and dynamics. For instance, the *shared machine* model provides strong VM level isolation and allows using VM migration techniques [5] for elastic scaling. However, as observed in a recent study [8], such a model results in up to an order of magnitude lower performance compared to the *shared process* model. On the other hand, the *shared table* model allows efficient sharing amongst tenants with small footprints but limits the tenant's schema and requires custom techniques for efficient query processing. The shared process model, therefore, provides a good balance of effective resource sharing, schema diversity, performance, and scale; we develop a live migration technique for this model.

2.2 Reference System Model

Our reference system model (see Figure 1) uses the shared process multitenancy model where a cell is entirely contained in a single database process which co-locates multiple cells. Application clients connect through a **query router** which abstracts physical database connections as logical connections between a tenant and its cell. Even though Figure 1 depicts the query router as a single logical unit, a deployment will have a distributed query router to scale to a large number of connections. The mapping of a cell to its server is stored as *system metadata* which is cached by the router.

A cluster of **DBMS nodes** serves the cells; each node has its own local transaction manager (**TM**) and data manager (**DM**). A TM consists of a *concurrency control* component for transaction execution and a *recovery component* to deal with failures. A cell is served by a single DBMS node, called its **owner**. The size of a cell is therefore limited by the capacity of a single DBMS node. This unique ownership allows transactions to execute efficiently without distributed synchronization amongst multiple DBMS nodes.

A network attached storage (**NAS**) provides a scalable, highly available, and fault-tolerant storage of the persistent image of the tenant databases. This *decoupling* of storage from ownership obviates the need to copy a tenant's data during migration. This architecture is however different from shared disk systems which use the disk for arbitration amongst concurrent transaction [2]. A system **controller** performs control operations including determining the cell to migrate, the destination, and the time to initiate migration.

2.3 Migration Cost

We now discuss four cost metrics to evaluate the effectiveness of a live database migration technique.

- **Service unavailability:** Duration of time for which a tenant is unavailable during migration.
- **No. of failed requests:** Number of well-formed requests that fail due to migration. Failed requests include both aborted transactions and failed operations. A transaction consists of one of more operations. Aborted transactions signify failed interactions with the system while failed operations signify the amount of work wasted as a result of an aborted transaction. The failed op-

erations account for transaction complexity; when a transaction with more operations aborts, more work is wasted for the tenant. We therefore include both types of failures in this cost metric.

- **Impact on response time:** Change in transaction latency (or response time) observed as a result of migration.
- **Data transfer overhead:** Data transferred during migration.

The first three cost metrics measure the external impact on the tenants and their SLAs while the last metric measures the internal performance impact. In a cloud data platform, a provider's service quality is measured through SLAs and satisfying them is foremost for customer satisfaction. A long unavailability window or a large number of failed requests resulting from migration might violate the availability SLA, thus resulting in a penalty. For instance, in Google AppEngine, if the availability drops below 99.9%, then tenants receive a service credit.[2] Similarly, low transaction latency is critical for good tenant performance and to guarantee the latency SLAs. For example, a response time higher than a threshold can incur a penalty in some service models [22]. A live migration technique must have minimal impact on tenant SLAs to be effective in elastic scaling.

2.4 Straightforward Migration Techniques

When using the shared machine multitenancy model, where each tenant has its independent database instance within a VM [22], live virtual machine migration techniques [5, 17] can be leveraged for elasticity. Such an approach, however, results in two levels of inefficiencies. First, migrating the entire VM incurs the additional (and unnecessary) overhead of copying the OS and VM state which can be avoided by a database migration technique cognizant of database semantics. Second, as reported in a recent study by Curino et al. [8], when compared to the shared process model, the shared machine model requires $2\times$ to $3\times$ more machines to serve the same number of tenants and for a given tenant assignment results in $6\times$ to $12\times$ less performance. Therefore, a shared process model is preferred for performance considerations where VM migration cannot be used to migrate individual tenant databases from a database process shared by multiple tenants.

Most traditional DBMSs do not support live migration since enterprise infrastructures were statically provisioned for peak expected load, and elasticity was not considered a first class feature in database systems. In such a scenario, a cell is migrated by stopping the cell at the source DBMS node, aborting all active transactions, flushing all the changes to the NAS, and restarting it at the destination. This approach, however, results in high migration cost: the tenant becomes unavailable during migration and all transactions active at the start of migration must be aborted. Furthermore, the entire database cache is lost when the cell is restarted at the destination DBMS node, thereby incurring a high post migration overhead for warming up the database cache. This approach, therefore, has a high impact on the tenant's SLA, thus preventing it from being effectively used for elastic scaling.

3. THE ALBATROSS TECHNIQUE

Albatross aims to have minimal impact on tenant SLAs while leveraging the semantics of the database structures for efficient database migration. This is achieved by iteratively transferring the database cache and the state of active transactions. For a two phase locking (**2PL**) based scheduler [2], the transaction state consists of the lock table; for an Optimistic Concurrency Control (**OCC**) [16] scheduler, this state consists of the read-write sets of active transactions and a subset of committed transactions. Figure 2 depicts the timeline of Albatross when migrating a cell (C_{migr}) from the source DBMS node (N_{src}) to the destination DBMS node (N_{dst}).

[2]`http://code.google.com/appengine/business/sla.html`.

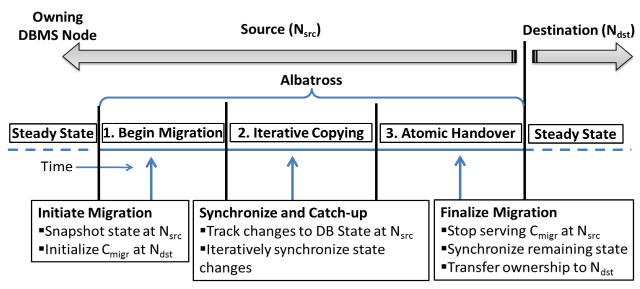

Owning DBMS Node
Source (N_{src})
Destination (N_{dst})
Albatross
Steady State
1. Begin Migration
2. Iterative Copying
3. Atomic Handover
Steady State
Time
Initiate Migration
▪Snapshot state at N_{src}
▪Initialize C_{migr} at N_{dst}
Synchronize and Catch-up
▪Track changes to DB State at N_{src}
▪Iteratively synchronize state changes
Finalize Migration
▪Stop serving C_{migr} at N_{src}
▪Synchronize remaining state
▪Transfer ownership to N_{dst}

Figure 2: Migration timeline (times not drawn to scale).

Phase 1: *Begin Migration:* Migration is initiated by creating a snapshot of the database cache at N_{src}. This snapshot is then copied to N_{dst}. N_{src} continues processing transactions while this copying is in progress.

Phase 2: *Iterative Copying:* Since N_{src} continues serving transactions for C_{migr} while N_{dst} is initialized with the snapshot, the cached state of C_{migr} at N_{dst} will lag that of N_{src}. In this iterative phase, at every iteration, N_{dst} tries to "catch-up" and synchronize the state of C_{migr} at N_{src} and N_{dst}. N_{src} tracks changes made to the database cache between two consecutive iterations. In iteration i, changes made to C_{migr}'s cache since the snapshot of iteration $i-1$ are copied to N_{dst}. This phase is terminated when approximately the same amount of state is transferred in consecutive iterations or a configurable maximum number of iterations have completed.

Phase 3: *Atomic Handover:* In this phase, the exclusive read/write access of C_{migr} (called **ownership**) is transferred from N_{src} to N_{dst}. N_{src} stops serving C_{migr}, copies the final un-synchronized database state and the state of active transactions to N_{dst}, flushes changes from *committed* transactions to the persistent storage, transfers control of C_{migr} to N_{dst}, and notifies the *query router* of the new location of C_{migr}. To ensure safety in the presence of failures, this operation is guaranteed to be *atomic*. The successful completion of this phase makes N_{dst} the owner of C_{migr} and completes the migration.

The iterative phase minimizes the amount of C_{migr}'s state to be copied and flushed in the handover phase, thus minimizing the unavailability window. In the case where the transaction logic is executed at the client, transactions are seamlessly transferred from N_{src} to N_{dst} without any loss of work. The handover phase copies the state of active transaction along with the database cache. For a 2PL scheduler, it copies the lock table state and reassigns the appropriate locks and latches at N_{dst}; for an OCC scheduler, it copies the read/write sets of the active transactions and that of a subset of committed transactions whose state is needed to validate new transactions. For a 2PL scheduler, updates of active transactions are done in place in the database cache and hence are copied over during the final copy phase; in OCC, the local writes of the active transactions are copied to N_{dst} along with the transaction state. For transactions executed as stored procedures, N_{src} tracks the invocation parameters of transactions active during migration. Any such transactions active at the start of the handover phase are aborted at N_{src} and are automatically restarted at N_{dst}. This allows migrating these transactions without moving the process state at N_{src}. Durability of transactions that committed at N_{src} is ensured by synchronizing the commit logs of the two nodes.

In the event of a failure, data safety is primary while progress towards successful completion of migration is secondary. Our failure model assumes reliable communication channels, node failures, and network partitions, but no malicious node behavior. Node failures do not lead to complete loss of data: either the node recovers or the data is recovered from the NAS where data persists beyond DBMS node failures. If either N_{src} or N_{dst} fails prior to Phase 3, migration of C_{migr} is aborted. Progress made in migration is not logged until Phase 3. If N_{src} fails during Phases 1 or 2, its state is recovered, but since there is no persistent information of migration in the commit log of N_{src}, the progress made in C_{migr}'s migration is lost during this recovery. N_{dst} eventually detects this failure and in turn aborts this migration. If N_{dst} fails, migration is again aborted since N_{dst} does not have any log entries for a migration in progress. Thus, in case of failure of either node, migration is aborted and the recovery of a node does not require coordination with any other node in the system.

The atomic handover phase (Phase 3) consists of the following major steps: (*i*) flushing changes from all committed transactions at N_{src}; (*ii*) synchronizing the remaining state of C_{migr} between N_{src} and N_{dst}; (*iii*) transferring ownership of C_{migr} from N_{src} to N_{dst}; and (*iv*) notifying the query router that all future transactions must be routed to N_{dst}. Steps (iii) and (iv) can only be performed once the Steps i and ii. Ownership transfer involves three participants—N_{src}, N_{dst}, and the query router—and must be atomic (i.e., either all or nothing). We perform this handover as a **transfer transaction** and a Two Phase Commit (2PC) protocol [13] with N_{src} as the coordinator guarantees atomicity in the presence of node failures. In the first phase, N_{src} executes steps (i) and (ii) in parallel, and solicits a vote from the participants. Once all the nodes acknowledge the operations and vote *yes*, the transfer transaction enters the second phase where N_{src} relinquishes control of C_{migr} and transfers it to N_{dst}. In the case when one of the participants votes *no*, this *transfer transaction* is aborted and N_{src} remains the owner of C_{migr}. This second step completes the transfer transaction at N_{src} which, after logging the outcome, notifies the participants about the decision. If the handover was successful, N_{dst} assumes ownership of C_{migr} once it receives the notification from N_{src}. Every protocol action is logged in the commit log of the respective nodes.

4. CORRECTNESS GUARANTEES

Migration correctness or *safety* implies that during normal operation or in case of a failure during migration, the system's state or data is not left in an inconsistent state as a result of migration.

DEFINITION 1. ***Safe Migration.*** *A migration technique is safe if the following conditions are met:* (*i*) Data Safety and Unique ownership: *The persistent image of a cell is consistent and only a single DBMS node* owns *a cell at any instant of time; and* (*ii*) Durability: *Updates from committed transactions are durable.*

We argue migration safety using a series of guarantees provided by Albatross and reason about how these guarantees are met.

GUARANTEE 1. ***Atomicity of handover.*** *In spite of failures, C_{migr} is owned by exactly one of N_{src} or N_{dst}.*

This is trivially satisfied when no failures occur. We now describe how logging and recovery ensures atomicity during failures. *At most one owner:* A failure in the first phase of the atomic handover protocol is handled similar to a failure during Phases 1 and 2—both N_{src} and N_{dst} recover normally and abort C_{migr}'s migration and N_{src} remains the owner. Failure in this phase does not need coordinated recovery. After receiving responses (both yes or no votes), N_{src} is ready to complete the *transfer transaction* and enters the second phase of atomic handover. Once the decision about the outcome is forced into N_{src}'s log, the transfer transaction enters the second phase. A failure in this phase requires coordinated recovery. If N_{src} decided to commit, N_{dst} is the new owner of C_{migr}, otherwise N_{src} continues as the owner. If N_{src} failed before notifying N_{dst}, N_{dst} must wait until the state of N_{src} is recovered before it starts serving C_{migr}. Therefore, the atomic handover protocol guarantees that there is at most one owner of C_{migr}.
At least one owner: A pathological condition arises when after committing the transfer transaction at N_{src}, both N_{src} and N_{dst} fail. Atomic handover guarantees that in such a scenario, both N_{src} and N_{dst} do not relinquish ownership of C_{migr}. If the handover was complete before N_{src} failed, when N_{src} recovers, it transfers ownership to N_{dst}. Otherwise N_{src} continues as the owner of C_{migr}. The synchronized recovery of N_{src} and N_{dst} guarantees at least one owner.

GUARANTEE 2. *Changes made by aborted transactions are neither persistently stored nor copied over during migration.*

This follows from the invariant that in the steady state, the combination of the database cache and the persistent disk image does not have changes from aborted transactions. In OCC, changes from uncommitted transactions are never publicly visible. In locking based schedulers, the cache or the persistent database image might have changes from uncommitted transactions. Such changes are undone if a transaction aborts. Any such changes copied over during the iterative phases are guaranteed to be undone during the first round of the atomic handover phase.

GUARANTEE 3. *Changes made by committed transactions are persistent and never lost during migration.*

Cache flush during the handover phase ensures that writes from transactions that have committed at N_{src}, are persistent. The log entries of such committed transactions on C_{migr} are discarded at N_{src} after successful migration.

GUARANTEE 4. *Migrating active transactions does not violate the durability condition even if the commit log at N_{src} is discarded after successful migration.*

This is ensured since Albatross copies the commit log entries for transactions active during migration to N_{dst}, which are then forced to N_{dst}'s commit log when these transactions commit.

Guarantee 1 ensures *data safety* and Guarantees 2, 3, and 4 together ensure *durability*, thus guaranteeing the safety of Albatross. Therefore, in the presence of a failure of either N_{src} or N_{dst}, the migration process is aborted without jeopardizing the safety.

GUARANTEE 5. ***Serializability.*** *Copying the transaction state in the final handover phase of Albatross ensures serializable transaction execution after migration.*

OCC guarantees serializability by validating transactions against conflicts with other concurrent and committed transactions. The handover phase copies the state of active transactions and that of a subset of transactions that committed after the earliest of the active transactions started. Therefore, all such active transactions can be validated at N_{dst} and checked for conflicts.

For a 2PL Scheduler, the two phase locking rule ensures serializability of a locking based scheduler. The final handover phase copies the state of the lock table such that active transactions have the locks that were granted to them at N_{src} when they resume execution at N_{dst}. Therefore, a transaction continues to acquire locks using the two phase rule at N_{dst}, thus ensuring serializability.

5. IMPLEMENTATION DETAILS

We implemented Albatross in ElasTraS [9, 10], a multitenant DBMS for the cloud. ElasTraS's architecture is similar to the abstract system model depicted in Figure 1. The *DBMS Nodes* are called *Owning Transaction Managers (**OTM**)* which *own* a number of cells and provide transactional guarantees on them using optimistic concurrency control (OCC). The *NAS* is a *distributed fault tolerant storage (**DFS**)* which stores the persistent database image and the transaction logs. The *controller* is called the **TM Master** which is responsible for system management such as load balancing, detecting and recuperating from failures, and initiating migration. Its role in migration is only limited to notifying the source OTM (N_{src}) and the destination OTM (N_{dst}) to initiate migration. ElasTraS uses an append only storage layer (the Hadoop Distributed File System) where updates to a tenant's data is periodically flushed to create new files on the DFS. The commit log of the OTMs is stored in the DFS. Transaction routing is handled by the **client library** that is linked to every application client; the router transparently migrates the client connections after migration without any changes to the application code. The client library uses a collection of **metadata tables** that store the mapping of a cell to the OTM which is currently serving the cell. The combination of the client library and the metadata tables constitute the query router.

ElasTraS uses a data storage format, called **SSTable**, designed specifically for append-only storage [4]. Conceptually, an SSTable is an immutable structure which stores the rows of a table in sorted order of their keys. Internally, an SSTable is a collection of **blocks** with an index to map blocks to key ranges. This SSTable index is used to read directly from the block containing the requested row and obviates an unnecessary scan through the entire SSTable. An OTM caches the contents of the SSTables. Due to the append only nature of storage, updates are maintained as a separate main memory buffer which is periodically flushed to the DFS as new SSTables; a flush of the update buffers is asynchronous and does not block new updates. The read-cache caches blocks from SSTables as they are accessed by the transactions; a least recently used policy is used for evicting blocks to accommodate new blocks.

Creating a database snapshot. In the first step of migration, N_{src} creates a snapshot of the tenant's database. Albatross does not require a transactionally consistent snapshot of the database cache. N_{src}'s database snapshot is a list of identifiers for the immutable SSTable blocks that are cached. This list of block ids is obtained by scanning the read cache using a read lock. It is passed to N_{dst} which then reads the blocks directly from the DFS and populates its cache. This results in minimal work at N_{src} and delegates all the work of warming up the cache to N_{dst}. The logic for this approach is that during migration, N_{dst} is expected to have less load compared to N_{src}. Therefore, transactions at N_{src} observe minimal impact during snapshot creation and copying. After N_{dst} has loaded all the blocks into its cache, it notifies N_{src} of the amount of data transferred (Δ_0); both nodes now enter the next iteration. No transaction state is copied in this phase.

Iterative copying phase. In every iteration, changes made to the read cache at N_{src} are copied to N_{dst}. After a first snapshot is created, the data manager of C_{migr} at N_{src} tracks changes to the read cache (both inserts and evictions) and incrementally maintains the list ids for the blocks that were evicted from or loaded to the read cache since the previous iteration which is then copied to N_{dst} in subsequent iterations. Again, only the block ids are passed; N_{dst} populates its cache using the ids and notifies N_{src} the amount of data transferred (Δ_i). This iterative phase continues until the amount of data transferred in successive iterations is approximately the same, i.e. $\Delta_i \approx \Delta_{i-1}$. The logic behind this termination condition is that when $\Delta_i \approx \Delta_{i-1}$, irrespective of the magnitude of Δ_i, little gain is expected from subsequent iterations. Δ_i is small for most cases, except when the working set of the database does not fit into the cache when the cache changes frequently. A maximum bound on the number of iterations ensures termination when Δ_i fluctuates between iterations. The write cache is periodically flushed during the iterative copying phase when its size exceeds a specified threshold. A write cache flush creates a new block whose identifier is passed to N_{dst} which loads the new block into its read cache. After the handover, N_{dst} starts serving C_{migr} with an empty write cache, but the combination of the read and write cache contains the same state of data as in N_{src}.

Copying the transaction state. ElasTraS uses OCC [16] for concurrency control. In OCC, the transaction state consists of the read and write sets of the active transactions and a subset of committed transactions needed to validate new transactions; the read/write sets of active transactions and committed transactions are maintained in separate main-memory structures. Two counters are used to assign transaction numbers and commit sequence numbers. In Albatross, the transaction state is copied only in the final handover phase. Writes of an active transaction are stored with the transaction's state and are copied to N_{dst} during handover, along with the counters maintained by the transaction manager. State of a subset of committed transactions (ones that committed after any one of the current set of active transactions started) are copied to N_{dst} to validate the active transactions at N_{dst}. The small size of transaction states allows efficient serialization. After handover, N_{dst} has the exact same transaction state of C_{migr} as N_{src}, thus allowing it to continue executing the transactions that were active at the start of the handover phase.

Handover phase. The handover phase flushes changes from committed transactions. After the transaction state and the final changes to the read cache have been copied, the atomic handover protocol makes N_{dst} the unique owner of C_{migr} and updates the mapping in the metadata used by the query router. The query router (ElasTraS clients) caches the metadata. After handover, N_{src} rejects any request to C_{migr} which invalidates the system metadata cached at the clients; the clients subsequently read the updated metadata. The metadata tables in ElasTraS are served by one of the live OTMs. The OTM serving the metadata tables participates in the *transfer transaction* of the atomic handover phase. The TM Master can be a participant of the transfer transaction so that it is aware of the outcome of migration; however, it is not needed for correctness. In our implementation, the TM Master is notified by N_{src} after handover completes. Clients that have open connections with C_{migr} at N_{src} are notified directly about the new address of N_{dst}. This prevents an additional network round-trip to read the updated metadata mappings. For a transaction accessing C_{migr} during the atomic handover phase, ElasTraS client library transparently retries the operation; once the handover completes, this retried operation is routed to N_{dst}. Since an OTM's commit log is stored in the DFS, it is not migrated. C_{migr}'s transaction log at N_{src} is garbage collected once the transactions active at the start of the handover phase have completed at N_{dst}, though the entries for transactions that committed at N_{src} can be purged after handover completes.

6. EXPERIMENTAL EVALUATION

We now evaluate our prototype implementation of **Albatross** using a variety of workloads. We measure migration cost using four cost measures: tenant unavailability window, number of failed requests (aborted transactions or failed operations), impact on transaction latency (or response time), and additional data transfer during migration. We compare performance with the **stop and mi-**

grate (S&M), a representative off-the-shelf technique, implemented in ElasTraS. In stop and migrate, a long unavailability window results from flushing cached updates from committed transactions. An optimization, called **flush and migrate (F&M)**, performs a flush while continuing to serve transactions, followed by the final stop and migrate step.

6.1 Experimental Setup

6.1.1 *Cluster Configuration*

Experiments were performed on a six node cluster, each with 4 GB memory, a quad core processor, and a 200 GB disk. The distributed fault-tolerant storage and the OTMs are co-located in the cluster of five worker nodes. The TM master (controller) and the clients generating the workloads were executed on a separate node. Each OTM was serving 10 tenants on average. When an operation fails due to tenant unavailability (due to migration or otherwise), the ElasTraS client library transparently retries these operations until the tenant becomes available again and completes the request. We set the maximum number of iterations in Albatross to 10; Albatross converged within $3-7$ iterations in our experiments.

In all experiments, except the one presented in Appendix B.1.7, we evaluate migration cost when both N_{src} and N_{dst} were lightly loaded, so that the actual overhead of migration can be measured. The load on a node is measured using the amount of resources (for instance CPU cycles, disk I/O bandwidth, or network bandwidth) being utilized at the node. When resource utilization is less than 25%, it is referred to as lightly loaded, utilization between $25-70\%$ is referred to as moderately loaded, and utilization above 70% is called overloaded. We only consider CPU utilization.

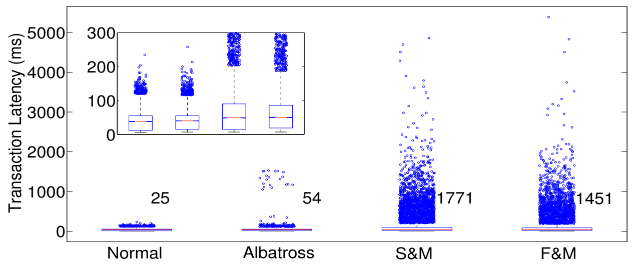
6.1.2 *Benchmarks*

We evaluate migration cost using two OLTP benchmarks: the Yahoo! cloud serving benchmark (YCSB) [7] and the TPC-C benchmark [20]. YCSB [7] is a recently proposed benchmark to evaluate systems that drive web applications. The initial benchmark was designed for Key-Value stores and hence did not support transactions. We extended the benchmark by adding support for multi-step transactions that access multiple tables belonging to a single tenant; we chose a tenant schema with three tables where each table has ten columns of type `VARCHAR` and 200 byte data per column. The workload comprises a set of multi-step transactions which are parameterized by the number of operations, percentage of reads and updates, and the distributions (uniform, Zipfian, and hotspot distributions) used to select the data items accessed by the transaction. In addition, we vary the transaction loads, database sizes, and cache sizes. We ran multiple instances of the benchmark, one for each tenant. Due to space constraints, some YCSB experiments and all TPC-C experiments are provided in Appendix B.

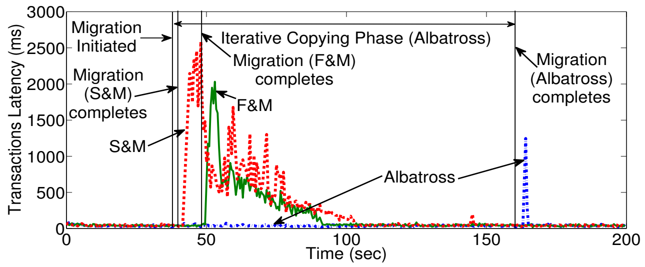
6.2 Methodology

We vary different YCSB parameters to cover a wide spectrum of workloads. These parameters include the percentage of read operations in a transaction (default is 80%), number of operations in a transaction (default is 10), size of the tenant database (default is 1 GB), load on the tenant (default is 50 transactions per second (TPS)), cache size (default is 250 MB), and the distribution from which the keys accessed are selected (default is a hotspot distribution[3]). For Zipfian distribution, the co-efficient is set to 1.0. In an experiment, we vary one of these parameters while using the default

[3]In a Hotspot distribution, $x\%$ operations access $y\%$ data items–default is 80% operations accessing 20% data items.



(a) Transaction latency distribution (box and whisker plot). Inset shows the same graph but with a limited value of the y-axis to show the box corresponding to the 25th and 75th percentiles.



(b) Transaction latency observed for different migration techniques. There are 3 different series and each correspond to an execution using one of the discussed migration techniques. All series are aligned at the time at which migration was initiated (about 38 seconds). Other vertical lines denote time instances when migration completed.

Figure 3: Impact of migration on transaction latency.

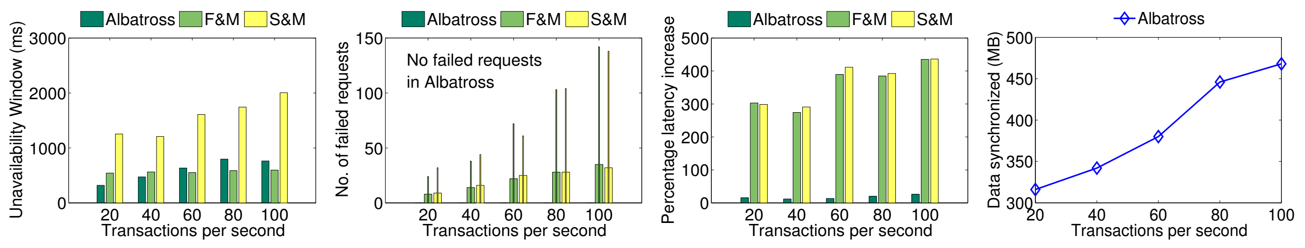
values for the rest of the parameters. In every experiment, we execute about $12,000$ transactions (about 240 seconds at 50 TPS) to warm up the cache, after which migration is initiated. Clients continue to issue transactions while migration is in progress. We only report the latency for committed transactions; latency of aborted transactions is ignored.

6.3 Evaluation

In the first experiment, we analyze the impact of migration on transaction latency using the default workload parameters described above. We ran a workload of $10,000$ transactions, after warming up the cache with another workload of $10,000$ transactions; Figure 3(a) plots the distribution of latency (or response time) of each individual transaction as a box and whisker plot. The four series correspond to the observed transaction latency of an experiment when migration was not initiated (Normal) and that observed when migration was initiated using each of the three different techniques. The inset shows the same plot, but with a restricted range of the y-axis. The box in each series encloses the 25th and 75th percentile of the distribution with the median shown as a horizontal line within each box. The whiskers (the dashed line extending beyond the box) extend to the most extreme data points not considered outliers, and outliers are plotted individually as circles (in blue).[4] The number beside each series denotes the number of outlier data points that lie beyond the whiskers.

As is evident from Figure 3(a), when migrating a tenant using Albatross, the transaction latencies are almost similar to that in the experiment without migration. A cluster of data points with latency about $1000-1500$ ms correspond to transactions that were active

[4]The whiskers denote the sampled minimum and sampled maximum (`http://en.wikipedia.org/wiki/Sample_minimum`).



(a) Tenant unavailability. (b) Failed requests. (c) Transaction latency increase. (d) Data transfer overhead.

Figure 4: Evaluating migration cost using YCSB by varying the transaction load on the tenant. For failed requests (4(b)), the wider bars represent aborted transactions and narrower bars represent failed operations.

during the handover phase which were stalled during the handover and resumed at N_{dst}. On the other hand, both S&M and F&M result in a high impact on transaction latency with about 1500 or more transactions having a latency higher than that observed during normal operation. The high impact on latency for S&M and F&M is due to cache misses at N_{dst} and contention for the NAS. Since all transactions active at the start of migration are aborted in F&M and S&M, they do not contribute to the increase in latency.

The low impact of Albatross on transaction latency is further strengthened by Figure 3(b) which plots the average latency observed by the tenants as time progresses; latencies were averaged in disjoint 500 ms windows. The different series correspond to the different migration techniques and are aligned based on the migration start time (about 38 seconds). Different techniques complete migration at different time instances as is shown by the vertical lines; S&M completes at about 40 seconds, F&M completes at around 45 seconds, while Albatross completes at around 160 seconds. The iterative phase for Albatross is also marked in the figure. As is evident, both F&M and S&M result in an increase in latency immediately after migration completes, with the latency gradually decreasing as the cache at N_{dst} warms up. On the other hand, even though Albatross takes longer to finish, it has negligible impact on latency while migration is in progress. This is because in Albatross, most of the heavy lifting for copying the state is done by N_{dst}, thus having minimal impact on the transactions executing at N_{src}. A small spike in latency is observed for Albatross immediately after migration completes which corresponds to transactions active during the final handover phase. The low impact on latency ensures that there is also a low impact on transaction throughput (see Figure 5 in the Appendix).

Therefore, for all the techniques, an impact on transaction latency (and hence throughput) is observed only in a time window immediately after migration completes. Hence, for brevity in reporting the impact on latency, we report the percentage increase in transaction latency for C_{migr} in a time window immediately after migration, with the base value being the average transaction latency observed before migration. We select 30 seconds as a representative time window based on the behavior of latency in Figure 3(b) where N_{dst} is warmed up within about $30 - 40$ seconds after the completion of migration. We also measured the percentage increase in latency in the period from start of migration to 30 seconds beyond completion of the respective migration techniques. Since Albatross takes much longer to complete compared to the other techniques and has minimal impact on latency during migration, this measure favors Albatross and unfairly reports a lower increase for Albatross. Therefore, we consider the 30 second window after migration such that all techniques can be evenly evaluated.

Figure 4 plots the migration cost as a function of the load, expressed as transactions per second (TPS), on C_{migr}. As the load on the tenant increases (20 TPS to 100 TPS), the amount of un-flushed changes in the write cache also increases. Hence the unavailability window of S&M increases with load (see Figure 4(a)). But since both Albatross and F&M flush the cache (at least once) before the final phase, they are not heavily impacted by load. The unavailability window of Albatross increases slightly since at higher load more transaction state must be copied during the final handover phase. Similarly, a higher load implies more transactions are active at the initiation of migration which are aborted in F&M and S&M, resulting in a large number of failed requests (see Figure 4(b) where the wider bars represent transactions aborted and the narrower bars represent failed operations). Albatross does not result in any failed requests since it copies transaction state and allows transactions to resume at N_{dst}. Both F&M and S&M incur a high penalty on transaction latency. The impact increases with load since more read operations incur a cache miss, resulting in higher contention for accessing the NAS (see Figure 4(c)). Albatross results in only $5 - 15\%$ transaction latency increase (over $80 - 100$ ms average latency) in the 30 second window after migration, while both F&M and S&M result in $300-400\%$ latency increase. Finally, Figure 4(d) plots the amount of data synchronized as a function of load. In spite of the increase in data transmission, we note that this does not adversely affect performance when using Albatross.

We also ran experiments varying a number of other parameters of the YCSB workload as well as with TPC-C. Most experiments follow a trend similar to that observed here, except for the case when the working set of a tenant does not fit in the cache. In that case, Albatross results in a longer unavailability window compared to S&M and F&M since more state must be synchronized in the final handover phase. This is because the state of the cache changes frequently during migration. Albatross, however, continues to have no failed requests and low impact on transaction latency. The low cost incurred by Albatross also makes it effective for live database migration allowing for lightweight elasticity as a first class notion in databases. A detailed analysis of the experiments is provided in Appendix B. In summary, Albatross's migration cost is considerably lower compared to S&M or F&M. Albatross has a small unavailability window, zero failed requests, and less than 15% increase in transaction latency immediately after migration.

7. RELATED WORK

Recently, much research has focussed on cloud databases, databases in a virtualized environment, and multitenancy models. However, little research has focused on live database migration for elastic load balancing. In [12], we propose Zephyr, a live database migration technique for shared nothing database architectures. In a shared nothing architecture, the disks are locally attached to every node. Hence, the persistent image must also be migrated. Zephyr, therefore, focusses on minimizing the service interruption and uses a synchronized *dual mode* where the source and destination nodes are both executing transactions. On the other hand, Albatross is developed for shared storage architectures where the persistent image is not migrated. Albatross not only minimizes the service interruption but also minimize the impact on transaction latency by copying the database cache during migration.

Even though little work has focused on live migration of databases, a number of techniques have been proposed in the virtualization literature to deal with the problem of live migration of virtual machines (VM) [5, 17]. The technique proposed by Clark et al. [5] is conceptually similar to Albatross. The major difference is that Clark et al. use VM level memory page copying and OS and networking primitives to transfer live processes and network connections. On the other hand, Albatross leverages the semantics of a DBMS and its internal structures to migrate the cached database pages (analogous to VM pages), state of active transactions (analogous to active processes), and database connections (analogous to network connections). This allows Albatross to be used for live database migration in the shared process multitenancy model where live VM migration cannot be used. Liu et al. [17] propose an improvement over [5] to reduce the downtime and the amount of data synchronized by using a log based copying approach. Our technique tracks changes to the database state, however, we do not use explicit log shipping for synchronization. Bradford et al. [3] propose a technique to consistently migrate a VM across a wide area network. Database migration in other contexts—such as migrating data as the database schema evolves, or between different versions of the database system—has also been studied; Sockut et al. [19] provide a detailed survey of the different approaches. Our focus is migration used for elasticity.

A large body of work also exists in scalable and distributed data management for the cloud [1, 8, 9, 10, 11, 18]. Even though a large number of such systems exist, the focus of the majority of such systems is to scale single large databases to the cloud and the focus is on performance optimization. Kraska et al. [15] propose the use of varying consistency models to minimize the operating cost in a cloud DBMS. On the other hand, our work proposes the use of live database migration as a primitive for elastic load balancing.

8. CONCLUSION

With the growing number of applications being deployed in different cloud platforms, the need for a scalable, fault-tolerant, and elastic multitenant DBMS has also increased. In such large multitenant systems, the ability to seamlessly migrate a tenant's database is an important feature that allows effective load balancing and elasticity to minimize the operating cost and to ensure efficient resource sharing. We presented Albatross, a technique for live database migration in a shared storage architecture that results in minimal performance impacts and minimal disruption in service for the tenant whose database is being migrated. Albatross decouples a cell from the DBMS node *owning* it, and allows the system to routinely use migration as a primitive for elastic load balancing. Our evaluation using YCSB and TPC-C benchmarks shows the effectiveness of Albatross and analyzes the associate trade-offs. In the future, we plan to extend the design by adding an intelligent system control that can model the cost of migration to predict its cost as well the behavior of the entire system.

Acknowledgments

The authors would like to thank Philip Bernstein, Pamela Bhattacharya, Aaron Elmore, and the anonymous reviewers for providing useful feedback. This work is partly funded by NSF grants III 1018637 and CNS 1053594 and NEC Labs America.

9. REFERENCES

[1] P. A. Bernstein et al. Adapting Microsoft SQL Server for Cloud Computing. In *ICDE (to appear)*, 2011.

[2] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. MK Publishers Inc., second edition, 2009.

[3] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE*, pages 169–179, 2007.

[4] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[5] C. Clark et al. Live migration of virtual machines. In *NSDI*, pages 273–286, 2005.

[6] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[7] B. F. Cooper et al. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, pages 143–154, 2010.

[8] C. Curino et al. Relational Cloud: A Database Service for the Cloud. In *CIDR*, pages 235–240, 2011.

[9] S. Das, S. Agarwal, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Technical Report 2010-04, CS, UCSB, 2010.

[10] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *USENIX HotCloud*, 2009.

[11] S. Das, D. Agrawal, and A. El Abbadi. G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. In *ACM SoCC*, pages 163–174, 2010.

[12] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD (to appear)*, 2011.

[13] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60, pages 393–481. Springer-Verlag, 1978.

[14] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In *BTW*, pages 514–521, 2007.

[15] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[17] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC*, pages 101–110, 2009.

[18] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR Perspectives*, 2009.

[19] G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41(3):1–136, 2009.

[20] The Transaction Processing Performance Council. TPC-C benchmark (Version 5.10.1), 2009.

[21] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD*, pages 889–896, 2009.

[22] P. Xiong et al. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE (to appear)*, 2011.

[23] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.

APPENDIX

A. FAULT-TOLERANCE

We now articulate two important properties of Albatross that allow the system to gracefully tolerate failures and characterize its behavior in the presence of failures during migration.

PROPERTY 1. ***Independent Recovery.*** *Except during the execution of the atomic handover protocol, recovery from a failure of N_{src} or N_{dst} can be performed independently.*

At any point of time before atomic handover, N_{src} is the owner of C_{migr}. If N_{src} fails, it recovers without interacting with N_{dst} and continues to be the owner of C_{migr}. Similarly, if N_{dst} fail, it recovers its state. Unless the handover phase was initiated (Phase 3 of the protocol), N_{dst} has no log record about the migration in progress, so it "forgets" the migration and continues normal operation. Similarly, once handover has been successfully completed, N_{dst} becomes the new owner of C_{migr}. A failure of N_{src} at this instant can be recovered independently as N_{src} does not need to recover state of C_{migr}. Similarly, a failure of N_{dst} requires recovery of only its state; N_{dst} can independently recover state of C_{migr} since it had successfully acquired the ownership of C_{migr}.

PROPERTY 2. ***A single failure does not incur additional unavailability.*** *Any unavailability of C_{migr} resulting from the failure of one of N_{src} or N_{dst} during migration is equivalent to unavailability due to a failure during normal operation.*

From an external observer's perspective, N_{src} is the owner of C_{migr} until the atomic handover phase (Phase 3) has successfully completed. Any failure of N_{dst} before Phase 3 does not affect the availability of C_{migr}. A failure of N_{src} during this phase makes N_{src} unavailable, which is equivalent to a failure of N_{src} under normal operation where C_{migr} would also become unavailable. Similarly, after migration is complete, N_{dst} becomes the owner of C_{migr}. Any failure of N_{src} does not affect C_{migr}, and a failure of N_{dst} which makes C_{migr} unavailable is equivalent to the failure of N_{dst} during normal operation. The only complexity arises in the case of a failure in Phase 3 when a coordinated recovery is needed. If N_{src} fails before successful completion of Phase 3, even if N_{src} had locally relinquished ownership of C_{migr}, if the transfer transaction did not complete, N_{dst} cannot start serving C_{migr} in which case it becomes unavailable. This is similar to the blocking behavior in 2PC [13]. However, since the handover transaction did not complete, from an observer's perspective, N_{src} was still the owner of C_{migr}, and hence this unavailability is equivalent to the failure of N_{src} during normal operation. Thus, it is evident, single site failures during migration does not impact availability of C_{migr}.

The ability to safely abort migration at an incomplete state and the single owner philosophy allow independent recovery of the failed node's state even after a failure during migration . This is crucial for effective use of migration for elasticity without unnecessarily making tenants unavailable when a node fails. Furthermore, one of the implications of Property 2 is that in spite of using a 2PC protocol, the handover phase does not block any system resources as a result of a failure, limiting the impact of failure to only the cells being served by the failed node. This is contrary to the case where a coordinator failure in 2PC causes other transactions conflicting with blocked transactions to also block.

During normal operation, the progress of Albatross is guaranteed by the maximum bound on the number of iterations that forces a handover. Since Albatross does not log the progress of migration, the state synchronized at N_{dst} is not persistent. This is because Albatross copies the main memory state of C_{migr} which is lost after a failure, little gain can be achieved by logging the progress at either node. Progress towards migration is therefore not guaranteed in case of failures.

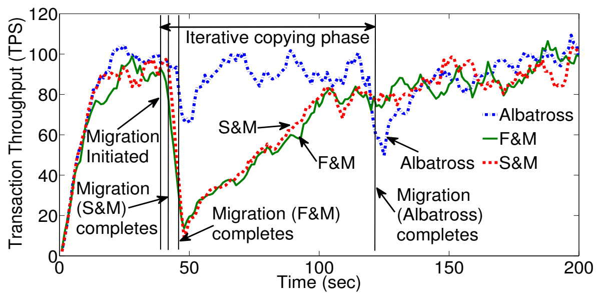


Figure 5: Impact of migration on transaction throughput.

B. DETAILED EXPERIMENTS

We now present a more in-depth evaluation of migration cost using YCSB and TPC-C benchmarks that augments the experiments presented in Section 6. In the figures reporting the number of failed requests, the wider bars represent transactions aborted and the narrower bars represent failed operations.

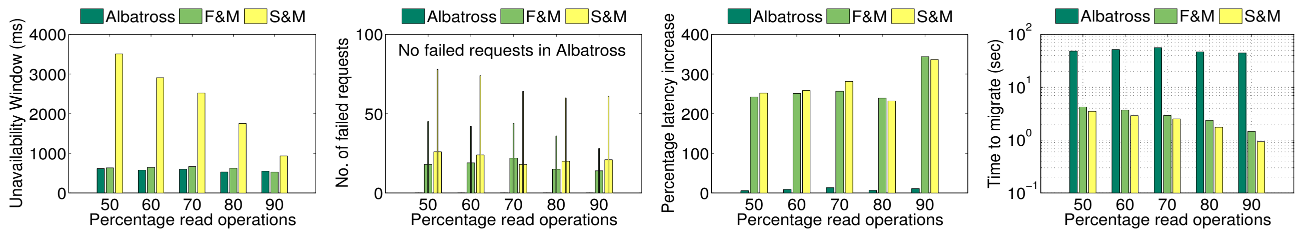
B.1 Yahoo! Cloud Serving Benchmark

B.1.1 Impact on Transaction Throughput

Figure 5 plots the impact of migration on throughput as time progresses (plotted along the x-axis). The y-axis plots the throughput measured for a second long window. The load is generated by four clients threads which issue transactions immediately after the previous transaction completes. The different series correspond to different migration techniques. As is evident from the figure, both S&M and F&M result in a high impact on the client throughput due to increased transaction latency after migration, coupled with throughput reduction during the unavailability window. On the other hand, Albatross results in minor throughput fluctuations, once during the first snapshot creation phase and once during the unavailability window in the handover phase; Albatross results in negligible impact during migration since the list of block identifiers in the cache snapshot is maintained incrementally and N_{dst} performs most of the work done during the synchronization phase.

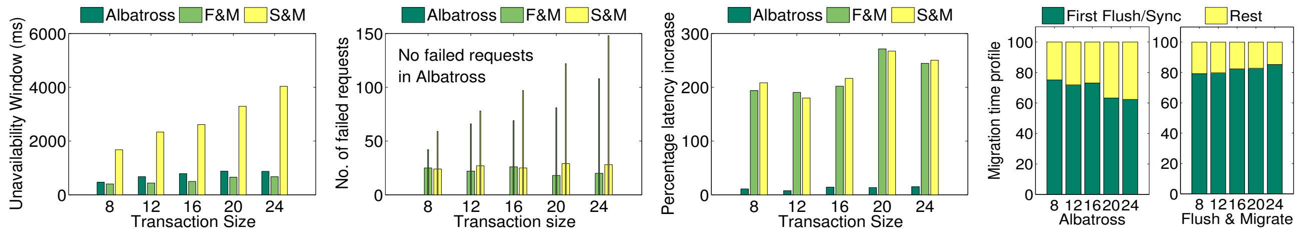
B.1.2 Effect of Read/Write Ratio

We now present results from experiments varying other parameters of YCSB. Figure 6 plots the migration cost measures as a function of the percentage read operations in a transaction; we vary the read percentage from 50 to 90. For an update heavy workload, the write cache has a large amount of un-flushed updates that must be flushed during migration. As a result, S&M incurs a long unavailability window of about $2-4$ seconds, the length of which decreases with a decrease in the percentage of writes (see Figure 6(a)). On the other hand, both F&M and Albatross flush the majority of updates before the final stop phase. Therefore, their unavailability window is unaffected by the distribution of reads and writes. However, since both S&M and F&M do not migrate transaction state, all transactions active at the start of migration are aborted, resulting in a large number of failed requests (see Figure 6(b)). Albatross, on the other hand, does not have any failed requests. As can be seen in Figure 6(c), Albatross results in only $5-15\%$ transaction latency increase, while both F&M and S&M incur a $300-400\%$ increase



(a) Tenant unavailability. (b) Failed requests. (c) Transaction latency increase. (d) Total time, in seconds, to migrate a cell (log scale).

Figure 6: Evaluating migration cost by varying the percentage of read operations for transactions in YCSB.



(a) Tenant unavailability. (b) Failed requests. (c) Transaction latency increase. (d) Percentage time distribution.

Figure 7: Evaluating migration cost by varying the number of operations in a transaction in YCSB.

in transaction latency due to the cost of warming up the cache at the destination. Since Albatross warms up the cache at the destination during the iterative phase, the total time taken by Albatross from the start to finish is much longer compared to that of F&M and S&M; S&M is the fastest followed by F&M (see Figure 6(d)). However, since C_{migr} is still active and serving requests with no impact on transaction latency, this background loading process does not contribute to migration cost from the tenant's perspective. The iterative copying phase transfers about 340 MB data between N_{src} and N_{dst}, which is about 35% greater that the cache size (250 MB). F&M and S&M will also incur network overhead of 250 MB resulting from cache misses at N_{dst} and a fetch from NAS.

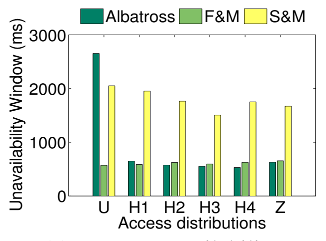
B.1.3 *Effect of Transaction Size*

Figure 7 shows the effect of transaction size on migration cost; we vary the number of operations in a transaction from 8 to 24. As the transaction size increases, so does the number of updates, and hence the amount of un-flushed data in the write cache. Therefore, the unavailability window for S&M increases with increased transaction size (see Figure 7(a)). In this experiment, F&M has a smaller unavailability window compared to Albatross. This is because Albatross must copy the transaction state in the final handover phase, whose size increases with increased transaction size. F&M, on the other hand, aborts all active transactions and hence does not incur that cost. The number of failed requests is also higher for F&M and S&M, since an aborted transaction with more operations result in more work wasted (see Figure 7(b)). The impact on transaction latency also increases with size since larger transactions have more reads (see Figure 7(c)). Figure 7(d) shows a profile of the total migration time. As expected, the majority of the time is spent in the first sync or flush, since it results in the greatest amount of data being transferred or flushed. As the number of operations in a transaction increases, the amount of state copied in the later iterations of Albatross also increases. Therefore, the percentage of time spent on the first iteration of Albatross decreases. On the other hand, since the amount of data to be flushed in F&M increases with transaction size, the time taken for the first flush increases.

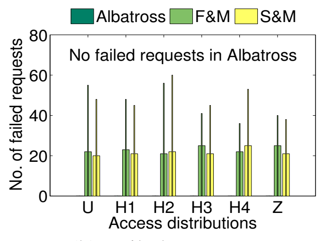
B.1.4 *Effect of Access Distributions*

Figure 8 plots the migration cost as a function of the distributions that determine the data items accessed by a transaction; we experimented with uniform, Zipfian, and four different variants of the hotspot distribution where we vary the size of the hot set and the number of operations accessing the hot set. Since the cache size is set to 25% of the database size, uniform distribution incurs a high percentage of cache misses. As a result, during the iterative copy phase, the database cache changes a lot because of a lot of blocks being evicted and loaded. As a result, every iteration results in a significant amount of data being transferred. Since Albatross checks the size of data transferred in each iteration, this value converges quickly; in this experiment, Albatross converged after 3 iteration. However, the final handover phase has to synchronize a significant amount of data, resulting in a longer unavailability window. Therefore, a high percentage of cache misses results in a longer unavailability window for Albatross. F&M and S&M are, however, not affected since these techniques do not copy the database cache. This effect disappears for skewed workload where as expected, Albatross and F&M have similar unavailability window and S&M has a comparatively longer unavailability window. Albatross does not result in any failed requests, while the number of failed requests in F&M and S&M is not heavily affected by the distribution (see Figure 8(b)). The uniform distribution results in a higher number of cache misses even at N_{src} which offsets the impact of cache misses at N_{dst}. Therefore, the percentage increase in transaction latency for S&M and F&M is lower for the uniform distribution when compared to other access patterns (see Figure 8(c)). Irrespective of the access distribution, Albatross has little impact on latency.

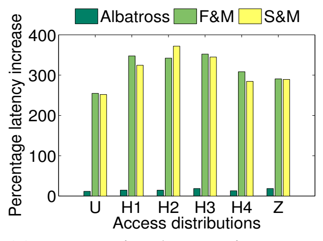
Figure 8(d) plots the amount of data synchronized by Albatross. Following directly from our discussion above, a uniform distribution results in a larger amount of data being synchronized when compared to other distributions. It is however interesting to note the impact of the different hotspot distributions on the data synchronized. For H1 and H3, the size of the hot set is set to 10% of the database, while for H2 and H4, the size of the hot set is set to



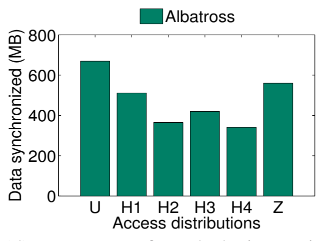
(a) Tenant unavailability.



(b) Failed requests.

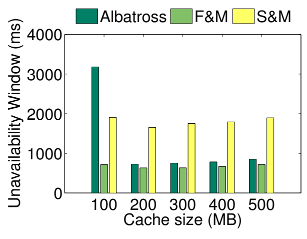


(c) Transaction latency increase.

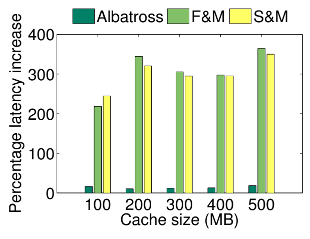


(d) Data transferred during migration.

Figure 8: Evaluating migration cost using YCSB for different data access distributions. U denotes uniform and Z denotes Zipfian. H1–H4 denote hotspot distributions: 90-10, 90-20, 80-10, and 80-20, where x-y denotes $x\%$ operations accessing $y\%$ data items.

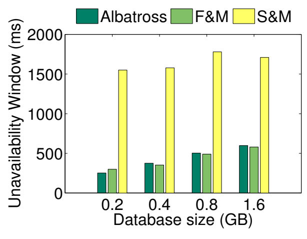


(a) Tenant unavailability.

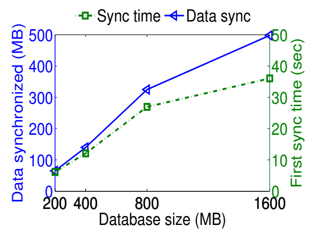


(b) Transaction latency increase.

Figure 9: Evaluating migration cost using YCSB by varying the cache size allocated to each tenant.



(a) Tenant unavailability.



(b) Data synchronized and time taken for first synchronization.

Figure 11: Evaluating migration cost using YCSB by varying the tenant database size.

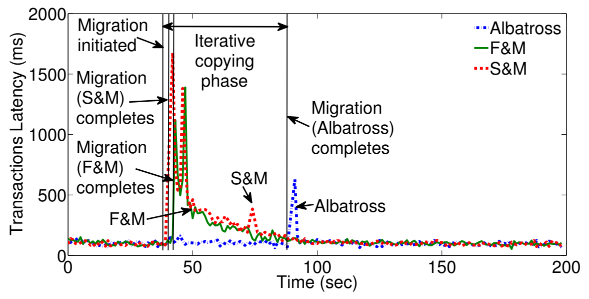


Figure 10: Impact of migration on transaction latency when working set does not fit in the cache.

20%. Since in H1 and H3, a fraction of the cold set is stored in the cache, this state changes more frequently compared to H2 and H4 where the cache is dominated by the hot set and hence its state does not change frequently. As a result, H1 and H3 result in a larger amount of data synchronized. For the Zipfian distribution, the percentage of data items accessed frequently is even smaller than that in the experiments with 10% hot set, which also explains the higher data synchronization overhead.

B.1.5 *Effect of Cache Size*

Figure 9 plots migration cost as a function of the cache size while keeping the database size fixed; the cache size is varied from 100MB to 500MB and the database size is 1GB. Since Albatross copies the database cache during migration, a smaller database cache implies lesser data to synchronize. When the cache size is set to 100MB, the unavailability window of Albatross is greater than that of F&M and S&M (see Figure 9(a)). This behavior is caused by the fact that at 100MB, the cache does not entirely accommodate the hot set of the workload (which is set to 20% of the data items or 200 MB), thus resulting in a high percentage of cache misses. This impact of a high percentage of cache misses on migration cost is similar to that observed for the uniform distribution. However, since the iterations converge quickly, the amount of data synchronized is similar to that observed in other experiments. For cache sizes of 200MB or larger, the hot set fits into the cache, and hence expected behavior is observed. Even though Albatross has a longer unavailability window for a 100MB cache, the number of failed operations and the impact on transaction latency continues to be low. For F&M and S&M, the impact on transaction latency is lower for the 100 MB cache because a large fraction of operations incurred a cache missed even at N_{src} which somewhat offsets the cost due to cache missed at N_{dst} (see Figure 9(b)). Number of failed operations and data synchronized show expected behavior.

Figure 10 plots the impact of migration on latency as time progresses. In this experiment we consider a scenario where the working set of the database does not fit in the cache. The cache size is set to 100 MB when using a hotspot distribution where the hot set is 20% of the database. This experiment confirms the observation that when the working set does not fit in the cache, even though Albatross results in a longer unavailability window, there is minimal impact on transaction latency.

B.1.6 *Effect of Database Size*

Figure 11 plots the migration cost as a function of the database size. Since the persistent image of the database is not migrated, the actual size of the database does not have a big impact on migration cost. We therefore vary the cache size along with the database size such that the cache is set to 25% of the database size. Since the cache is large enough to accommodate the hot set (we use the default hotspot distribution with the hot set as 20% of the database), the migration cost will be lower for a smaller database (with a smaller cache); the cost increases with an increase in the database size (see Figure 11(a)). Similarly, as the size of the database cache increases, the amount of state synchronized and the time taken for the synchronization also increases (see Figure 11(b)).

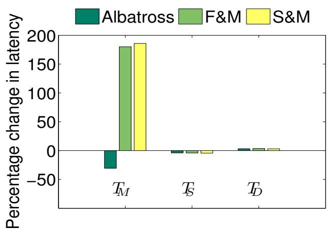
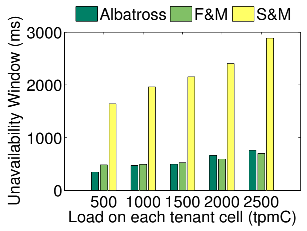
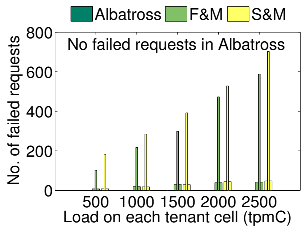


Figure 12: Impact of migrating tenant T_M from a heavily loaded node to a lightly loaded node. T_S and T_D represent a representative tenant at N_{src} and N_{dst} respectively.



(a) Unavailability window.



(b) No. of failed requests.

Figure 13: Evaluating migration cost using TPC-C.

B.1.7 Migration Cost during Overload

We now evaluate the migration cost in a system with high load. Figure 12 shows the impact of migrating a tenant from an overloaded N_{src} to a lightly loaded N_{dst}. In this experiment, the load on each tenant is set to 50 TPS and the number of tenants served by N_{src} is gradually increased to 20 when N_{src} becomes overloaded. As the load on N_{src} increases, all tenants whose database is located at N_{src} experience an increase in transaction latency. At this point, one of the tenants at N_{src} is migrated to N_{dst}. In Figure 12, the y-axis plots the percentage change in transaction latency in the 30 second window after migration; a negative value implies reduction in latency. T_M is the tenant that was migrated, T_S is a tenant at N_{src} and T_D is a tenant at N_{dst}. The latency of T_M's transactions is higher when it is served by an overloaded node. Therefore, when T_M is migrated from an overloaded N_{src} to a lightly loaded N_{dst}, the transaction latency of T_M should decrease. This expected behavior is observed for Albatross, since it has a low migration cost. However, the high cost of F&M and S&M result in an increase in transaction latency even after migrating T_M to a lightly loaded N_{dst}. This further asserts the effectiveness of Albatross for elastic scaling/load balancing when compared to other heavyweight techniques like S&M and F&M. All migration techniques, however, have low overhead on other tenants co-located at N_{src} and N_{dst}. This low overhead is evident from Figure 12, where a small decrease in latency of T_S results from lower aggregate load on N_{src} and a small increase in transaction latency of T_D results from the increased load at N_{dst}.

B.2 TPC-C Benchmark

We now evaluate Albatross using the TPC-C benchmark [20] adapted for a multitenant setting. The goal is to evaluate the performance of Albatross using complex transaction workloads representing real-life business logic and tenant databases with complex schema. The TPC-C benchmark is an industry standard benchmark for evaluating the performance of OLTP systems. The benchmark suite consists of nine tables and five transactions that portray a wholesale supplier. The five transactions represent the business needs and workloads: (i) the NEWORDER transaction which models the placing of a new order; (ii) the PAYMENT transaction which simulates the payment of an order by a customer; (iii) the ORDER-STATUS transaction representing a customer query for checking the status of the customer's last order; (iv) the DELIVERY transaction representing deferred batched processing of orders for delivery; and (v) the STOCKLEVEL transaction which queries for the stock level of some recently sold items. A typical transaction mix consists of approximately 45% NEWORDER transactions, 43% PAYMENT transactions, and 4% each of the remaining three transaction types, representing a good mix of read/write transactions. The system comprises of a number of warehouses which in turn determines the scale of the system. Since more than 90% of transactions have at least one write operation (insert, update, or delete), TPC-C represents a write heavy workload. More details about the benchmark can be found in [20]. Each tenant represents an instance of the TPC-C benchmark and multiple benchmark instances are simultaneously executed in the system.

Figure 13 plots the results from the experiments using the TPC-C benchmark where we varied the load on each of the tenant cells. In both sub figures, the y-axis plots the migration cost measures, while the x-axis plots the load on the system. In these experiments, each tenant database size was about 1 GB and contained four TPC-C warehouses and the cache per tenant is set to 500 MB. We vary the load on each tenant from 500 tpmC (transactions per minute TPC-C) to 2500 tpmC. As the load on each cell increases, the amount of data transferred to synchronize state also increases. As a result, the length of the unavailability window increases with an increase in the load on the tenant (see Figure 13(a)). Furthermore, since the arrival rate of operations is higher at a higher load, an increase in the unavailability window has a greater impact at higher loads resulting in more failed requests. This increase is observed in Figure 13(b). Even using such complex transactional workloads, the performance of Albatross is considerably better than S&M and F&M. The behavior of transaction latency increase and amount of data synchronized is similar to previous set of experiments. Albatross incurred less than 15% increase in transaction latency compared to 300% increase for F&M and S&M, while Albatross synchronized about 700MB data during migration for a 500MB cache.

B.3 Discussion

In a multitenant system, due to aggressive consolidation and resource sharing between tenants, a low cost migration technique is important. It enables the system to guarantee that if the need arises, tenants can be migrated to improve performance while ensuring that their SLAs are met. For instance, as is evident from the experiment reported in Figure 12, even though the load on every tenant at N_{src} is only 50 TPS, as the number of tenants at N_{src} increases, it causes an overload. A low cost migration technique can help alleviate such scenarios commonly encountered in multitenant systems.

A lightweight live migration technique is also helpful in a scenario when one tenant faces a load spike. It allows the system to either migrate other lightly loaded tenants from the overloaded node to another node or migrate the overloaded tenant to a node with more resources. The first option minimizes the total load on the source node while isolating other tenants from being impacted by the heavily loaded tenant. Moreover, as observed in our experiments, migrating lightly loaded tenants is less expensive compared to migrating a tenant with a high load. On the other hand, the second option requires migrating only one tenant, though at a higher migration cost. The option chosen will depend on the workload and tenant characteristics. An intelligent system controller can make prudent use of live migration for such elastic load balancing.