Madeus: Database Live Migration Middleware under Heavy Workloads for Cloud Environment

Takeshi Mishima
NTT Software Innovation Center
mishima.takeshi@lab.ntt.co.jp

Yasuhiro Fujiwara
NTT Software Innovation Center
fujiwara.yasuhiro@lab.ntt.co.jp

ABSTRACT

Database-as-a-service has been gaining popularity in cloud computing because multitenant databases can reduce costs by sharing off-the-shelf resources. However, due to heavy workloads, resource sharing often causes a hot spot; one node is overloaded even while others are not. Unfortunately, a hot spot can lead to violation of service level agreements and destroy customer satisfaction. To efficiently address the hot spot problem, we propose a middleware approach called Madeus that conducts database live migration. To make efficient database live migration possible, we also introduce the lazy snapshot isolation rule (LSIR) that enables concurrently propagating syncsets, which are the datasets needed to synchronize slave with master databases. Madeus provides efficient database live migration by implementing the LSIR under snapshot isolation. Unlike current approaches, Madeus is pure middleware that is transparent to the database management system and based on commodity hardware and software. To demonstrate the superiority of our approach over current approaches, we experimentally evaluated Madeus by using PostgreSQL with the TPC-W benchmark. The results indicate that Madeus achieves more efficient live migration than three other types of middleware approaches, especially under heavy workloads; therefore, it can effectively resolve hot spots.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Theory, Design, Experimentation, Performance

Keywords

cloud, middleware, database live migration, hot spot

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.272458.

1. INTRODUCTION

Recently, there has been a great deal of enthusiasm in cloud computing since it can reduce costs by using the multitenant model [1]. In this model, multiple tenants are consolidated on a single node; hardware and operation costs can be reduced by sharing nodes and administrators. Additionally, cloud computing can also reduce the engineering, testing, and maintenance costs by using off-the-shelf hardware and software. To increase resource utilization, cloud providers consolidate as many tenants on a single node as possible. However, as a node may have many tenants or a tenant may have an unusually heavy workload, the node may become a hot spot; one node is overloaded even while others are not. The hot spot can lead to violation of service level agreements (SLAs) and destroy customer satisfaction with the service; users expect to obtain responses in 2 s or less in Internet services [2, 3, 4]. Unfortunately, it is difficult to predetermine the ideal number of tenants in a node. To make matters worse, workloads are erratic and unpredictable. To address the hot spot problem, cloud providers adopt live migration, which involves transferring tenants among nodes to better balance loads as well as determining whether a heavy or light tenant should be transferred.

Cloud database services are offered today in pre-packaged VM-instances (called database instance) [5, 6, 7]. However, VMs incur significant throughput deterioration in normal operation [30] which will also impact the performance of the database applications running on them. One of the reasons is that each database instance uses its own transaction log file leading to random access of these files. The shared process model proposed by Curino et al. [22] can be a solution to this problem. With such a model, multiple databases (tenants) share the same DBMS process and hence share a transaction log file [22]. Since this model avoids random access of transaction logs, it can significantly reduce overhead in normal operation. Therefore, in our work, we assume this (Database-as-a-Service) (DBaaS) model.

Recent studies have proposed two shared process models for database live migration [23, 25]. In these models, a tenant on a node called the source is migrated to another node called the destination, while the migrated database is called the master and the created one the slave. Das et al. proposed to start a slave DBMS process with a warm cache on a shared-disk architecture [23]. Elmore et al. proposed to leverage index information in a shared-nothing architecture [25]. Unfortunately, both models need to be implemented within the core of the DBMSs. This means rewriting of existing DBMSs is required. Since the source code

of a DBMS is massive and complex, such modification is impractical in a DBaaS that uses off-the-shelf software.

For efficient live migration at low cost, we focus on a middleware-based approach; we use off-the-shelf hardware and software with no modification. Barker et al. proposed a middleware approach for database live migration [16]. The advantage of their approach is that it does not require any VM or DBMS modification. Unlike our approach, their approach uses transaction logs to synchronize the slave with the master. Unfortunately, since the transaction log format depends on the version and type of DBMS [34], their approach requires dedicated middleware for each DBMS pair (master and slave databases). Furthermore, their approach cannot use the shared process model since it is difficult to unearth data of a particular database from transaction logs. Moreover, their approach relies strongly on the commercial backup tool (i.e., Percona XtraBackup [8]).

Live migration needs replication between the master and slave. Replication approaches are classified into eager and lazy [27]. Eager replication keeps all replicas (databases) exactly synchronized at all nodes by updating all the replicas as part of one atomic transaction. In lazy replication, updates of the master are asynchronously propagated to the slave after the update transaction commits. In live migration, the slave is empty at first and before replication, we must create a database by using snapshot. During this process, the system must continue to execute customers' transactions; this leads us to adopt the asynchronous approach of lazy replication. However, it is difficult to achieve efficient live migration by naively extending existing lazy replication [24, 36, 37]. If syncsets[1] are serially propagated, migration time can be long since each syncset is processed individually [36, 37]. Moreover, even if write operations are propagated concurrently, commit operations are not [24]. This limitation decreases concurrency and prevents group commit (If multiple commit operations are propagated concurrently, the DBMS induces only one I/O access for all the commit operations) and its advantages. This results in inefficient database live migration.

In this paper, we propose Madeus, a pure middleware approach that provides efficient database live migration. We assume the shared process model [22]. Since this model does not use any VMs, it can significantly reduce throughput degradation. We also assume snapshot isolation (SI) as the transaction isolation level since SI is a strong isolation level and results in higher performance [17]. Under SI, a transaction T_i reads data from the committed state of the database, namely a *snapshot*, and writes its own snapshot. Since a transaction reads data items from a snapshot, the transaction never refers to the results of other transactions. That is, transaction T_i detects all the changes made by other transactions committed before transaction T_i starts. Transaction T_i does not detect any changes made by other transactions committed after transaction T_i started. In SI, a read operation never blocks write operations, and vice versa.

Madeus is transparent to a DBMS based on commodity hardware and software without modification and independent of any dedicated tool. Our key idea is that Madeus propagates commit operations concurrently as well as the first read operation and write operations to the slave. Madeus is especially designed to handle database live migration under heavy workloads. Our contributions are as follows:

- We introduce a lazy snapshot isolation rule (LSIR) for efficient migration and the minimum set of queries in the LSIR to make the slave consistent with the master in database live migration.
- We propose Madeus, a pure middleware approach that provides efficient database live migration. Madeus concurrently propagates not only the first read operations and write operations but also commit operations by using the LSIR; Madeus benefits from group commit unlike the current approaches.
- We conducted experiments with the TPC-W benchmark. The results showed that Madeus is more efficient than three other types of middleware approaches, especially under heavy workloads; the first one propagates operations serially, the second one propagates minimum operations serially [36, 37], and the third one propagates write operations concurrently but commit operations serially by obeying the previously proposed rules [24]. Moreover, we can give an answer for the question of which tenant, heavy or light, we should migrate to address a hot spot.

The remainder of this paper is organized as follows. In Section 2 we give the background of our work and theoretically discuss a database model and introduce the LSIR in Section 3. In Section 4, we propose Madeus, a lazy replication based middleware approach. In Section 5, we explain the experimental results and discuss related work in Section 6. We conclude the paper in Section 7.

2. PRELIMINARY

Section 2.1 explains the database model in our approach, Section 2.2 defines six types of dependencies among transactions, and Section 2.3 discusses the feature of SI.

2.1 Database Model

A *database* consists of a set of data items. Each data item has a *value*. The values at any one time comprise the *state* of the database. We denote data items by lower-case letters, typically x, y, or z. A *DBMS* is a collection of hardware and software modules that support commands to access the database; these commands are called *operations*. A *transaction* T_i is a sequence of read, write, and end (commit or abort) operations. The subscript i identifies the i-th *version* of transactions and distinguishes it from other transactions. Thus, x_i denotes the data item x written by transaction T_i, $w_{i,p}(x_i)$ denotes the p-th write operation by transaction T_i on data item x_i, and $r_{i,q}(x_j)$ represents the q-th read operation of item x_j. The terms c_i and a_i denote T_i's commit and abort operations, respectively.

We say transactions T_i and T_j ($i < j$) are *serial* if transaction T_i commits before transaction T_j starts. If transaction T_i starts but does not commit until after transaction T_j starts, then transactions T_i and T_j are *concurrent*. When a set of transactions is executed concurrently, their operations may be interleaved. We define a *schedule* as the order in which the DBMS scheduler executes operations in the transactions. A *history* indicates the order in which the operations in the transactions were actually executed. Note that a schedule is a plan for the future and is not necessarily the same as the actual history.

[1] Queries to synchronize the slave with the master.

2.2 Dependency

In this paper, we say transactions T_i and T_j have a *dependency* if a slave database executes transactions T_i and T_j in a different order from a master database and the two databases can have different outputs. To synchronize the master and slave, we must replay all dependencies in the slave. We use the phrase, data item x_j is an *immediate successor* of data item $x_i (i < j)$, to indicate that data item x_i is read or written, data item x_j is written, and no data item is written between data items x_i and x_j. We say that operations o_i and o_j have dependency if at least one of the operations is write since changes in execution order of the same operations can yield different results. We define three types of *transactional dependencies* as follows:

Definition 1 (TRANSACTIONAL DEPENDENCIES). *The following three types of dependencies between two operations are defined according to read or write operations:*

- *Operations o_i and o_j have a wr-dependency if operation o_i writes data item x_i by transaction T_i for data item x and operation o_j later reads this version x_i.*
- *When operation o_i reads data item x written by transaction T_k and operation o_j writes the immediate successor version x_j of data item x, operations o_i and o_j have an rw-dependency.*
- *There is a ww-dependency between operations o_i and o_j when operation o_i writes data item x of version x_i and the immediate successor version x_j is written by operation o_j.*

It is clear that two read operations have no impact on the results of the operations. Therefore, we eliminate rr-dependency from further discussion. We also categorize the two remaining types of dependencies in terms of intra or inter transaction. If operations o_i and o_j belong to the same transaction, they have *intra-transaction dependency*. If operations o_i and o_j belong to different transactions, they have *inter-transaction dependency*.

Since the two categorizations are orthogonal, we have the following six types of dependencies; *intra-wr-dependency*, *inter-wr-dependency*, *intra-rw-dependency*, *inter-rw-dependency*, *intra-ww-dependency*, and *inter-ww-dependency*.

2.3 How To Resolve Inter-WW-Dependency

With regard to inter-ww-dependency, SI follows the *first-updater-wins* rule [26] as follows[2]: If transaction T_i updates data item x, it sets a write lock for data item x. If transaction T_j subsequently attempts to update data item x while transaction T_i is still active, transaction T_j will be prevented by the lock on data item x from making further progress. If transaction T_i then commits, transaction T_j will abort; transaction T_j will continue only if transaction T_i drops its lock for data item x by aborting. On the other hand, if transaction T_i updates data item x but then commits before transaction T_j attempts to update data item x, there will be no delay due to locking. However, transaction T_j will abort immediately if it attempts to update data item x; the abort does not wait until transaction T_j attempts to commit.

[2]When two transactions attempt to modify the same data item, only the first updater modifies successfully and the other aborts. The first-updater-wins rule is used in standard DBMSs such as Oracle, SQL Server, and PostgreSQL.

Table 1: Definition of main symbols

Symbol	Definition
T_i	i-th transaction
T^m	transaction in a master database
T^s	transaction in a slave database
x_i	data item x written by transaction T_i
$r_{i,p}(x_j)$	p-th read operation of data item x_j in T_i
$w_{i,p}(x_i)$	p-th write operation of data item x_i in T_i
c_i	commit operation in transaction T_i
a_i	abort operation in transaction T_i
o_i	read or write operation in transaction T_i
R^m	master database
R^s	slave database
H^m	history of database R^m
H^s	history of database R^s
S^s	schedule of database R^s
$\mathcal{T}^m$	set of transactions T^m
$\mathcal{T}^s$	set of transactions T^s

3. EFFICIENT LAZY REPLICATION

Our goal is to efficiently conduct live migration under heavy workloads. To this end, we introduce the minimum sets of queries to make the slave consistent with the master as well as efficient scheduling rules.

If two operations in master and slave databases do not have any dependencies, these databases have the same outputs regardless of the execution order. In other words, if the two operations have a dependency, the master and slave databases can have different outputs. This section describes the minimum query set that makes the slave consistent with the master to achieve efficient database live migration. Let R^m and R^s be a master and slave database in a lazy replicated database system, respectively. If all dependencies of database R^m are also mirrored on database R^s, database R^s achieves consistency with database R^m. We assume that databases R^m and R^s are consistent at first. In Section 3.1, we illustrate the snapshot creation rule that we assume. Section 3.2 discusses unnecessary/necessary dependencies with regard to making the master and slave database consistent. In Section 3.3, we describe our approach to obtaining the necessary dependencies in the middleware level for live migration. We finally introduce the LSIR, which makes the slave consistent with the master in live migration, in Section 3.4. Table 1 lists the main symbols and their definitions.

3.1 Snapshot Creation

We assume that we have no blind write operation [9, 29]; it modifies a data item without reading the data in advance. In a previous study [24], the snapshot was created by start operation explicitly. In fact, in practical DBMSs, such as Oracle, SQL Server, and PostgreSQL, the snapshot of transaction is implicitly created just before the first operation is executed. We assume this realistic case. Considering we have no blind write, the first operation must be a read operation. Therefore, the snapshot of transaction T_i is created just before the first read operation $r_{i,1}(x_p)$ is executed.

3.2 Necessary Dependencies

In lazy replication, if the slave database replays all the same dependencies as the master database, the same output can be obtained; the slave is consistent with the master. However, this naive approach is obviously inefficient in database live migration, especially under heavy workloads. To achieve efficient live migration without sacrificing the consistency between master and slave databases,

we identify those dependencies that the slave database does not need to replay. We can reduce replay overhead in the slave database by discarding unnecessary dependencies in the master database.

As described in Section 2.2, we have six types of dependencies between operations; intra-wr-dependency, inter-wr-dependency, intra-rw-dependency, inter-rw-dependency, intra-ww-dependency, and inter-ww-dependency. Among these dependencies, inter-ww-dependency and intra-wr-dependency are unnecessary for the slave database to replay in terms of database live migration under SI as follows.

Lemma 1 (UNNECESSARY INTER-WW-DEPENDENCY). *In lazy replication, we need not replay inter-ww-dependency to make the slave consistent with the master under SI.*

Due to space limitation, all proofs of lemmas and theorems are located in Appendixes.

Lemma 2 (UNNECESSARY INTRA-WR-DEPENDENCY). *In making the slave consistent, replay of the intra-wr-dependencies is not necessary in lazy replication under SI.*

From Lemmas 1 and 2, it is clear that the necessary dependencies are the remaining four dependencies as follows.

Lemma 3 (NECESSARY DEPENDENCIES). *The slave database must replay inter-wr-dependency, inter-rw-dependency, intra-rw-dependency, and intra-ww-dependency to make the slave consistent with the master under SI.*

3.3 Dependency-Query Relationships

Section 3.2 revealed the four types of dependencies necessary for database live migration. In this section, we describe the properties that are used to replay the dependencies between transactions in the middleware level.

In our approach, the first read operation creates a snapshot and the other read operations read data items from the snapshot. Furthermore, write operations modify the snapshot. Therefore, the first read operations cause wr-dependency and rw-dependency.

Lemma 4 (INTER-WR-DEPENDENCY). *Transactions T_i and T_j have inter-wr-dependency if T_i is an update transaction and $c_i < r_{j,1}(x_i)$.*

Lemma 5 (INTER/INTRA-RW-DEPENDENCY). *If T_j is an update transaction and $r_{j,1}(x_k) < c_i$, transactions T_i and T_j have inter-rw-dependency or intra-rw-dependency.*

Lemma 6 (INTRA-WW-DEPENDENCY). *If T_i is an update transaction and $w_{i,p}(x_i) < w_{i,p+1}(x_i)$, transaction T_i has intra-ww-dependency.*

3.4 Lazy Snapshot Isolation Rule

This section introduces the **LSIR**, our core innovation for efficient database live migration without loss of consistency. As described in Section 3.2, the master database can have unnecessary transactions to be executed in the slave database in terms of database live migration under SI. Therefore, we identify unnecessary transactions and discard them to reduce overhead in synchronizing the slave with the master. To achieve efficient live migration at the middleware level, the LSIR is based on the properties introduced in Section 3.3, as they can replay the dependencies between transactions at the middleware level. The LSIR defines the scheduling rules for the minimum set of necessary operations from the operations of the master. Before detailing the LSIR, we introduce a mapping function that is used in the LSIR. The mapping function detects transactions to be propagated to the slave. This function outputs syncsets in the slave database from the transactions of the master database and is defined as follows.

Definition 2 (MAPPING FUNCTION). *Let $\mathcal{T}^m$ be a set of master transactions, where each master transaction T_i^m has operations $o_i^m \in \{r_{i,1}, r_{i,2}, ..., r_{i,p}, w_{i,1}, w_{i,2}, ..., w_{i,q}, c_i, a_i\}$. Let $\mathcal{T}^s$ be syncsets in the slave database. The mapping function $\mathcal{F}$ outputs syncsets $\mathcal{T}^s$ in the slave database from master transactions $\mathcal{T}^m$ as follows:*

(1) If master transaction T_i^m is a read-only or abort transaction, the mapping function $\mathcal{F}$ outputs an empty set, i.e., $\mathcal{F}(T_i^m) = \emptyset$.

(2) If master transaction T_i^m is a committed update transaction, the mapping function $\mathcal{F}$ maps the first read operation $r_{i,1}^m$ of the master transaction T_i^m to the first read operation $r_{i,1}^s$ of the syncsets T_i^s and discards other read operations $r_{i,2}^m, ..., r_{i,p}^m$ of the master transaction T_i^m.

(3) If master transaction T_i^m is a committed update transaction, the mapping function $\mathcal{F}$ maps the write operations $w_{i,1}^m, w_{i,2}^m, ..., w_{i,q}^m, c_i^m$ of the master transaction T_i^m to the write operations $w_{i,1}^s, w_{i,2}^s, ..., w_{i,q}^s, c_i^s$ of the syncsets T_i^s.

Intuitively, from Definition 2, the mapping function obtains syncsets as follows. It discards read-only and aborted transactions. The first read operation of a committed update transaction is preserved because the first read operation creates a snapshot of the database in SI and causes intra/inter-rw-dependency. Because the others read only the state of a data item of the snapshot, they do not cause dependencies; thus, the other read operations are discarded. It preserves all write and commit operations because they change the database. Consequently, the mapping function yields $o_i \in \{r_{i,1}, w_{i,1}, w_{i,2}, ..., w_{i,q}, c_i\}$ in the slave database for an operation in the master database such that $o_i^m \in \{r_{i,1}, r_{i,2}, ..., r_{i,p}, w_{i,1}, w_{i,2}, ..., w_{i,q}, c_i, a_i\}$. Based on the mapping function, we introduce the LSIR, which makes the slave consistent with the master. The LSIR is defined as follows.

Definition 3 (LSIR). *Let S^s be a schedule over a set of syncsets $\mathcal{T}^s$ in the slave database R^s. The LSIR for schedule S^s is defined as follows.*

(1) If T_i^m is a committed update transaction in the master database, we control the slave database by the rule such that
(1-a) $c_i^m < r_{j,1}^m \in H^m \Rightarrow c_i^s < r_{j,1}^s \in S^s$ and
(1-b) $r_{j,1}^m < c_i^m \in H^m \Rightarrow r_{j,1}^s < c_i^s \in S^s$.

(2) we control the slave to execute write operations in the same order as the master's history H^m, i.e.,

$$w_{i,p}^m(x_i) < w_{i,p+1}^m(x_i) \in H^m \Rightarrow w_{i,p}^s(x_i) < w_{i,p+1}^s(x_i) \in S^s$$

We have the following property of LSIR.

Theorem 1 (LSIR). *The slave database R^s is consistent with the master database R^m if schedule S^s in the slave database is determined based on the LSIR under SI.*

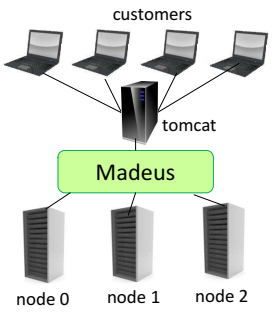

Figure 1: Model of Madeus

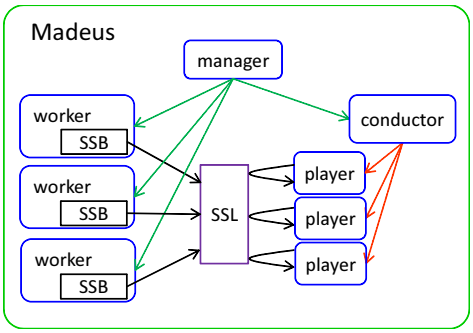

Figure 2: Architecture of Madeus

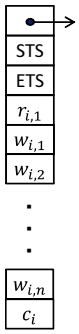

Figure 3: SSB

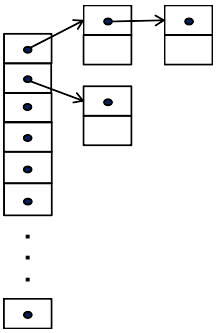

Figure 4: SSL

4. MADEUS

We now explain our lazy replication pure middleware approach called Madeus, which is designed to efficiently handle hot spots in DBaaS since they can trigger SLA violation and reduce customer satisfaction with the service. Although we can extend the current lazy replication pure middleware approach [36, 37] for database live migration, it is not effective since the corresponding serial protocols are not designed for efficient live migration. In this section, we discuss our new protocol that concurrently propagates not only the first read and write operations but also commit operations. Section 4.1 describes the operations (queries) concurrently propagated with Madeus. We explain the architecture of Madeus in Section 4.2. In Section 4.3, we outline live migration with Madeus. We discuss the details of the algorithms of Madeus in Section 4.4. In Appendix C we give an example of the behavior of Madeus. In Section 4.5, we argue that Madeus theoretically guarantees database live migration without sacrificing consistency between master and slave tenants.

4.1 Eff cient Concurrent Propagation

As shown in Definition 3, the LSIR is a scheduling rule defined for the two operation pairs of (1) commit operation c_i^s and the first read operation $r_{j,1}^s$ and (2) write operations $w_{i,p}^s(x_i)$ and $w_{i,p+1}^s(x_i)$. Since DBMSs do not generally have the functionality of controlling operation order, the operation pairs must be propagated serially to keep the order. Conversely, the LSIR defines no scheduling rule for any operation pairs other than the two operation pair noted above. This indicates that, except for the two operation pairs, we can concurrently propagate any operation pairs. Specifically, we can concurrently propagate the following operation pairs.

- First read operations: Since the LSIR provides no relationship between the first read operations, we can propagate them concurrently. For example, if we have $r_{i,1}^s < c_k^s$ and $r_{j,1}^s < c_k^s$, we can propagate the read operations $r_{i,1}^s$ and $r_{j,1}^s$ concurrently.
- Write operations: We can propagate write operations belonging to different transactions concurrently since the LSIR provides no relationship between write operations. For example, we can concurrently propagate write operations $w_{i,p}^s(x_i)$ and $w_{j,q}^s(x_j)$ since they come from different transactions.
- Commit operations: We can concurrently propagate commit operations unlike with the previous approach [24]. This is because the LSIR does not provide any relationship between commit operations. For example, if we have $c_i^s < r_{k,1}^s$ and $c_j^s < r_{k,1}^s$, we can concurrently propagate commit operations c_i^s and c_j^s.

Among the above three operation pairs, the pair of commit operations (third pair) is most effective in terms of efficient live migration. This is because, if we propagate commit operations c_i^s and c_j^s concurrently, DBMSs can execute a group commit where multiple commit operations induce only one disk access. This can significantly reduce the high I/O cost of database live migration.

4.2 Architecture

Figure 1 shows the model of Madeus. To receive operations sent to a DBMS, Madeus lies between customers and DBMSs. Madeus uses commodity hardware and has no shared disk, resulting in low cost; Madeus is a so-called shared-nothing architecture. It uses off-the-shelf DBMSs guaranteeing SI and does not modify them. Each node runs a single DBMS instance, and each instance has multiple databases (tenants), i.e., a shared process model. A customer of a service uses a particular database. For example, a cluster consists of three nodes: 0, 1, and 2 running DBMS instances. Node 0 has tenant A (database A for customer A), and node 1 has tenants B and C (databases B and C for customers B and C, respectively). Node 2 has no database. In this example, customers B and C share the same DBMS instance.

When a customer sends an operation, Madeus receives it and picks up necessary information by parsing the operation. Then, it transmits it to the customer's tenant. After receiving the response of the operation from Madeus, the customer sends a new operation.

Figure 2 shows the architecture of Madeus with its four types of threads: single *manager*, multiple *workers*, single *conductor*, and multiple *players*. All the components of Madeus run on one node because the management of a few DBMS nodes for Madeus introduces little overhead and does not become a bottleneck. If we managed several hundreds of DBMS nodes, we would consider locating some components to one node and the others to another for load balance. When a customer sends an operation, a *worker* receives it and transmits it to the node that has the customer's database. The *worker* dynamically allocates its own *syncset buffer (SSB)* to store the operation as an element of a syncset. The SSBs are linked to the *syncset list (SSL)*. The *manager* is an administration thread that controls all the

other threads. When an operator issues a migration order, the *manager* receives the order and sends it to the other threads. The multiple *players* perform an important role in concurrently propagating operations. A *conductor* notifies the timing at which each *player* should propagate syncsets.

Figure 3 shows the architecture of an SSB. It has rooms to store a *start timestamp (STS)* and *end timestamp (ETS)*. Additionally, an SSB has multiple entries for each element of the syncset. It is important that the entries be held in a first-in, first-out (FIFO) queue to maintain the query order. Figure 4 shows the architecture of an SSL. It is the list of pointers to SSBs. One pointer has SSBs that have the same STS; *players* concurrently propagate the SSBs.

We can prepare a standby Madeus node in addition to an active Madeus node to circumvent a single point of failure. Since Madeus keeps a small amount of state information for normal processing, we can smoothly switch the active Madeus node to the standby Madeus node during normal processing. The standby Madeus node restarts from the first step if the active Madeus node malfunctions during migration processing. We can provide multiple slaves to avoid stopping migration; Madeus can propagate syncsets to multiple slaves at the same time. If a slave fails, Madeus discards the slave and continues to propagate the remaining syncsets to the others. It can also synchronize multiple active masters with Pangea's protocol [33]; an eager replication middleware approach to synchronize multiple masters.

4.3 Database Live Migration

Madeus migrates a tenant from master to slave by the following steps.

Step 1: Creating a snapshot. When an operator issues a database live migration order to Madeus, Madeus creates a snapshot of the master by issuing a dump transaction. During creation, the master executes the customer's operations. Note that the snapshot does not include the customer's operations during snapshot creation. Therefore, Madeus saves the operations as a syncset.

Step 2: Creating a slave. After the snapshot has been created, Madeus creates a slave on the destination using the snapshot while the master executes the customer's operations. Madeus continues to save them as syncsets.

Step 3: Propagating syncsets. After the slave has been created, Madeus propagates syncsets to the slave. We concurrently propagate a minimum query set with minimum serialization. At the same time, the master executes the customer's operations, and Madeus saves the syncsets.

Step 4: Conducting switch-over. Once Madeus propagates all syncsets, the master and slave become consistent. At this time, Madeus fires a *switch-over* operation that redirects the customer's operations from the master to the slave.

Unlike current propagation approaches [24, 36, 37], our key idea is how to concurrently propagate a minimum query set with minimum serialization according to the LSIR in Step 3. Current approaches are too restrictive; serial propagation of all queries [36, 37] or serial propagation of commit queries [24]. Commit serialization induces serious problems: it produces the overhead of mutex lock completion and lessens the benefit from group commit. Madeus avoids these drawbacks and improves performance by concurrently propagating a minimum query set with minimum serialization based on the LSIR.

Algorithm 1 Worker for update transaction

```
 1: if first read operation then
 2:     enter critical region;
 3:     /* there is no commit operation executed */
 4:     send operation to master;
 5:     receive response from master;
 6:     STS := MLC;
 7:     allocate SSB;
 8:     save first operation to SSB;
 9:     leave critical region;
10:     send response to customer;
11: else if write operation then
12:     send operation to master;
13:     receive response from master;
14:     send response to customer;
15:     save operation to SSB;
16: else if commit operation then
17:     enter critical region;
18:     /* there is no first read operation executed */
19:     send operation to master;
20:     receive response from master;
21:     ETS := MLC++;
22:     save operation to SSB;
23:     if during migration then
24:         link SSB to SSL;
25:     else
26:         discard SSB;
27:     end if
28:     leave critical region;
29:     send response to customer;
30: else
31:     send operation to master;
32:     receive response from master;
33:     send response to customer;
34: end if
```

Algorithm 2 Worker for read-only transaction

```
1: send operation to master;
2: receive response from master;
3: send response to customer;
```

4.4 Algorithms

In this section, we explainthe detailed behavior of *workers*, *manager*, *conductor*, and *players*.

4.4.1 Workers

A *worker* manages the *master logical clock (MLC)* to determine the relative order of operations. The *worker* saves the MLC as an STS when it stores the first read operation in the SSB. The *worker* also saves the MLC as an ETS when it stores a commit operation in the SSB.

Algorithm 1 is the behavior of a *worker* upon receiving an update transaction. In this algorithm, the first read operation and commit operation are executed exclusively to determine the relative order between them (lines 2-9 and lines 17-28). When a *worker* receives the first read operation, it creates its own SSB since the operation starts a transaction (line 7). The *worker* stores the first operation as an element of a syncset since the slave must replay inter-rw/wr-dependencies or intra-rw-dependency (line 8). The MLC increases by one whenever a commit operation is executed. This is because the commit operation of an update transaction creates a new state of the database, namely, a new snapshot (line 21). An SSB including a syncset is linked to the SSL only during live migration (lines 23-27).

If a transaction is read-only, Madeus does not need to save any operation (Algorithm 2).

4.4.2 Manager

Algorithm 3 describes the behavior of the *manager*. The *manager* controls Steps 1, 2, 3, and 4 of database live migration (see Section 4.3). In this critical region, no commit operation is executed; this means that the MLC is not changed

Algorithm 3 Manager

```
1: /* Step 1 */
2: enter critical region;
3: CreateSnapshot();
4: MTS := MLC;
5: leave critical region;
6: WaitUntilSnapshotCreated();
7: /* Step 2 */
8: CreateDatabase();
9: WaitUntilDatabaseCreated();
10: /* Step 3 */
11: StartConductorAndPlayers();
12: WaitUntilAllSyncsetsExecuted();
13: /* Step 4 */
14: SuspendAllTransactions();
15: WaitUntilAllSyncsetsExecuted();
16: SwitchOverFromMasterToSlave();
17: ResumeAllTransactions();
18: EndConductorAndPlayers();
```

Algorithm 4 Conductor

```
1: SLC := GetSmallestSTS();
2: /* step 3 */
3: while SLC ≤ MLC do
4:     OrderToPropagateTheFirstOperation();
5:     WaitUntilAllTheFirstOperationPropagated();
6:     oldSLC := SLC;
7:     SLC := GetSmallestSTS();
8:     CCN := GetConcurrentCommitNumber();
9:     OrderToPropagateCommitOperation();
10:    WaitUntilAllCommitOperationPropagated();
11: end while
12: WakeUpManager();
```

in the region (lines 1-5). For Step 1, once the CreateSnapshot() function launches the transaction that creates a snapshot of the master, it returns control (line 3), then the *manager* waits until the snapshot has been created (line 6). For Step 2, after creation of snapshot, the *manager* creates a slave on the destination node using the snapshot (line 8). Step 3 conducts concurrent propagation of syncsets; to this end, the *manager* creates the *conductor* and *players* then leaves Step 3 to them (lines 11-12). When the slave catches up with the master, i.e., all SSBs linked to the SSL have been propagated to the slave, the *manager* transits to Step 4. To ensure the slave has become consistent with the master, the *manager* suspends sending operations (line 14). After all transactions are suspended and all syncsets propagated (line 15), the *manager* executes *switch-over* to change the destination of operations from master to slave (line 16). The *manager* resumes sending operations (line 17).

4.4.3 Conductor

Algorithm 4 shows the algorithm of the *conductor*. The *conductor* plays an important role in achieving concurrent propagation of syncsets by elegantly coordinating multiple *players*. The *conductor* manages the *slave logical clock (SLC)* to control operation propagation without violating consistency. The *conductor* uses an important local variable, *concurrent commit number (CCN)*, which refers to the number of commit operations concurrently being executed. The *conductor* obtains the SLC as the smallest STS by scanning the SSL (line 1). While the condition SLC ≤ MLC holds, the following steps are repeated (lines 3-11): The *conductor* orders *players* to concurrently propagate the first read operations whose STS equals the SLC (line 4). The *conductor* waits until all the first operations have been propagated (line 5). The *conductor* obtains the next SLC (line 7) and calculates the CCN using the next SLC (line 8). The *conductor* orders

Algorithm 5 Player

```
1: /* propagate first operation */
2: WaitUntilOrder();
3: SendOperation();
4: RecvResponse();
5: InformToConductor();
6: /* propagate write operations */
7: while write operation exists do
8:     SendOperation();
9:     RecvResponse();
10: end while
11: /* propagate commit operation */
12: WaitUntilOrder();
13: SendOperation();
14: RecvResponse();
15: InformToConductor();
```

players to concurrently propagate the commit operations whose ETS meets the following equation (line 9):

$$oldSLC \leq ETS \leq oldSLC + CCN - 1 \qquad (1)$$

For example, if the current SLC is 3 and the next SLC is 5, the CCN is $2(= 5 - 3)$. In this case, Equation (1) is computed as $3 \leq ETS \leq 3 + 2 - 1 = 4$. Therefore, the *conductor* gives the order of concurrent propagation to the *players* if the *players* have the SSBs whose ETSs are 3 and 4. Note that this process can result in group commit. The *conductor* waits until all the commit operations have been propagated (line 10). When the *conductor* finishes the loop, it wakes the *manager* up (line 12).

4.4.4 Players

Algorithm 5 shows the algorithm of a *player*; players propagate syncsets concurrently to the slave. Note that the conductor gives orders to propagate syncsets for players, and workers relay packets between a customer and master tenant and create syncsets. A *player* waits until the *conductor* orders the propagation of the first operation (line 2). If the *player* receives the order, it propagates the operation to the slave (line 3). When the *player* receives a response (line 4), it informs the *conductor* (line 5). A *player* propagates all write operations in FIFO order (lines 7-10). The *player* waits until the *conductor* orders the propagation of a commit operation (line 12). Upon receiving the order, the *player* propagates the operation to the slave (line 13). Since multiple *players* propagate commit operations, we can benefit from group commit. After the *player* receives a response (line 14), the *player* informs the *conductor* of it (line 15). See Appendix C for an example of Madeus behavior.

4.5 Theoretical Analysis

This section discusses the theoretical property of Madeus.

4.5.1 Live Migration Consistency

Theorem 2 (Live migration consistency). *Madeus conducts database live migration guaranteeing that the slave is consistent with the master under SI.*

Since the LSIR is a loose rule, Madeus provides opportunities to propagate syncsets concurrently. More importantly, since Madeus concurrently propagates commit operations, we can benefit from group commit for the operations. The experiments discussed in the next section show the effectiveness of Madeus relative to current approaches.

4.5.2 Effectiveness of LSIR

In this section, we theoretically discuss the effectiveness of the LSIR. Let C_r, C_w, and C_c be the cost of read, write, and commit operations, respectively. Let N_r and N_w be the number of read and write operations in one transaction, respectively. Let the total number of transactions be N_{total}. Let C_c' and N' be the cost of the group commit operation and the number of group commit operations, respectively. Then, let the total cost of Madeus be C_{madeus} and the total cost that does not include any rules of the LSIR be (C_{ALL}) as follows[3]:

$$C_{madeus} = N_{total}(C_r + N_w C_w) + N' C_c' + (N_{total} - N') C_c \quad (2)$$

$$C_{ALL} = N_{total}(N_r C_r + N_w C_w + C_c) \quad (3)$$

Therefore, the difference is as follows:

$$C_{ALL} - C_{madeus} = N_{total}(N_r - 1) C_r + N'(C_c - C_c') \quad (4)$$

Considering $N_r \geq 1$, $N' \geq 0$ and $C_c > C_c'$, $C_{madeus} - C_{ALL} \geq 0$; C_{madeus} is never larger than C_{ALL}. Furthermore, as N_{total} or N' is larger, $C_{madeus} - C_{ALL}$ is larger. This means that, under heavy workload, there are many concurrent operations, i.e., N_{total} and N' are large; therefore, Madeus improves performance.

5. EXPERIMENTAL EVALUATION

We implemented Madeus and conducted a performance evaluation with the TPC-W benchmark [15] to investigate its migration time, response time, and throughput. Section 5.3 discusses migration times for various middleware. Section 5.4 discusses the performance with an 800-MB database. Section 5.5 discusses the effect of changing database size. Section 5.6 answers the question which database should be migrated, heavier or lighter, in a multitenant environment.

5.1 TPC-W Benchmark

In assessing a cloud computing service, it is important to evaluate the end-to-end performance of practical enterprise applications involving transaction processing. Hence, we used the TPC-W benchmark [15]. This benchmark models customers that access an online bookstore. It simulates three different profiles by varying the ratio of browsing requests: shopping, browsing, and ordering mixes. All three profiles consist of the same basic read-only or update interactions. The difference between the three profiles is the ratio of read-only interactions to update interactions. In browsing, 95% of interactions are read-only. For shopping, the percentage is 80%. For ordering, the percentage is 50%. We selected the ordering mix for our tests since the update-intensive workload is more severe for replication than shopping or browsing.

The TPC-W benchmark allows us to study different workloads by varying the number of emulated browsers (EBs). Each EB simulates one customer who issues a request to an application server and receives a response. After receiving a response, the EB waits for a specified time then issues the next new request. As the number of EBs increases, the workload becomes heavier.

[3]Although there is an abort operation, the frequency is very small; therefore, we ignore the cost.

Table 2: Difference among middleware approaches

	MIN	CON-FW	CON-COM
B-ALL			
B-MIN	√		
B-CON	√	√	
Madeus	√	√	√

5.2 Experimental Setup

In our experiment, a dedicated node was used for every component, i.e., middleware, PostgreSQL, Tomcat, and load generator of EBs. Thus, we used one node for the master, one node for the slave, one node for middleware, one node for tomcat, and one node for load generator for the experiments discussed in Sections 5.3, 5.4 and 5.5. For the experiment discussed in Section 5.6, we used one node for the master, one node for the slave, one node for the middleware, three nodes for tomcat, and three nodes for the load generator to emulate a multitenant environment. We used only one machine per middleware for all experiments. This is because the middleware node did not become a bottleneck at any time during our experiments. According to the percentages of total CPU time, determined by the `vmstat` command, the time spent idle was almost 100% at any time. All machines had the same configuration of one 3.1-GHz Xeon E3-1220 CPU, 4-core, 4-thread, 8-MB-cache, 16-GB RAM and one 250-GB SATA HDD. All machines were connected through a 1-Gbps Ethernet LAN. The software used was Linux kernel 2.6.32, PostgreSQL 9.2.6, Tomcat 7.0.27. We used an open source tool of the TPC-W benchmark [10].

We used PostgreSQL as the underlying DBMS without modification since PostgreSQL provides SI. A customer does not send a transaction as a unit but operations individually. When a customer sends an operation, Madeus receives it and picks up necessary information by parsing the operation. To interpret the operation directly, we implement the `libpq` and type 4 JDBC protocol of PostgreSQL. Then, Madeus submits the operation to the DBMS. Next, after Madeus receives an answer from the DBMS, Madeus transmits it to the customer. If we modify this protocol, we can use other DBMSs. While Madeus is the first complete and practical middleware implementation, it uses less than 5,000 lines of C code. This is because the LSIR is simple; the implementation of Madeus is not complicated.

A preliminary experiment with one tenant was conducted to examine how many EBs create light, medium, and heavy workloads in the environment. As shown in Figure 5, mean response times were less than 100 ms for 100, 200, and 300 EBs. For 400, 500, and 600 EBs, the mean response times were over 100 ms but less than 2 s. When we increased the workloads further, the mean response times were over 2 s for 700, 800, 900, and 1000 EBs. From the 2-second-rule [2, 3, 4], heavy workloads were set at 700, 800, 900, and 1000 EBs. Additionally, 400, 500, and 600 EBs were set as medium workloads and 100, 200, and 300 EBs as light workloads. Therefore, we selected 100, 400, and 700 EBs to measure light, medium, and heavy workloads, respectively.

5.3 Migration Time

We investigated the efficiency of Madeus conducting database live migration by concurrent propagation. We compared the migration time of Madeus against three baseline middleware approaches with current propagation protocols.

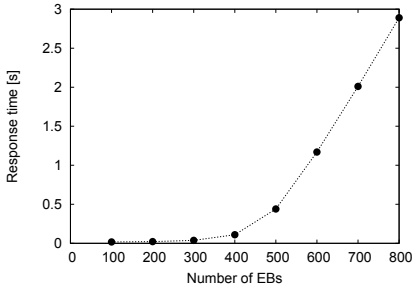

Figure 5: Results of preliminary experiment

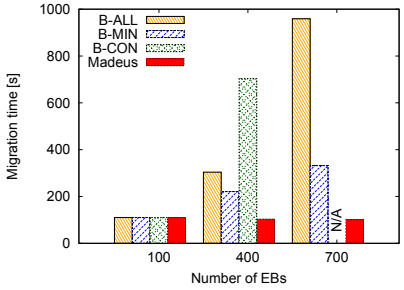

Figure 6: Migration time of each approach

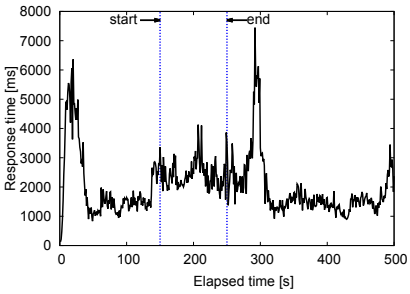

Figure 7: Response time of 800-MB Database

5.3.1 Three Baseline Middleware Approaches

To the best of our knowledge, there are no pure middleware approaches that provide database live migration. Therefore, we implemented three baseline middleware approaches that exploit other propagation protocols in providing database live migration. The first is *baseline middleware with all serial propagation (B-ALL)*. It uses a serial propagation protocol of all transactions. If the master concurrently executes multiple transactions, B-ALL serializes them in commit order, and the slave serially executes them in that order. The second is *baseline middleware with minimum serial propagation (B-MIN)*. It uses a serial propagation protocol of minimum syncsets, the same as in previous studies [36, 37]. The difference with B-ALL is that B-MIN implements the sending of minimum query sets of the LSIR. The third is *baseline middleware with concurrent propagation (B-CON)*. It uses a similar propagation protocol of syncsets as in the previous study [24]. Note that B-CON concurrently propagates the first read operation and write operations similar to the LSIR but does not commit operations. Therefore, B-CON cannot benefit from group commit, unlike Madeus. We implemented the functionality that propagates the minimum query set as well as the first read operation and write operations stated in Section 3.4.

We summarize the differences among B-ALL, B-MIN, B-CON, and Madeus in Table 2. In this table, if we implement the functionality of propagating the minimum query set, the abbreviation MIN is used. If we implement the concurrent propagation of the first read operations and write operations for live migration, the abbreviation CON-FW is used. If we implement the concurrent propagation of commit operations, the abbreviation is CON-COM.

5.3.2 Results

We selected a database size of 800 MB (100 browsers, 100,000 items) based on prior studies [23, 24, 25]. Figure 6 shows the migration times. The x-axis represents different workloads (the number of EBs) and the y-axis represents the migration time from the start to end of the database live migration.

Under light workload (100 EBs), the migration times of B-ALL, B-MIN, B-CON, and Madeus were almost the same at 110 s. Under medium workload (400 EBs), the migration times of B-ALL, B-MIN, B-CON, and Madeus were 304, 221, 703, and 104 s, respectively. Under heavy workload (700 EBs), the migration times of B-ALL B-MIN, B-CON, and Madeus were 959, 332, N/A, and 101 s, respectively. The migration times of B-ALL and B-MIN linearly increased. The migration time of B-CON increased dramatically under medium workload. We could not measure the migration time of B-CON under heavy workload since the slave could not catch up with the master. The migration time of Madeus decreased slightly as we increased workloads.

In Madeus, the total time for database live migration under heavy workload (approximately 101 s) was shorter than those under light and medium workloads (approximately 110 and 104 s). The reasons are twofold. First, recall that the LSIR means the minimum sequential order to synchronize the slave with the master. The total amount of queries is smaller under light workload, but most of them must be propagated serially because the queries are executed serially on the master. The frequency of the queries that must be propagated serially under heavy workload is probably almost the same as that under light workload. However, under heavy workload, because many queries besides the serial queries are executed concurrently on the master, the queries can be propagated concurrently to the slave. Therefore, the total amount of queries executed concurrently during a unit time under heavy workload is larger. Consequently, the slave under heavy workload quickly becomes "warmer", and this results in shorter time to migrate. The second reason is that Madeus can groups concurrent commit queries and benefits from group commit because Madeus has many concurrent commit queries under heavy workload.

By comparing B-ALL with B-MIN, B-CON, and Madeus, we recognize the benefit of the LSIR since B-ALL does not have any of the following three rules of the LSIR; (1) produce a minimum operation set to synchronize the slave with the master, (2) control the sequential order between the first read operation and commit operations, and (3) control the sequential order of write operations. By comparing B-ALL with B-MIN, the migration time of B-MIN was shorter than that of B-ALL. Therefore, we recognize that the first rule is useful.

By comparing B-ALL with B-CON, the migration time of B-CON was longer than that of B-ALL. Although we benefit from concurrent propagation, sequential commit propagation lessens this advantage. To conduct commit operations in the master's transaction order in B-CON, all players compete for the pthead mutex lock at every commit time; B-ALL, B-MIN, and Madeus do not include this code. Therefore, only B-CON struggles against the competition. This overhead increases the response time of B-CON. Consequently, B-CON lessens the advantage of the concurrent propagation of operations.

By comparing B-ALL with Madeus, we can recognize the advantage of the LSIR because B-ALL does not implement

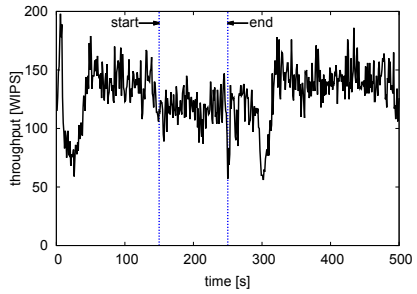

Figure 8: Throughput of 800-GB Database

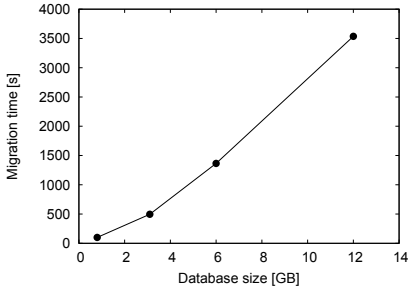

Figure 9: Migration time with changing database size

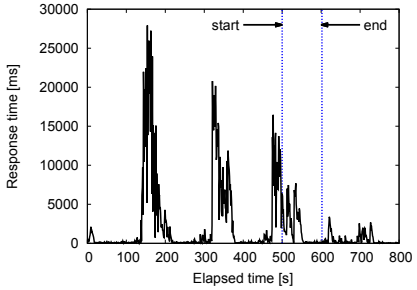

Figure 10: Response time of tenant A when tenant B was migrated

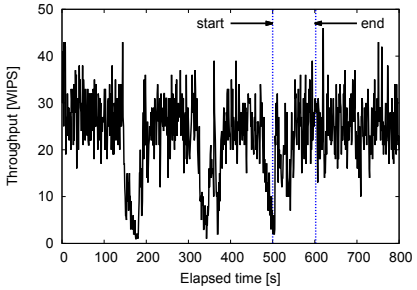

Figure 11: Throughput of tenant A when tenant B was migrated

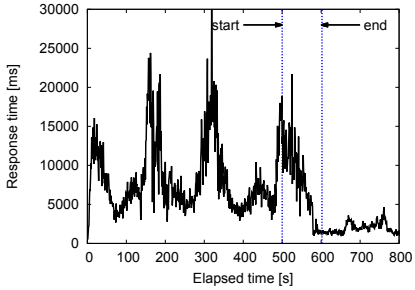

Figure 12: Response time of tenant B when tenant B was migrated

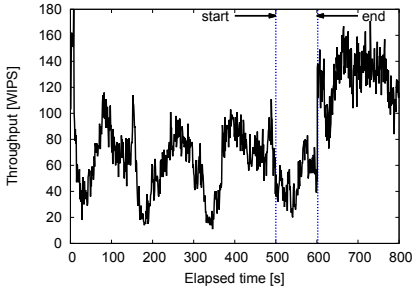

Figure 13: Throughput of tenant B when tenant B was migrated

any of the rules of the LSIR, but Madeus implements all the rules. We can confirm the discussion stated in Section 4.5.2 experimentally, as shown in Figure 6. Under light workload, the response times of Madeus and B-ALL were very similar; therefore, the LSIR results in little benefit. However, Madeus was 9.5 times faster than B-ALL at 700 EBs under heavy workload. Consequently, the LSIR was effective, especially under heavy workload.

5.4 Performance

In this experiment, we measured the response time and throughput of one tenant in Madeus. We selected a database size of 800 MB (288,000 customers, 100,000 items). Figure 7 shows the response time under heavy workload (700 EBs). The degradation between 0 and 50 s was caused by the warming up of the TPC-W benchmark. Database live migration started and ended at approximately 150 and 250 s, respectively. At the start of migration, the response time was increased. This is because, to obtain the MTS, the *manager* entered the critical region, which ensured that no commit operations would be executed (lines 17-28 in Algorithm 1 and lines 2-5 in Algorithm 3). The response time was also increased at the end of migration since the *manager* suspended all transactions to conduct switch-over (lines 14-15 in Algorithm 3). Fortunately, the response time during migration was only slightly longer than those in normal operation. There is a whisker around 290 s, which was caused by a checkpoint of PostgreSQL. Figure 8 shows the throughput under heavy workload (700 EBs). The degradation between 0 and 50 s was caused by the warming up of the TPC-W benchmark. Database live migration started and ended at approximately 150 and 250 s, respectively. At the start and end of migration, the throughput decreased due to small suspension, as stated above. During migration, the throughput was only slightly smaller than those in normal operation.

Table 3: Database size

items	emulated browsers	database size [GB]
100000	100	0.8
500000	500	3.1
1000000	1000	6.2
2000000	2000	12

We observed a decrease of around 290 s because of a checkpoint of PostgreSQL. Considering that the degradation of the checkpoint was larger than those imposed by migration, the migration overhead was quite small.

5.5 Changing Database Size

Figure 9 shows the migration time of Madeus with varing database sizes, 0.8, 3.1, 6.2, and 12 GB with heavy workload (700 EBs). The database sizes are determined by setting two parameters of the TPC-W benchmark, items and emulated browsers, as shown in Table 3. The migration times were 101, 496, 1365, and 3536 s, respectively. As database size increased, the migration time increased. When Madeus creates databases on the slave, Madeus not only inserts data but also alters the attributes of the databases and creates indexes. Therefore, creating databases takes longer than dumping databases. This longer time induces many syncsets and a longer time to propagate the syncsets. Generally, all live migration approaches have this challenge because as the size of database increases, the time to propagate the database increases and then the data to synchronize the slave with the master increases. Considering Madeus conducts migration with real data size in realistic time, Madeus is effective.

5.6 Performance in Multi-tenant Environment

In practical use, multiple tenants are located on the same node, and one tenant with heavy workload can burden the

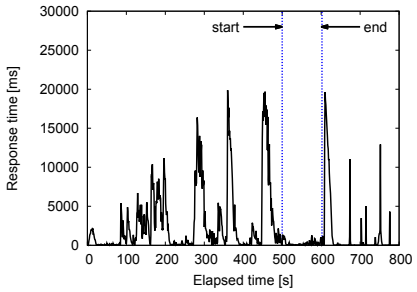

Figure 14: Response time of tenant A when tenant C was migrated

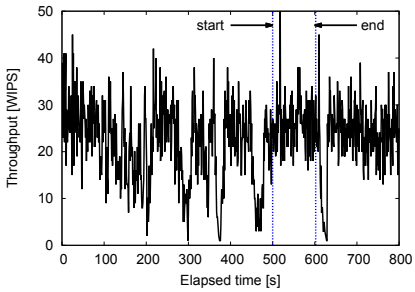

Figure 15: Throughput of tenant A when tenant C was migrated

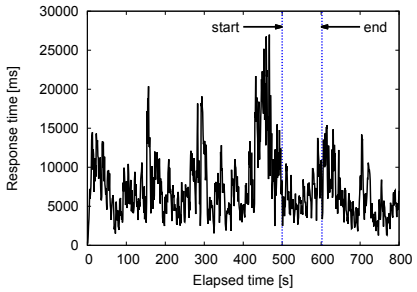

Figure 16: Response time of tenant B when tenant C was migrated

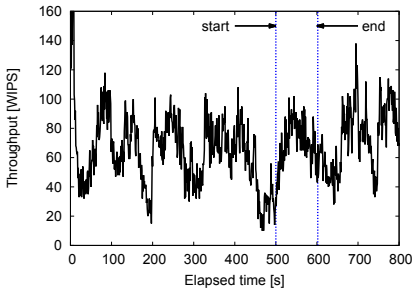

Figure 17: Throughput of tenant B when tenant C was migrated

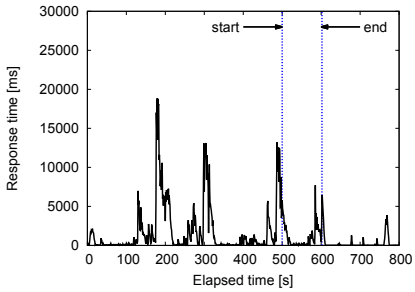

Figure 18: Response time of tenant C when tenant C was migrated

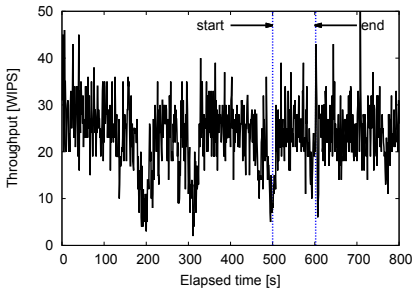

Figure 19: Throughput of tenant C when tenant C was migrated

others. Database live migration is necessary to address this situation. The question is, which should be migrated: a light or heavy tenant? We conducted an experiment to answer this question. The cluster used for this experiment consisted of two nodes: nodes 0 and 1. First, node 0 had three tenants (databases): tenants A, B, and C. Node 1 had no tenants. In addition, tenant B had a heavy workload (700 EBs), and tenants A and C had light workloads (200 EBs). In this experiment, node 0 was a hot spot. We evaluated the response times and throughputs for the following two cases.

Case 1: Migrating heavy tenant. Tenant B with heavy workload was migrated from node 0 to node 1, and we evaluated the response time and throughput. The response times of tenants A and B are shown in Figures 10 and 12, respectively, and the throughputs of tenants A and B are shown in Figures 11 and 13, respectively. We omit the response time and throughput of tenant C since they were similar to those of tenant A. Database live migration started at approximately 500 s and ended at approximately 600 s. Therefore, about 100 s was needed to migrate heavy tenant B.

Although the workload of tenant A was light, as shown in Figure 10, its response time did not decrease since the heavy workload of tenant B burdened tenant A. We can see the performance degradation due to checkpointing of PostgreSQL in Figures 10 and 11. Figure 10 indicates that, in tenant A, the maximum response time during migration was less than that during normal processing; the response time of tenant A was not affected by migration. After migration, the response time of tenant A decreased since tenant B with heavy workload had been migrated. Figure 11 indicates that the throughput during migration is similar to that of normal processing for tenant A.

Figure 12 shows that the response time of tenant B was long during normal processing since its workload was heavy. Fortunately, the response time during migration was not longer than that during normal processing. Throughput degradation was also small during migration, as shown in Figure 13. This means that the overhead of migration processing was very slight. Since the slave on node 1 executed queries during migration, node 1 was not cold after migration. Therefore, we could only see small performance deterioration just after tenant B migration. This suggests that we can avoid a cold cache of the slave, which can lead to an increase in response time [23]. As a result, the response time of tenant B with heavy workload decreased after migration to node 1, which initially had no tenant. More importantly, we can see that throughput increased after migration.

Case 2: Migrating light tenant. Tenant C with light workload was migrated from node 0 to node 1. Figures 14, 16, and 18 plot the response times of tenants A, B, and C, respectively. Figures 15, 17, and 19 plot the throughputs of tenants A, B, and C, respectively. Database live migration started and ended at approximately 500 and 630 s, respectively; migration of the light tenant took about 130 s.

Figure 14 shows that the response time of light tenant A did not decrease due to migration. There are two reasons for this. First, the heavy workload of tenant B still burdened tenant A after migration. Second, the migration time of tenant C was longer than that of tenant B (130 s in Case 2 vs. 100 s in Case 1). This confirms the experimental results on migration time in Section 5.3.2 (Figure 6). In addition, as shown in Figure 16, the response time of tenant B did not decrease since its workload was still heavy after migration of the light tenant; the total workload of node 0 was 900 EBs (200 EBs + 700 EBs) after migration. Figure 18 indicates that the response time of tenant C before migration was similar to that of tenant A (Figure 14). After migration, the response time was very short since node 1 had only tenant C.

Which migration is better?. Which is better, migrating a tenant with light workload or one with heavy workload in a multitenant environment? Our answer is that heavy workload tenants should be migrated. There are two reasons for this; first, we can effectively reduce the response time of a hot spot by eliminating the heavy workload. Second, we can achieve short migration time under heavy workload because of "warm" cache effect and benefit of group commit. The experimental results indicate that Madeus is very useful, especially in heavy workload environments.

6. RELATED WORK

Several related approaches have been proposed for database live migration. However, to the best of our knowledge, there is no efficient migration approach supporting DBaaS.

Live Migration for Virtual Machines. Clark et al. proposed the first approach for live migration of VMs [20]. It uses a pre-copy approach in which pages of memory are iteratively copied from the source machine to the destination host. Bradford et al. proposed an approach for WAN environment [18]. It propagates not only memory image but also data on disks. Hines et al. propopsed post-copy approach [28]. Sväard et al. propose an approach to transfer compressed memory pages to increase migration throughput [39]. Song et al. proposed to transfer pages in pararell [38]. Mashtizadeh et al. proposed to propagate memories and disk image effectively for local and wide area network [31].

Database Live Migration Systems. Minhas et al. proposed RemusDB, a high availability DBMS approach that modifies Remus [21] for DBMS workloads [32]. All migration functionalities are implemented in the VM layer. Furthermore, RemusDB needs modification of DBMSs. Two built-in replication approaches have been proposed. Das et al. proposed to iteratively copy the cache during migration [23]. Their approach starts the slave with a warm cache after migration. Furthermore, it relies on shared disks. Elmore et al. proposed another built-in replication approach [25] in which pages are transmitted according to the index structure. It relies on the impractical assumption that the index structure is immutable during migration. Therefore, if a client requests a transaction to change the index structure, the transaction fails and aborts. Both also involve modification of the DBMS engine, as opposed to our middleware-based approach.

Lazy Replication Middleware. Several approaches have been proposed that use lazy serial propagation [36, 37]. However, such approaches exhibit the problem in which the slave may not catch up with the master. Therefore, serial-propagation-based approaches have low efficiency in propagating queries. The typical and well known solution for this problem is to slow the master by throughput restrictions so that the slave can keep up with the master [19]. Unfortunately, this prevents the efficient use of computing resources.

Lazy Replication Systems. Daudjee et al. proposed a lazy replication system with a similar rule to the LSIR [24]. Their approach is not middleware; it requires the modification of existing DBMSs. Because they assume the first-committer-wins rule[4], as written in a previous paper [17], their proposed rules are too restrictive; the master and slave must execute commit operations in exactly the same order. In fact, practical DBMSs, such as Oracle, SQL Server, and PostgreSQL, adopt the first-updater-wins rule instead of the first-committer-wins rule. Because we assume the first-updater-wins rule, we can propose a relaxed rule, LSIR. Madeus groups concurrent commit operations and benefits from group commit. Moreover, it is based on unrealistic assumptions, i.e., a master's operations are broadcasted in the time-stamped order to slaves and the existence of start operation that creates a snapshot (in fact, a snapshot is created just before the first operation is executed.). Because of these unrealistic assumptions, this approach does not include a practical prototype; evaluation was conducted through simulation.

Log-based Replication Systems. Barker et al. proposed the database live migration middleware called Slacker [16]. It uses a well known PID controller [11]. Slacker creates syncsets by parsing binary transaction logs. Slacker, therefore, is specific to particular versions and types of DBMSs. In addition, it is not based on the shared process model; it requires a DBMS instance for each tenant, the same as with the VM-based model. Therefore, this approach fails to share resources efficiently and degrades performance. PostgreSQL provides log-based replication that uses transaction logs as syncsets [35]. In this approach, all databases (tenants) share the transaction logs; thus, it cannot be used for the shared process model. Furthermore, since the log format is different for a particular version, this does not permit a heterogeneous environment that has multiple versions and multiple types of DBMSs. If we constracted DBMS systems with heterogeneous versions, we could avoid that one bug shut down whole systems.

High Availability Replication Systems. Oracle Real Application Cluster (RAC) [12], IBM DB2 pureScale [13] and Microsoft Windows Server Failover Cluster (WSFC) [14] are commercial high availability systems. Unlike Madeus, they rely on a shared disk, which is not only expensive but also a single point of failure.

7. CONCLUSION

We proposed a middleware approach called Madeus that provides efficient database live migration in DBaaS. Madeus is pure middleware that is transparent to a DBMS, uses commodity hardware and software, and is independent of any dedicated tool. Madeus uses a novel concurrent propagation protocol that makes the slave consistent with the master under SI. This protocol concurrently propagates the first read, write, and commit operations. As a result, our approach can benefit from concurrency and group commit. We implemented Madeus on top of PostgreSQL. The code size is less than 5,000 lines in C language. We conducted experimental evaluations with the TPC-W benchmark. Madeus exhibited shorter migration times than current approaches, especially for heavy workloads. Furthermore, it could resolve a hot spot in a multitenant environment. Madeus is a very practical and effective approach for achieving DBaaS. We can consider an alternative that combines primary/secondary DBMS replication and migration. We will compare this setting to our approach for future work.

[4]When two transactions attempt to modify the same data item, only the first committed transaction modifies successfully and the other aborts.

APPENDIX

A. PROOF OF LEMMA

A.1 Lemma 1

Proof. Let H^m be a history of committed transactions on master database R^m. If master database R^m is executed under SI, the history of the master database H^m does not include any inter-ww-dependencies. This is because the first-updater-wins rule permits only the first-update transaction to commit; the other transactions must abort. If the first-update transaction aborts, the second-update transaction can commit, but the other transactions must abort. In short, only one transaction can commit while the other transactions must abort. Therefore, the history of master database H^m does not include inter-ww-dependencies. Thus, slave database R^s does not need to replay any inter-ww-dependencies. □

A.2 Lemma 2

Proof. We need read operations $r_{i,p}(p > 1)$ to obtain the state of a database. These read operations do not conflict with any operations. In addition, they do not change the state under SI; they read only the snapshot created by $r_{i,1}$. Therefore, slave database R^s does not need to replay any intra-wr-dependencies. □

A.3 Lemma 3

Proof. This is obvious from Lemmas 1 and 2. □

A.4 Lemma 4

Proof. Since T_i is an update transaction, we have $w_{i,p}(x_i) < c_i$. Therefore, $w_{i,p}(x) < c_i < r_{j,1}(x_i)$. From definition 1, this equation corresponds to inter-wr-dependency. □

A.5 Lemma 5

Proof. Since T_j is an update transaction, we have $r_{j,1}(x_k) < w_{i,q}(x_i) < c_i$ or $w_{i,q}(x_i) < r_{j,1}(x_k) < c_i$. If $i = j$ and $r_{j,1}(x_k) < w_{i,q}(x_i) < c_i$, we have $r_{i,1}(x_k) < w_{i,q}(x_i)$. This indicates that transactions T_i and T_j have intra-rw-dependency. If $i = j$ and $w_{i,q}(x_i) < r_{j,1}(x_k) < c_i$, we have $w_{i,q}(x_i) < r_{j,1}(x_k)$. This indicates that transactions T_i and T_j have intra-wr-dependency. From Lemma 2, we can ignore this case. If $i \neq j$, the equation corresponds to inter-rw-dependency. Note that in both cases $r_{j,1}(x_k) < w_{i,q}(x_i) < c_i$ and $w_{i,q}(x_i) < r_{j,1}(x_k) < c_i$, the snapshot of transaction T_j does not include the modification of $w_{i,q}(x_i)$. Therefore, we have intra-rw-dependency or inter-rw-dependency if T_j is an update transaction and $r_{j,1}(x_k) < c_i$. □

A.6 Lemma 6

Proof. If we have $w_{i,p}(x_i) < w_{i,p+1}(x_i)$, it is clear that transaction T_i has intra-ww-dependency from the definition of 1. □

B. PROOF OF LSIR

Proof. From Lemma 3, we can make the slave consistent with the master if we replay the same four dependencies as the master on the slave: inter-wr-dependency, inter-rw-dependency, intra-rw-dependency, and intra-ww-dependency. The rule of $c_i^m < r_{j,1}^m \in H^m \Rightarrow c_i^s < r_{j,1}^s \in S^s$ in rule (1-a) indicates that the scheduler replays the same inter-wr-dependency on the slave database as the master from Lemma 4. In addition, from Lemma 5, the rule $r_{j,1}^m < c_i^m \in H^m \Rightarrow r_{j,1}^s < c_i^s \in S^s$ in rule (1-b) indicates that the scheduler replays the same inter/intra-rw-dependencies on the slave as the master. The rule $w_{i,p}^m(x_i) < w_{i,p+1}^m(x_i) \in H^m \Rightarrow w_{i,p}^s(x_i) < w_{i,p+1}^s(x_i) \in S^s$ in rule (2) indicates that the scheduler replays the same intra-ww-dependency on the slave as the master from Lemma 6. As a result, the LSIR can make the slave consistent with the master under SI. □

C. EXAMPLE OF MADEUS BEHAVIOR

This section gives an example of how Madeus handles operations. We assume that the set of transactions is $\mathcal{T} = \{T_i = r_{i,1}(x_p), w_{i,1}(x_i), c_i, T_j = r_{j,1}(y_q), w_{j,1}(y_j), c_j, T_k = r_{k,1}(x_i), w_{k,1}(x_k), c_k\}$. First, the MLC is 3. Figure 20 shows the behavior of *workers*. When $worker_i$ receives the first read operation $r_{i,1}(x_p)$, it stores the current MLC as an STS and the first read operation $r_{i,1}(x_p)$ in its SSB. $Worker_j$ receives the first read operation $r_{j,1}(y_q)$ and stores the current MLC as an STS and the first read operation $r_{j,1}(y_q)$. When $worker_i$ receives write operation $w_{i,1}(x_i)$, it stores the operation in its SSB. $Worker_j$ receives write operation $w_{j,1}(y_j)$ and stores the operation in its SSB. When $worker_i$ receives commit operation c_i, it stores the current MLC as an ETS and operation c_i in its SSB. Furthermore, it increases the MLC by 1 to 4. Since commit operation c_j goes to $worker_j$, it stores the current MLC as an ETS and operation c_j in its SSB. It also increases the MLC by 1, to 5. When $worker_k$ receives the first read operation $r_{k,1}(x_i)$, it stores the current MLC as an STS and the first operation $r_{k,1}(x_i)$ in its SSB. $Worker_k$ receives write operation $w_{k,1}(x_k)$ and commit operation c_k and stores them in the SSB. It also stores the current MLC as an ETS and increases the MLC by 1, to 6. Figure 21 shows the current SSL. Note that transactions T_i and T_j were executed concurrently but transaction T_k was executed after transactions T_i and T_j have committed.

Figure 22 shows an example of the behavior of the *conductor* and *players*. The *conductor* sets the SLC to 3 since the smallest STS in the SSL is 3. The *conductor* designates $player_i$ and $player_j$ to propagate the first read operations $r_{i,1}(x_p)$ and $r_{j,1}(y_q)$ concurrently. Subsequently, they propagate write operations $w_{i,1}(x_i)$ and $w_{j,1}(y_j)$ concurrently. Since the current SLC is 3 and the next SLC is 5, CCN is 2. Therefore, the two commit operations whose ETS are 3 and 4 can be propagated concurrently. $Player_i$ and $player_j$ propagate commit operations c_i and c_j concurrently. Since both commit operations are propagated at the same time, we can benefit from group commit. Since $player_i$ and $player_j$ increase the SLC by 1, SLC becomes 5. $Player_k$ then propagates the first read operation $r_{k,1}(x_i)$ since the STS of $player_k$ is 5. Then, $player_k$ propagates write operation $w_{k,1}(x_k)$ and commit operation c_k. Note that transactions T_i and T_j are executed concurrently. It is especially important to execute the two commit operations concurrently since this is to receive the benefits of group commit.

D. PROOF OF LIVE MIGRATION CONSISTENCY

Proof A *worker* does not create any SSBs when executing a read-only or abort transaction. When *worker* receives the first read operation of an update transaction, it saves the operation in its SSB (lines 1-10 in Algorithm 1). It discards

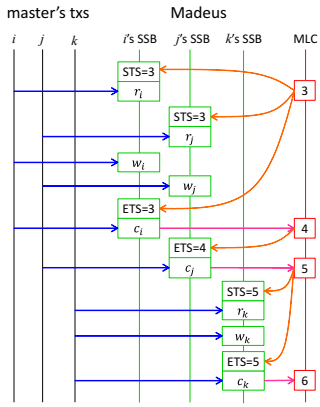

Figure 20: Master's sequence

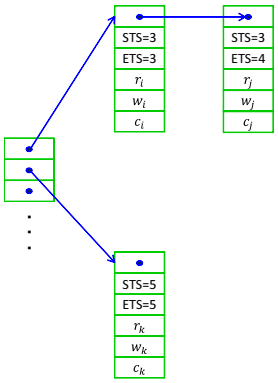

Figure 21: Example of SSB

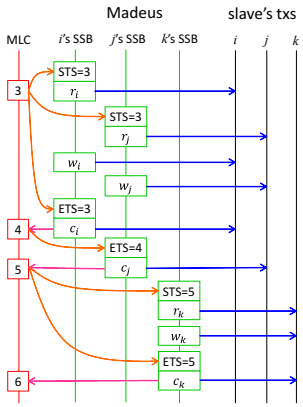

Figure 22: Slave's sequence

the other read operations except for the first operation; it does not save the read operations in the SSB (lines 30-34 in Algorithm 1). If it receives an write operation, it saves the operation in its SSB (lines 11-15 in Algorithm 1). Therefore, the *worker* provides the mapping function $\mathcal{F}$, which is described in Definition 2. With regard to processing operations, Madeus has the following three cases:

(1) For the operations such that $c_i^m < r_{j,1}^m \in H^m$, we have $ETS_i < STS_j$ because the MLC is increased by 1 after commit operation c_i^m is executed (line 21 in Algorithm 1). The *players* propagate the first read operation and commit operations of syncsets with a smaller STS and ETS, as shown in Algorithms 4 and 5. Therefore, if we have $c_i^m < r_{j,1}^m \in H^m$ in the master database, we have $c_i^s < r_{j,1}^s \in S^s$ in the slave database.

(2) For $r_{j,1}^m < c_i^m \in H^m$, we have two cases; there is or is not a commit operation between the first read operation $r_{j,1}^m$ and commit operation c_i^m. If there is no commit operation c_k^m such that $r_{j,1}^m < c_k^m < c_i^m$, we have $STS_j = ETS_i$ because MLC is increased by 1 only after commit operation c_i^m has been executed (line 21 in Algorithm 1). The first read operation $r_{j,1}^m$ and commit operation c_i^m are replaced with the first read operation $r_{j,1}^s$ and commit operation c_i^s, respectively (lines 8 and 22 in Algorithm 1). We propagate the first read operation $r_{j,1}^s$ and commit operation c_i^s (lines 3-10 in Algorithm 4). The *conductor* makes a *player* propagate the first read operation $r_{j,1}^s$ (line 4 in Algorithm 4). The *conductor* then makes sure that the first read operation $r_{j,1}^s$ has been executed (line 5 in Algorithm 4). The *conductor* permits the *player* to propagate commit operation c_i^s (line 9 in Algorithm 4). Therefore, if we have $r_{j,1}^m < c_i^m \in H^m$ in the master, we have $r_{j,1}^s < c_i^s \in S^s$ in the slave if there is no commit operation c_k^m such that $r_{j,1}^m < c_k^m < c_i^m$.

In addition, if there exists commit operation c_k^m such that $r_{j,1}^m < c_k^m < c_i^m$, we have $STS_j < ETS_i$ because the MLC is increased by 1 after commit operation c_k^m has been executed (line 21 in Algorithm 1). The *players* propagate the first read operation and commit operations in syncsets of a smaller STS and ETS, as shown in Algorithms 4 and 5. As a result, if we have $r_{j,1}^m < c_i^m \in H^m$ in the master, we have $r_{j,1}^s < c_i^s \in S^s$ in the slave if there exists commit operation c_k^m such that $r_{j,1}^m < c_k^m < c_i^m$.

(3) Since an SSB uses an FIFO queue and the *player* propagates syncsets in order, the *player* propagates the k-th write operation before the $k+1$-th write operation. Therefore, we have $w_{i,k}^s(x_i) < w_{i,k+1}^s(x_i) \in S^s$ in the slave if we have $w_{i,k}^m(x_i) < w_{i,k+1}^m(x_i) \in H^m$ in the master.

It is clear that Madeus processes the operations by strictly obeying the LSIR described in Definition 3. As shown in Theorem 1, if the schedule in the slave is determined based on the LSIR, the slave is consistent with the master under SI. Therefore, it is clear that Madeus achieves consistent live migration between master and slave under SI.

□

E. REFERENCES

[1] Multitenancy, http://en.wikipedia.org/wiki/Multitenancy.

[2] A new 2 second rule, http://blogs.keynote.com/the_watch/2009/09/the-new-2-second-website-rule.html.

[3] Akamai Reveals 2 Seconds as the New Threshold of Acceptability for eCommerce Web Page Response Times, http://www.akamai.com/2seconds.

[4] Performance Index: Top Retailers, http://www.keynote.com/keynote_competitive_research/performance_indices/top_retailers/index.html.

[5] Amazon RDS, http://aws.amazon.com/.

[6] Microsoft Azure, http://azure.microsoft.com/.

[7] Google Cloud SQL, http://cloud.google.com/.

[8] Percona XtraBackup, http://www.percona.com/.

[9] Blind write, http://en.wikipedia.org/wiki/Blind_write.

[10] Java TPC-W implementation distribution, http://www.ece.wisc.edu/ pharm/tpcw.shtml.

[11] PID controller, http://en.wikipedia.org/wiki/PID_controller.

[12] Oracle RAC, http://www.oracle.com/technetwork/database/options/clustering/rac.

[13] DB2 pureScale, http://www.ibm.com/developerworks/data/library/dmmag/pureScale.

[14] Microsoft Windows Server Failover Cluster, http://www.microsoft.com/windowsserver2008/en/us/failover-clustering-main.aspx.

[15] Transaction processing performance council, tpc-w.

[16] S. Barker, Y. Chi, H. J. Moon, H. Hacigumus, and P. Shenoy. Cut me some slack: Latency-aware live

migration for databases˙In *EDBT*, pages 432–443, 2012.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD*, pages 1–10, 1995.

[18] 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.

[19] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *ACM SIGMOD*, pages 739–752, 2008.

[20] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, pages 273–286, 2005.

[21] B. Cully, G. Lefebre, D. Meyer, M. Feeley, N. Nutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, pages 161–174, 2008.

[22] C. Curino, E. P. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational clound: A database-as-a-service for the cloud. In *CIDR*, pages 235–240, 2011.

[23] S. Das, S. Nishimura, D. Agrawal, and A. E. Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. In *PVLDB*, pages 494–505, 2011.

[24] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726, 2006.

[25] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, pages 301–312, 2011.

[26] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005.

[27] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD*, pages 173–182, 1996.

[28] M. R. Hines and K. Gopalan. Post-Copy Based Live Virtual Macine Migration Using Adaptie Pre-Paging and Dynamic Self-Ballooning. In *VEE*, pages 51–60, 2009.

[29] H. Jung, H. Han, A. Fekete, and U. Röhm. Serializable snapshot isolation for replicated databases in high-update scenarios. In *VLDB*, pages 783–794, 2011.

[30] A. Koto, K. Kono, and H. Yamada. A guideline for selecting live migration policies and implementations in clouds. In *CloudCom*, 2014.

[31] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty. XvMotion: Unified Virtual Machine Migration over Long Distance. In *ATC*, pages 97–108, 2014.

[32] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: Transparent high availabililty for database systems. In *PVLDB*, pages 29–45, 2011.

[33] T. Mishima and H. Nakamura. Pangea: An eager database replication middleware guaranteeing snapshot isolation without modification of database servers. In *PVLDB*, 2009.

[34] 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 Transactions on Database Systems*, 17(1):94–162, March 1992.

[35] PosgreSQL Global Development Group, http://www.postgresql.org.

[36] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, pages 155–174, 2004.

[37] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Fas – a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB*, pages 754–765, 2002.

[38] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. Parallelizing Live Migration of Virtual Machines. In *VEE*, pages 85–96, 2013.

[39] P. Sväard, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. In *VEE*, pages 111–120, 2011.