Downtime-Free Live Migration in a Multitenant Database

Nicolas Michael and Yixiao Shen

Oracle Corporation, 4180 Network Circle, Santa Clara, California 95054
{nicolas.michael,yixiao.shen}@oracle.com

Abstract. Multitenant databases provide database services to a large number of users, called *tenants*. In such environments, an efficient management of resources is essential for providers of these services in order to minimize their capital as well as operational costs. This is typically achieved by dynamic sharing of resources between tenants depending on their current demand, which allows providers to oversubscribe their infrastructure and increase the density (the number of supported tenants) of their database deployment. In order to react quickly to variability in demand and provide consistent quality of service to all tenants, a multitenant database must be very elastic and able to reallocate resources between tenants at a low cost and with minimal disruption. While some existing database and virtualization technologies accomplish this fairly well for resources *within* a server, the cost of migrating a tenant to a *different* server often remains high. We present an efficient technique for live migration of database tenants in a shared-disk architecture which imposes no downtime on the migrated tenant and reduces the amount of data to be copied to a minimum. We achieve this by gradually migrating database connections from the source to the target node of a database cluster using a self-adapting algorithm that minimizes performance impact for the migrated tenant. As part of the migration, only frequently accessed cache content is transferred from the source to the target server, while database integrity is guaranteed at all times. We thoroughly analyze the performance characteristics of this technique through experimental evaluation using various database workloads and parameters, and demonstrate that even databases with a size of 100 GB executing 2500 transactions per second can be migrated at a minimal cost with no downtime or failed transactions.

Keywords: database, live migration, multitenancy

1 Introduction

The rise of virtualization has eased resource provisioning by abstracting from physical resources and allowing to create new, now virtualized, resources on demand. Databases are a central component for most applications, which makes database virtualization an important consideration of virtualized infrastructures. Virtualization implies many challenges on database management systems

(DBMS). Whether used for consolidation or multitenancy in Cloud-based services, the *density* of virtualized databases, that is the number of hosted users or *tenants* on a given amount of physical resources, becomes a key factor for capital and operational cost. Density can be increased both by lowering the footprint (the resource demand) of a tenant as well as by sharing resources between tenants. If the peak resource demand of all tenants exceeds the amount of physical resources, the deployment is referred to as *oversubscribed*. One of the key challenges for multitenant databases is to efficiently manage oversubscribed resources without violating a tenant's service level agreement (SLA). Since the resource needs of a tenant are often unpredictable and can be subject to sudden change, a multitenant database must be able to provide resources on demand to ensure good performance and quality of service for all tenants. This becomes more difficult as resources may currently be in use by other tenants. The ability to quickly reassign resources from one tenant to another is therefore essential for the *elasticity* of a database.

Resources managed by a database typically include CPU, memory (for cache, metadata and other data structures), storage (capacity, as well as I/O bandwidth and latency), and network. The choice of virtualization technology can have significant impact on the cost at which each of these resources can be shared or reassigned between tenants. For example, two tenants that are not bound to particular physical processors might easily share CPU cycles given to them by a scheduler or hypervisor on a granularity of microseconds, while sharing of memory might be more coarse-grained if both tenants are deployed in different virtual machines, each with dedicated memory. Consolidation technologies such as multitenant databases [14] provide an interesting alternative to virtual machines if they can lower the footprint of each tenant by sharing resources more efficiently and dynamically [27]. The sharing of a common cache between tenants for example would reduce the cost of allocating and deallocating cache buffers to the level of traditional cache management with replacement policies such as Least Recently Used (LRU). Load-balancing and resource management within a server are important features for multitenant databases to allocate a proper amount of resources to every tenant at any point in time.

With unforeseen load changes of some tenants, or more tenants being provisioned and becoming active, the physical resources of a server may become exhausted and make load-balancing *across* servers necessary. In such a case, a tenant may have to be *migrated* to another server. Also planned maintenance operations often require to take a server offline and thereby require tenants to be migrated. While maintenance operations might be scheduled during low traffic times such as at night time, load-balancing can become necessary during peak traffic hours. Many database applications, especially when they handle online transactions, may not be able to tolerate any outage of the database as this may lead to a loss of revenue. Those databases cannot be shut down before migration, but must be migrated online with as little disruption as possible, referred to as *live migration*. Minimizing the *downtime*, that is the period of time in

which the database is not able to serve requests, is therefore a key objective of implementing database live migration.

While VM live migration has been studied by many researchers [5][12][13], only few have attempted to migrate databases in virtual machines [18][15]. More recently, some approaches for live migration in multitenant databases have been proposed [7][8][2]. However, none of them accomplish to migrate databases truly free of downtime without failing any requests.

We present a technique for live migration in a multitenant shared-disk database aimed at providing efficient migration of transactional online databases with no service interruption or failed requests and minimal impact on quality of service. The migration is facilitated by a connection pool we have implemented, which migrates connections from the source to the target node using a self-adapting algorithm to control the migration rate. The algorithm throttles or accelerates the rate based on workload behavior and system load, attempting to minimize impact on the migrated tenant while keeping overall migration time low. Rather than copying memory pages, the target node pulls frequently accessed database cache blocks on demand from the source node or the shared storage, reducing the amount of data to be copied compared to traditional VM migration to cache content at most.

Our solution for live migration is implemented for the Oracle Database 12c [9] and leverages the Real Application Clusters (RAC) [21] and Multitenant [19] options. We thoroughly analyze the influence of different workload characteristics on the performance and scalability of our proposed technique through an extensive series of experiments. In our study, we not only consider downtime as a metric, but also analyze our results with respect to total migration time, amount of migrated data, and migration overhead.

The main contributions of this paper are:

- We present a new technique for database live migration in a shared-disk multitenant database using a client-side connection pool with an adaptive connection migration algorithm.
- We evaluate the performance and scalability of this migration technique by using various workloads, and provide an in-depth analysis of key performance metrics based on workload characteristics.
- We demonstrate how this technique successfully accomplishes to migrate database tenants without downtime or failed transactions even for large databases running high transaction rates.

The remainder of this paper is organized as follows: In section 2 we describe the technologies used. In section 3 we present the design and implementation details of our technique. In section 4 we explain the methodology of our experiments and analyze their results. Further considerations are described in section 5, related work is summarized in section 6, and we conclude with section 7.

2 Background

2.1 Connection Pooling

On-Line Transaction Processing (OLTP) workloads often serve hundreds or thousands of requests per second with response time requirements in the range of milliseconds. To avoid the cost of establishing database connections for each request, OLTP applications use connection pools of fixed or variable sizes from which they acquire and release connections as needed. We leverage this by implementing live migration logic inside a connection pool rather than a query router or proxy to avoid extra latency by additional nodes in the communication channel. While this implies that our solution requires a certain degree of cooperation of the client to enable a smooth migration, it is entirely implemented inside the connection pool and therefore transparent for the application.

2.2 Database Live Migration Techniques

The choice of virtualization technology largely determines the possible techniques available for live migration. In the context of database virtualization, we distinguish between two primary concepts of virtualization, using virtual machines or in-database virtualization.

Virtual Machines

A virtual machine (VM) virtualizes the underlying hardware or operating system (OS) and provides a database running inside the VM access to resources such as CPU, memory, I/O, OS kernel, and file systems. Regardless of whether the physical resources are dedicated to a VM or shared between VMs, each database inside a VM is isolated from other databases in the sense that it can only access the (virtualized) resources in its own VM. While this model has advantages with respect to isolation, it also limits the degree of sharing, as it prevents for example the sharing of common data structures or processes across databases in different VMs. Examples of such virtual machines are KVM [16], Microsoft Hyper-V [22], Solaris LDoms and Zones [25], VMWare ESX [30], and Xen [1]. A discussion of their features and differences is beyond the scope of this paper.

VM Live Migration. The live migration of a VM can be accomplished in different ways, which typically include the concepts of *pre-copy* [5][24][3], *stop-and-copy*, and *post-copy* [11] of pages from the source to the target VM. Most VMs like Xen, KVM, and VMWare ESX use a combination of the first two concepts [24][5]. They first attempt to transfer the majority of pages from the source to the target VM while the source VM is still running. This is often done in multiple phases, as pages in the source VM are continuously being modified and some pages may need to be transferred again. After some iterations, the source VM is then brought to a stop, and during a short phase of downtime, remaining pages are copied to the target VM to bring it into a consistent state with the source VM. Operation is then resumed on the target VM.

For a database running in a VM this approach means that not only database content itself is transferred between VMs, but also temporary data, process stacks and heaps, unused cache blocks, operating system pages, and others. Depending on the database size relative to the size of the VM, the memory to be transferred often not only exceeds the (cached) database size, but also the VM size due to repeated transfers of pages [12][11]. The advantage of such a migration is that the database can be completely unaware of the migration and does not need to provide any migration support.

In-Database Virtualization

Virtualization inside the database moves the concept of virtualization into the database layer by hosting multiple tenants inside a common database. By doing so, not only database structures can be efficiently shared between tenants, but also live migration can be implemented in a way that considers the special attributes and characteristics of databases.

Das et al. [7] have shown that by migrating the database cache rather than VM pages in a shared-disk multitenant database, a downtime as short as 300 ms is achievable. In another study, Elmore et al. [8] use a combination of pulling and pushing of database pages in a shared-nothing architecture to migrate a multitenant database without downtime and only few failed operations. In our study, we show that an Oracle Multitenant database running on Real Application Clusters (RAC) can be migrated without any downtime, no failed transactions, and minimal impact on quality of service.

Oracle RAC. Oracle Real Application Clusters is an option of the Oracle database that allows multiple database instances, running on different *nodes* (servers) in a *cluster*, to access a common database simultaneously. The database resides on a shared storage and can be partially or completely cached in each instance, where instances may cache both identical as well as different data blocks. When an instance needs to access a block (e.g. when executing a query), it has to request this block from another instance's cache or from storage if it does not hold the current copy itself. A distributed lock manager keeps track where the current copy of each block is held, guaranteeing data consistency across all instances. Instances communicate over a dedicated private network called *cluster interconnect*. A cache transfer from one instance to another is referred to as *cache fusion* [17].

Oracle Multitenant. Oracle Database 12c introduced a new option called *Oracle Multitenant* that virtualizes databases within the database. The hosting database is referred to as *container database* (CDB), into which virtualized databases called *pluggable databases* (PDB) are deployed (*plugged*). Within a CDB, all PDBs are isolated in terms of namespace, but share a common cache as well as database background processes. The container database can be a RAC database spanning multiple nodes. A database *service* is associated with each PDB. The PDB is accessed by establishing database connections to that service,

which creates a database connection to the node where the service is running. While it is possible to run a service on multiple nodes at the same time and thus connect to the same PDB through multiple nodes, we instead propose the use of *singleton services* that only run on one node at a time. By doing so, all data of a PDB is accessed on one node only, which increases cache reach by avoiding duplicate copies of identical data blocks and reduces cache fusion traffic.

3 Design and Implementation

Based on the Oracle database options *Multitenant* and *Real Application Clusters*, we present a new technique by which a pluggable database can be migrated from one RAC node to another without any downtime. To achieve this goal, we have implemented a connection pool that upon receiving a migration request slowly drains connections to the source node while at the same time establishing new ones to the target node. Our implementation adapts the rate at which connections are migrated automatically to workload behavior and system load and attempts to minimize the impact of migration on ongoing requests while at the same time keeping overall migration time low. By doing so, we smoothly migrate the database from one node to another, allowing the target node enough time to fetch frequently accessed cache blocks from the source node without causing disruption for the migrated tenant. During the migration, the database is accessed in both nodes simultaneously, while data integrity is maintained by Oracle RAC's cache fusion protocol and distributed lock manager [17].

3.1 Service Migration

The migration is initiated by relocating the singleton service associated with the tenant's pluggable database to another node. During service relocation, the service is first stopped on the current node and then started on the target node. As part of starting the service, the associated PDB is opened on the target node (if it has not been opened before), which just requires metadata operations that typically take a few seconds only. Even though the service may be down for a short period during migration, already established connections remain usable, so the database continues to serve requests even if the service is down. The only consequence is that *new* connections cannot be established during this time, for example in case of pool resize operations or new applications being started. The period in which the service is down can be further reduced to below 1 second by first opening the PDB and then relocating the service. While this could be a useful optimization for production systems to minimize the probability of the service being down while applications try to connect, we do not test such a scenario and therefore did not apply this optimization for our tests.

3.2 Connection Pool

We have implemented a client connection pool that allows to handle the live migration of a PDB transparently for the application. It registers itself at the

database for events through the Oracle Notification Service (ONS). When a service is stopped as part of a relocation, ONS sends out a *service down* event. Since the stopping of a service has no effect on already established connections, the client keeps using the connections in the pool just as before. Shortly after, the service will come up on the target node. The connection pool will then receive another ONS notification, a *service up* event. Only after receiving this event[1], we will now start migrating connections to the new node by disconnecting idle connections (connections that are currently not borrowed from the pool) from the source node and reconnecting them to the same service again, which is now running on the target node. During a certain period of time to which we refer as the *migration time* in this paper, the client is connected to both nodes simultaneously and accesses database blocks on both nodes. The first requests on the target node will lead to cache misses as blocks for this PDB have not yet been cached. These blocks will be fetched either from the source node or from disk. Previous work shows that typical workloads have a working set of frequently accessed data that is smaller than the overall database size [28][10]. While first requests are executed on the target node, it can quickly build up a cache of the most frequently accessed blocks without needing to load the entire database into cache. As we continue migrating connections, more and more work shifts from the source to the target node, until finally all connections have been migrated. From that point on, clients exclusively access the database on the target node.

In our tests we found that the ideal migration speed depends heavily on the workload characteristics such as access patterns, working set size, and transaction rate. If connections are migrated too quickly, the target node is not given enough time to warm up its cache, resulting in high response times for many transactions and eventually exhaustion of the number of database connections. This leads to queuing of requests on the application side waiting for connections to become available. To avoid this situation as much as possible, we found it beneficial to start with a very low migration speed. Since many workloads have a small subset of blocks such as index blocks that are frequently accessed, a few migrated connections can be sufficient to fetch these blocks from the source node without overwhelming the target node with too many requests at once. As the migration of connections continues, the initially chosen speed may be too low. At best, this only results in a longer than necessary overall migration time. For update-intensive workloads however, especially if they have a small subset of frequently updated blocks, a low migration speed can also lead to adverse effects as blocks that have already been transferred to the target node are now again being requested by the source node. In this situation, system resources can be wasted for repeated block transfers (also referred to as *block pinging*).

We have therefore implemented an algorithm that automatically adjusts the rate at which connections are being migrated to the workload and performance characteristics of the database. It attempts to migrate a tenant as quickly as

[1] If the service is taken down permanently or in case of error situations, a client is advised to stop using the service. Our prototype does not consider this situation.

possible under the constraint of affecting its quality of service (QoS), namely throughput and response times, as little as possible. When balancing these two (sometimes) conflicting goals, we value QoS over migration speed.

Migration Algorithm

To allow the implementation of different migration rates and policies as the migration is progressing, we evenly divide the migration into four stages. The end of each stage is defined by the number of connections that have already been migrated relative to the total number of connections (25%, 50%, 75%, and 100% for stages 1, 2, 3, and 4). During early stages, we use a low migration rate and the possibility of additional throttling. In later stages, we allow higher migration rates and the possibility of additional acceleration. For each stage, the algorithm computes every second a *base migration rate*, being the number of connections to migrate this second (1%, 2%, 4%, and 10% of the total number connections in stage 1, 2, 3, and 4), rounded to the next integer. Additionally, it computes average response times for all requests that were served by the source and target node in the previous second. The base migration rate may then be adjusted based on the following policies, applied in the order as described, which then determine the *actual migration rate* of how many connections to migrate each second[2].

- Throttling: In the first three stages, we throttle the migration rate if response times on the target node significantly exceed those on the source node: If response times are 2 or more times higher, we throttle the actual migration rate to 50%; if they are 3 or more times higher, we throttle to 25%. High response times on the target node are an indication for a cold cache. By throttling the migration rate, we give more time to the target node to build up its cache and not overwhelm it with too many requests. In the last stage, where more than 75% of all connections have been migrated, the risk of already transferred cache blocks being requested again by the source node increases, which is counter-productive to the migration. We therefore implement no further throttling in the last stage.
- Acceleration: If response times on the source node exceed those on the target node, we double the migration rate. Once caches on the target node have sufficiently filled and requests are running better on the target than on the source node, there is no reason to hold migration back, so accelerating it reduces overall migration time.

This algorithm has proven to reduce the impact on response times for the migrated tenant by starting migration at a low pace, throttling the connection

[2] The number of connections migrated each second is the integral component of the calculated *actual migration rate*, while the remainder of it is rolled over to the next second. For example, if a rate of 1.75 has been calculated, one connection will be migrated, and the remaining value of 0.75 will be added to the rate calculated in the next second.

migration rate even more when needed, and then accelerating as the cache on the target node is warming up (section 4.4; see figure 2 for an illustration).

While the base migration rates as well as the response time thresholds, throttling and acceleration factors could be made configurable for fine-tuning, we believe this will generally not be necessary. The base migration rate is chosen relative to the number of connections and therefore adapts to different connection pool sizes. The response time thresholds are based on relative differences between source and target node and independent of absolute response times, and the throttling and acceleration factors are reasonable adjustments to the base migration rate. By considering response time differences, the algorithm adapts not only to workload characteristics, but also performance differences between various platforms. We successfully verified this algorithm for different workloads and connection pool sizes between 10 and 200 connections.

4 Experimental Evaluation

We now evaluate the live migration of a PDB with our connection pool implementation using a variety of workloads to analyze how different attributes of a workload, such as database size, transaction rate, and access type and distribution affect migration time, impact, and cost.

4.1 Test Setup

System Configuration. We conduct this analysis on an Oracle SuperCluster T4-4 configuration using 2 T4-4 servers for the database software, and 7 X2-2 Exadata Storage Servers, connected through Infiniband fabric. Each database server has 4 SPARC T4 processors (8 cores and 64 threads each), running at 2.998 GHz, and is equipped with 512 GB of memory. The servers run Solaris 11 Update 1 with the latest Oracle 12.1.0.1 database software. As load generator, we use a server with 2 Intel Xeon X5670 processors running Oracle Enterprise Linux 6, which is connected to the database servers through 10 GbE.

Load Test Environment. For load generation and statistics collection we use *CloudPerf*, a Java-based performance test environment we have developed. It uses an open load generator [26] capable of maintaining a configurable injection rate regardless of the performance of the system under test (SUT). Requests are served by a pool of worker threads which acquire connections from a connection pool. Since every request needs exactly one database connection, we configure the worker thread pool size identical to the connection pool size, with the same minimum and maximum setting for both. Requests that wait for a worker thread will be queued. As we adequately want to mimic the perceived performance of applications, we include this queuing time in all reported response times. If a request has been queued for more than 1000 ms, we discard it and count it as a failed request to avoid infinite queuing in case the SUT does not keep up. CloudPerf also captures all relevant operating system and database statistics used in this study.

4.2 Workloads

In our evaluation, we use two different OLTP workloads which we have implemented on CloudPerf. For our in-depth analysis of the influence of workload characteristics on our migration technique we use CRUD, an internal workload that performs random insert, select, update, and delete operations on a single table. It allows us to easily change the table size and transaction mix and thus create different data access patterns. While CRUD is a well-suited workload for such analysis, it lacks the complexity of real-world workloads. As a more relevant workload that resembles typical OLTP workloads more closely, we use ODB-CL, an implementation of the Oracle Database Benchmark for CloudPerf.

CRUD. CRUD[3] is an OLTP workload we developed for database experiments to investigate certain characteristics of high-level workloads in a deterministic and controlled manner. It performs a configurable mix of random select, update, insert, and delete operations on a single table of arbitrary size, using a unique number (ID) as a primary key, and a partition key (PART) for further filtering. In a BLOB[4] field (DATA), we store an arbitrary amount of binary data. The table is partitioned based on the partition key.

Table 1. Table Sizes and Attributes (CRUD)

Rows	Partitions	Index	Data Blks	Index Blks	DB Size
1 M	128	ID	271,148	6,550	2.1 GB
10 M	512	ID	2,065,880	41,367	16.1 GB
50 M	16	ID,PART	8,472,506	361,807	67.4 GB

We conduct our experiments with table sizes of 1, 10, and 50 million rows with 1024 byte of data stored in the BLOB field using a default block size of 8192 byte, resulting in a database size of 2.1, 16.1, and 67.4 GB, respectively (table 1). The mix of queries we run consists of *select*, *update*, *insert*, and *delete* operations, which each select, update, insert, or delete 5 rows per execution. For select and update operations, the first of these rows is randomly picked using a uniform distribution across the entire range of IDs, while the remaining 4 are rows with the next-highest ID in the same partition. Both operations fetch and update the data in the BLOB field. The remaining operations insert or delete rows beyond the last provisioned row at a predetermined index that is incremented with every insert and delete.

For the first set of experiments with 1 and 10 million rows, we only use a unique index on the ID column, which forces select and update queries to scan parts of a table partition, thus accessing a large number of blocks on each execution. In these experiments, block accesses spread equally across all blocks,

[3] The name CRUD refers to Create, Read, Update, Delete (in database context Insert, Select, Update, Delete).

[4] Binary Large Object

Table 2. Block Accesses per Query (CRUD)

Rows	Select	Update	Insert	Delete
1 M	2020	2072	43	48
10 M	4079	4080	77	95
50 M	24	51	65	56

with about 99% of the blocks accessed being table blocks (tables 2 and 3). For the experiments with 50 million rows, we create an additional index on ID and PART. By doing so, we eliminate table blocks scans, reducing overall block accesses for select and update operations significantly, and shift the access distribution more towards index blocks, which now account for 56% of all block accesses (tables 2 and 3). Since the number of index blocks is just a fraction of the number of data blocks, this leads to a small set of frequently accessed blocks, while the majority of the blocks is less frequently accessed, a pattern more typical for many real workloads [28][10]. With respect to block modifications, the update operations only modify data blocks, which again spread equally across the entire table. The insert and delete operations however need to maintain the index as well, reflected in modifications of index blocks. Since there are much fewer index than data blocks, index modifications can lead to concurrent updates of index blocks (especially root index blocks), which might need to be repeatedly transferred between nodes if accessed on both nodes simultaneously. We use this to include effects arising from increased update concurrency in our analysis. For each of the three aforementioned table sizes, we run a set of four experiments, varying the transaction mix between 100% select, 80% select and 20% update, 20% select and 80% update, and 20% select, 40% update, 20% insert, 20% delete.

Table 3. Block Access by Table and Index (CRUD)

Rows	Statement Mix	Reads (Tbl/Idx)	Updates (Tbl/Idx)
1M, 10M	Select only	99% / 1%	none
1M, 10M	Select/Update	99% / 1%	100% / 0%
1M	S/U/I/D	98% / 2%	65% / 35%
10M	S/U/I/D	99% / 1%	23% / 77%
50M	S/U/I/D	44% / 56%	64% / 36%

ODB-CL. ODB-CL is an implementation of the Oracle Database Benchmark (ODB) [10] for CloudPerf. Its data model consists of 9 tables, accessed by 5 transactions, which portray the order management of a wholesale supplier with a number of warehouses organized in districts[5]. We report throughput as trans-

[5] While ODB-CL has similarities with the industry-standard TPC-C Benchmark [29], it is not a compliant TPC-C implementation. Any results presented here should not be interpreted as or compared to any published TPC-C Benchmark results. TPC-C Benchmark is a trademark of Transaction Processing Performance Council (TPC).

actions per second (tps), being the total number of executed transactions of any of the five types.

4.3 Methodology

In an initial series of experiments, we migrate a single database running at a steady load from one node to another, using different workloads and workload parameters. For these experiments, this database is the only active database, that is the target node is idle before the migration, and the source node is idle after the migration. After analyzing the results obtained from these experiments, we then migrate a tenant's database running concurrently with other tenants from a node under load to another node under equal load.

Each experiment consists of a warmup phase long enough to bring the workload into a steady state, followed by three phases of five minutes each: steady state before migration, migration, and steady state after migration (figure 1).

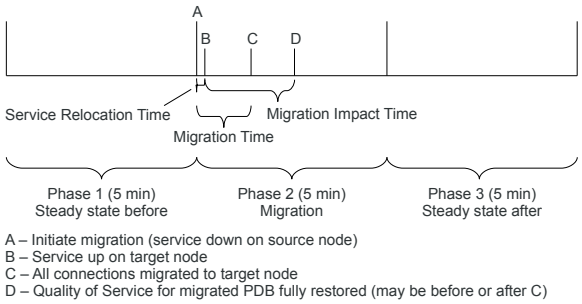

Fig. 1. Overview of Migration Phases

For single-tenant migration tests, the container database cache size has been configured to 32 GB (unless otherwise noted), which is sufficiently large to cache the tenant's entire database in memory. In the test with multiple active tenants, we use a cache size of 320 GB.

Migration Phases

Phase 1 - Steady State before Migration. In the first phase, we collect data during steady state before the migration. This data is used as a reference point to later determine migration overhead by comparing performance metrics during migration against the metrics collected in this phase.

Phase 2 - Migration. During the second phase, we migrate the database from the source to the target node. This phase begins with initiating the migration by failing over the service to the target node, which includes opening of the PDB on that node. After the service is back up, we begin migrating connections until all connections have been reconnected to the target node. Once the last

connection has been migrated, we consider the migration as completed and note this time as the *migration time*. Since the target node fetches database blocks only on demand, either from the other node or from disk, the transfer of blocks to the target node may continue after migrating the last connection. The tenant might therefore still face some degradation in quality of service if the cache on the target node has not yet fully been build-up. We measure the time during which a tenant's response times are affected, beginning with migrating the first connection, as the *migration impact time*.

Phase 3 - Steady State after Migration. The last phase of 5 minutes only serves as a verification of whether throughput and response times as well as CPU utilization match those of phase 1 again, where the expectation for CPU utilization is that the target node is now running at the same CPU utilization as the source node before the migration, and vice versa.

Metrics

Common metrics used by researchers to evaluate live migration performance are downtime, total migration time, amount of migrated data, and migration overhead [13]. Since by design our technique does not impose any downtime, we omit this metric. In the beginning of the migration phase, the service is relocated, and the database is opened on the target node. While we measure the service relocation time (few seconds in our tests), we do not explicitly report it in this paper as it does not affect the workload, but include this time in the reported migration time.

With our solution, the *total migration time* is difficult to determine as there is no clearly defined end of the migration. Therefore we note the time from initiating the service relocation until the last connection has been migrated as *migration time*. As a second metric, we calculate the time from migrating the first connection until the quality of service of the migrated tenant has been fully reestablished, which we define as average response times being within 10% of those during steady state before migration, and request failure rate being zero. We refer to this time as the *migration impact time*. Requests that cannot be handled by the database immediately (because all connections in the pool are in use) will be queued for up to 1 second in the load generator. After 1 second, they will be discarded and counted as failed requests. For a failure rate of 0, throughput is identical to the injection rate. Instead of throughput of successful transactions, we report injection rate and the number of failed requests, and in case of failures also the failure rate (based on overall requests in phase 2).

We measure average response times across all database transactions in the steady-state phase before migration as well as during migration, and report latter ones for the period of time in which a tenant is impacted (*migration impact time*). These response times include any potential queuing time in the load generator.

Due to the nature of our migration technique, we split the amount of transferred data during migration into two categories: data transferred from the source to the target node, and data transferred in the opposite direction. The latter can

occur if blocks modified on the target node during migration are again requested by transactions still running on the source node.

We calculate CPU cost of migration as the ratio of aggregate CPU utilization from source and target node during the migration phase (phase 2) divided by the aggregate CPU utilization during the steady-state phase before migration. A cost of 1.1 would mean that the combined CPU utilization of both nodes was 10% higher during migration than during steady-state.

For each experiment, we verify that response times and CPU utilization during steady-state before and after migration are within 10% (with CPU utilization on both hosts interchanged), and that not a single request has failed in any of the three phases. Only then we call an experiment *successful*. Otherwise we consider it *failed*.

For each experiment, we capture and report the following metrics:

- migration time (*Migr Tm*)
- migration impact time (*Impact Tm*)
- number of failed transactions for migrated tenant (*Failed TX*))
- response times for migrated tenant during steady-state before migration (*Resp Steady*) and during the migration impact time (*Rsp Migr*)
- amount of data transferred from source to target node (*Data Rcvd*)
- amount of data transferred from target back to source node (*Data Sent*)
- CPU cost of migration (*CPU Cost*)
- test result (*Result*, *successful* (unless otherwise stated) or *failed*)

4.4 Experiments and Analysis

CRUD (Connection Migration)

In the first series of experiments (table 4), we evaluate the behavior of different connection migration algorithms by migrating a single tenant running the CRUD workload on a table with 10 million rows at a rate of 2500 transactions per second (80% select, 20% update), using a connection pool size of 200.

Table 4. CRUD 10M rows (16.1 GB), 200 conn, IR=2500, S/U/I/D Ratio 80/20/0/0

Conn Pool	Conn Migr Rate	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Result
UCP	abrupt/max	17.0 s	38 s	62,006 (8.3%)	24.2 ms	1932.0 ms	failed
CloudPerf	10 Conn/s	30.8 s	41 s	15,113 (2.0%)	24.3 ms	369.2 ms	failed
CloudPerf	5 Conn/s	47.6 s	37 s	0	24.3 ms	49.6 ms	successful
CloudPerf	3 Conn/s	73.6 s	31 s	0	24.3 ms	41.5 ms	successful
CloudPerf	adaptive	62.5 s	38 s	0	24.2 ms	32.4 ms	successful

As a baseline, we compare against the Oracle Universal Connection Pool (UCP) version 12.1.0.1, which in this version[6] immediately after receiving the

[6] Based on our work, UCP version 12.1.0.2 will implement a similar connection migration as presented in this paper, including a timeout in case the service is taken down permanently.

service down event terminates all connections that are returned to the pool, and begins establishing new connections after the service is back up at the fastest possible rate until the configured minimum pool size has been reached again. Table 4 shows that this leads to a high rate of failed transactions (8.3% of all transactions in phase 2) and average response times (including queuing) of almost 2 seconds over a period of 38 seconds, caused by a combination of a short time during which the service is down on both nodes, and a high rate of requests hitting the target node and its empty cache all at once as soon as service is resumed. Note that the failed transactions are exclusively a result of queuing exceeding 1 second; the database itself does not abort or fail any transactions.

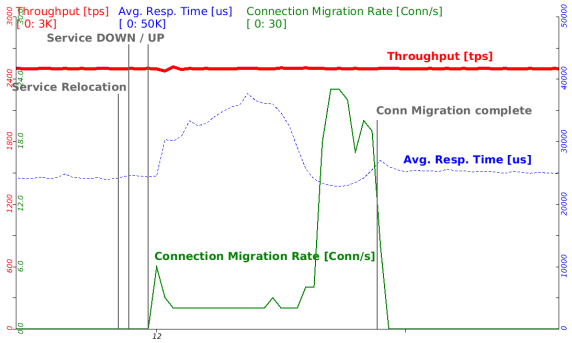

Fig. 2. CRUD 10M rows, 200 conn, IR=2500, S/U=80/20: Connection Migration Rate

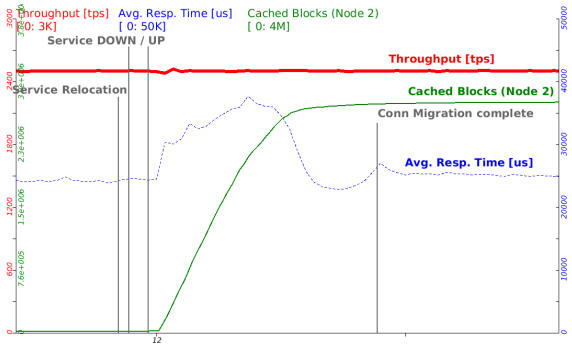

Fig. 3. CRUD 10M rows, 200 conn, IR=2500, S/U=80/20: Cached Blocks

Our connection pool prototype for CloudPerf only starts migrating connections after the service has come back up, therefore completely avoiding any downtime. With a fixed connection migration rate of 10 connections per second, we still see requests failing because of queuing as the target node does not keep up servicing requests in a timely manner due to a cold cache, even though the failure rate has reduced to 2.0%. In order to give the target node enough time to warmup its cache, connections need to be migrated at an even lower rate, such

as 5 or 3 connections per second. The difficulty with a fixed rate however is to find the right balance between migration time and migration impact.

Our connection migration algorithm described in section 3.2 minimizes response times during migration and achieves a similar migration time without the need of manual tuning (table 4). Figure 2 shows how our algorithm starts migrating connections at a very low rate, giving the target node enough time to fetch most frequently accessed blocks (figure 3), and then accelerates the rate once response times on the target node stabilize again. In the beginning of the migration, the source node has to serve cache blocks for the target node, increasing its CPU utilization temporarily (figure 4), which then drops again as more and more work is transferred to the target node.

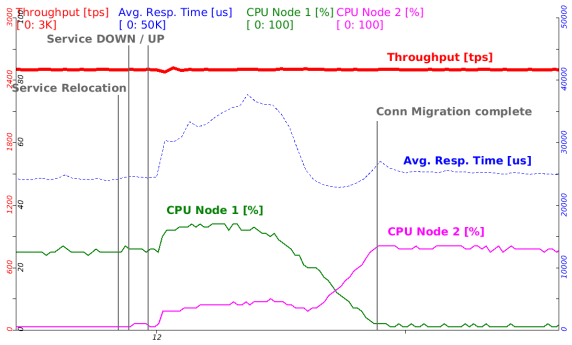

Fig. 4. CRUD 10M rows, 200 conn, IR=2500, S/U=80/20: CPU Utilization

CRUD (Single Tenant, fully cached)

For the next series of experiments, we again migrate a single database tenant running the CRUD workload from one node to another, while no other tenants are active. The tenant's database is fully cached in the source node before migration. We analyze the impact of different database sizes, transaction rates, and transaction mixes on live migration performance, using our connection pool prototype with adaptive tuning of the connection migration rate. The results are shown in tables 5, 6, and 7.

Table 5. CRUD 1M rows (2.1 GB), 30 connections, IR = 1000 tps

S/U/I/D Ratio	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data Sent	CPU Cost
100/0/0/0	24.5 s	11 s	0	7.7 ms	9.1 ms	2.0 GB	0.0 GB	1.01
80/20/0/0	40.4 s	56 s	0	9.0 ms	11.9 ms	4.1 GB	0.1 GB	1.19
20/80/0/0	42.2 s	91 s	0	11.9 ms	16.3 ms	6.1 GB	0.4 GB	1.13
20/40/20/20	42.8 s	123 s	0	7.5 ms	10.7 ms	5.8 GB	0.6 GB	1.19

The transaction rate has only a small influence on migration time and impact. Since a higher transaction rate requires a larger number of connections to sustain the traffic, our algorithm migrates connections at a rate proportional to the pool

size, and throttles if necessary when transactions on the target node face too high response times. For this reason, migration times increase a bit with higher transaction rate, but the migration impact time decreases as higher throughput (given a fixed database size) reduced the time until all blocks have been requested on the target node at least once.

Table 6. CRUD 1M rows (2.1 GB), 100 connections, IR = 2500 tps

S/U/I/D Ratio	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data Sent	CPU Cost
100/0/0/0	29.1 s	10 s	0	10.9 ms	11.9 ms	2.1 GB	0.0 GB	0.98
80/20/0/0	43.6 s	38 s	0	14.0 ms	17.3 ms	6.3 GB	0.4 GB	1.11
20/80/0/0	54.3 s	73 s	0	15.8 ms	23.6 ms	10.6 GB	1.9 GB	1.16
20/40/20/20	54.2 s	119 s	0	9.4 ms	15.5 ms	10.2 GB	4.1 GB	1.36

While it is not a surprise that a larger (cached) database increases migration time, it is worth noting that the additional time needed is underproportional to the database size: Eight times more cached data can be migrated in just about twice the time, while the time the tenant is impacted is even less than double. The reason lies in the self-adapting migration algorithm which imposes an upper bound on migration time, provided network bandwidth is sufficient.

Table 7. CRUD 10M rows (16.1 GB), 200 connections, IR = 2500 tps

S/U/I/D Ratio	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data Sent	CPU Cost
100/0/0/0	45.3 s	21 s	0	21.7 ms	26.2 ms	15.8 GB	0.0 GB	1.02
80/20/0/0	62.5 s	38 s	0	24.2 ms	32.4 ms	21.1 GB	0.3 GB	1.10
20/80/0/0	86.0 s	153 s	0	30.3 ms	40.1 ms	33.7 GB	1.3 GB	1.28
20/40/20/20	87.7 s	104 s	0	16.8 ms	29.5 ms	33.6 GB	4.0 GB	1.55

The workload mix however has a dominating effect on both migration time and cost. In a read-only workload, each block only has to be fetched from the source node at most once, limiting the amount of data to be transferred to the amount of cached blocks on the source node. For these experiments, the database was fully cached in memory, and the amount of data transferred from the source to the target node matches the database size. The CPU cost is negligible[7], and response times during migration are only 10-20% higher than at steady state. Even migrating a 16 GB fully-cached database impacts the tenant's response times for just 21 seconds. When transactions modify blocks, Oracle clones them to provide consistent read (CR) for concurrent sessions. The cache on the source node before migration consists of both regular data blocks as well as co-called *CR blocks*, which during migration may both be transferred to the target node. Additionally, transactions executed on the source node during migration may request some already transferred blocks, visible in the results as data sent back to the source node, which might then be requested again at a later point. Hot

[7] For the test of 1 million rows and 2500 tps, the average CPU utilization across both nodes is even *lower* then during steady state, caused by better hardware efficiency (reduced cache misses in CPU caches) as both nodes share traffic.

blocks, such as frequently updated index blocks, contribute most to this scenario and may ping back and forth between nodes multiple times: They contribute 30% to the blocks being sent back to the source node in the tests with insert and delete operations. Those tests have a 2-3 times higher number of blocks pinging than tests with a high update rate on data blocks only, even though index blocks in this workload make up only for 2% of overall blocks. As a consequence, also response times during migration are up to 75% higher than at steady state when concurrent updates on a small number of blocks increase.

CRUD (Single Tenant, partially cached)

Typical workloads do not access all content of a database with equal probability, but rather have a small set of frequently needed blocks, allowing databases to be significantly larger than the cache, which attempts to cache hot blocks while purging less frequently used ones. To analyze the effects partially cached workloads have on our migration technique, we grow the database to 50 million rows (67 GB) and create an additional index that eliminates the scans of table blocks (table 8). In this configuration, index blocks make up for 4% of the total blocks (2.8 GB), but are subject to 56% of all read accesses and 36% of all updates. With 64 GB of cache, the database is (nearly) fully cached. While the migration itself is fast (47 seconds), the time during which the tenant is impacted by at least 10% higher response times lasts for 327 seconds. As a result of eliminating the partial table scan, it now takes much longer for the workload to access all data blocks at least once, so only index blocks are transferred quickly. Once we shrink the cache and the database is not fully cached any more, the impact time drops to 43 - 49 seconds as the tenant's queries face a higher cache miss rate even in steady state. During migration, after the frequently accessed index blocks have been cached in the target node, the tenant quickly reaches similar response times as before the migration. As we shrink the cache, more and more blocks which the target node needs to read into its cache are read from disk rather than fetched from the source node[8]. From this experiments we conclude that our migration technique is friendly to partially cached workloads and benefits from them as it only transfers frequently accessed cache blocks between nodes and is independent of the database size on disk.

Table 8. CRUD 50M rows (67.4 GB), 50 conn, IR=2500, S/U/I/D Ratio 60/20/10/10

DB Cache	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd Rmt	Data Read Dsk
64G	47.4 s	327 s	0	2.6 ms	3.1 ms	46%	54%
32G	40.6 s	49 s	0	3.1 ms	4.2 ms	29%	71%
16G	43.3 s	43 s	0	3.2 ms	4.3 ms	25%	75%
8G	39.8 s	49 s	0	3.2 ms	4.5 ms	20%	80%

[8] *Dynamic remastering* changes block mastership to the node where blocks are most frequently accessed. Once blocks are mastered on the target node, it may prefer to read them from disk rather than remote cache, resulting in some disk reads even for fully-cached databases.

ODB-CL (Single Tenant)

After analyzing the characteristics of our migration technique using a simple workload such as CRUD, we apply it to a more complex database workload. In these experiments, we migrate a single database tenant running the ODB-CL workload using different number of warehouses and transaction rates as described in section 4.2 from one node to another. The results are shown in table 9.

Table 9. ODB-CL 10, 500 and 1000 warehouses (1, 50 and 100 GB)

# WH	IR	Conn	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data Sent	CPU Cost
10	100 tps	10	22.2 s	147 s	0	4.8 ms	7.2 ms	0.9 GB	0.0 GB	1.06
500	1000 tps	25	54.1 s	287 s	0	6.0 ms	9.0 ms	19.1 GB	0.4 GB	1.40
1000	1000 tps	25	54.1 s	302 s	0	7.5 ms	11.6 ms	21.4 GB	0.3 GB	1.45
1000	2500 tps	100	72.2 s	166 s	0	7.6 ms	13.9 ms	31.5 GB	1.8 GB	1.53

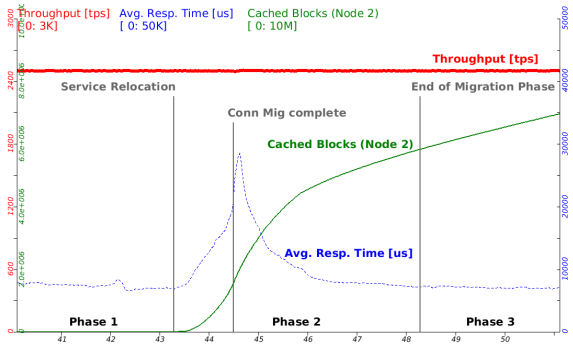

Fig. 5. ODB-CL (1000 warehouses, 100 connections, IR = 2500 tps): Cached Blocks

In contrast to CRUD, the data access in ODB-CL spreads across multiple tables and indexes, with none of them contributing more than 5% (a very small table of only 8 MB size) to the overall accesses. The four tables which together account for 50% of all block changes also consume half the database size. With this, the time during which the migrated tenant is impacted by higher response times is fairly large with values between 2.5 and 5 minutes. While this may seem long, it is caused by a very long tail of slightly increased response times as the cache on the target node is slowly being filled (figure 5). The total amount of data transferred from source to target node, with only about 20 GB transferred data for a 50 GB database, and 32 GB transferred data for a 100 GB database, is a factor 2-3 smaller than the database size. While this is partially due to the fact that we configured only a cache of 32 GB, a tenant in a consolidated environment will also not get the entire cache for itself, as the next test will show. With a cache limitation of 32 GB, we are trying to simulate limited cache resources even for this single-tenant experiment, which still allowed the tenant to achieve a cache hit rate of about 98%. The percentage of the data that is transferred back

to the source node, mostly data blocks from the stock and warehouse tables[9], is with 2-4% of the total amount of transferred data much smaller than in some of the previous experiments with the CRUD workload. While the CRUD workload allowed us to study the dependencies of our algorithm, the more realistic ODB-CL workload behaves more balanced.

ODB-CL (Multiple Tenants)

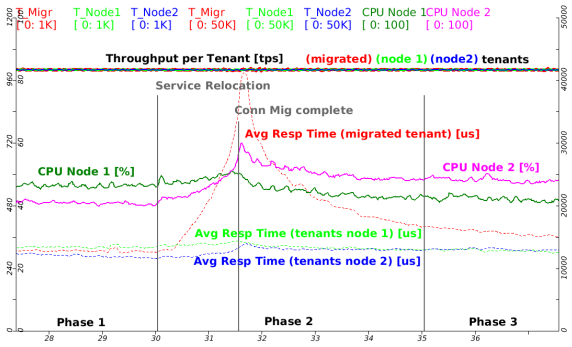

Fig. 6. ODB-CL - 33 active tenants (IR=1000); migrated tenant: 500 WH, 50 conn

For the last experiment, we deploy a total of 33 tenants: 16 tenants on node 1 and 16 tenants on node 2, issuing 1000 transactions per second against a ODB-CL database of 500-1000 warehouses each, and one tenant that is being migrated from node 1 to node 2. The migrated tenant's database comprises 500 warehouses, and is being accessed at the same rate of 1000 transactions per second, using a connection pool of 50 connections. The result of this test is shown in table 10 as well as figure 6. The database cache has a size 320 GB per node. As our tenant has to compete for cache with the other tenants, the amount of its data cached in the source node is 23.8 GB before migration, similar to previous tests where we artificially limited the database cache size. While the database servers are running at around 50% CPU utilization, migrating this tenant takes 91 seconds. Its response times, on average 50% higher than before migration, are affected for a duration of 370 seconds, slowly approaching a steady level, with a long tail of slightly elevated response times. For the database size of 50 GB, only 18.5 GB had to be transferred to the target node, which caused CPU utilization to increase by only 7% during the migration phase. This increase is lower than in earlier experiments where the migrated tenant was running in isolation; while the absolute CPU cost is similar, the relative cost on a heavily utilized system with many tenants becomes marginal.

As for all other experiments with our connection pool and migration algorithm, this tenant faced no downtime, and not a single transaction was aborted.

[9] Both tables are frequently updated. The warehouse table is a small table with high concurrency.

The effect on other tenants is negligible: Their response times increase no more than 4% during a short period in which CPU utilization increases as blocks are being transferred. After the migration, tenants on node 2 face slightly higher response times (13 ms) than before migration (12 ms) as overall load on node 2 has increased, while response times for tenants on node 1 have dropped from 13 ms to 12 ms on the now lower utilized node.

Table 10. ODB-CL - 33 active tenants (IR=1000); mig. tenant: 500 WH , 50 conn

Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Cached before Migr	Data Rcvd	Data Sent	CPU Cost
91.1 s	370 s	0	12.8 ms	19.8 ms	23.8 GB	18.5 GB	0.7 GB	1.07

Summary

The experiments demonstrate the scalability of our migration technique to large databases of even 100 GB size and transaction rates of 2500 tps, proving the feasibility of this approach also in consolidated environments with many active tenants. Our self-adapting algorithm successfully controls the connection migration rate, limiting the effect on the migrated tenant by automatically throttling or accelerating migration speed as needed. In all experiments, the tenant was migrated with not a single failed request and no downtime. Migration took between 24 and 91 seconds and increased response times for the migrated tenant by 20-50% for most experiments, with only a few tests in which response times almost doubled.

With our migration technique, transactions will never fail at the database layer. The only possible cause for failed requests in our experiment are due to queuing times in the load generator exceeding 1 second when injection rate exceeds processing rate. In order to maintain throughput when response times increase, a sufficient number of connections is needed. For our experiments, we used a connection pool size about twice as high as the connection demand during steady state. Based on our experience with production systems, such a pool size is typical for many OLTP workloads, which have to accommodate peaks in response time also in situations other than live migration. While we would typically not advise the use of dynamic connection pools with the maximum pool size being set higher than the minimum size, a temporary increase of the pool size during migration could be a worthwhile extension of our technique.

5 Further Considerations

5.1 Provider's View

Our migration technique is implemented inside a client connection pool, which reacts to service relocation events it receives from the database. For environments in which the database service provider also controls the applications, this might be sufficient. However, to protect against misbehaving clients, especially when

they are external, the provider should close the PDB after a certain time on the source node, which then disconnects all clients from that node and flushes remaining cache contents to disk. By doing so, further access to the database on this node is prevented, both for clients as well as other nodes in the cluster. For experiments not quoted in this paper, we have explicitly closed the PDB on the source node 120 seconds after relocating the service. This shall give clients sufficient time to gradually migrate their connections, while at the same time providing a guarantee to the provider that no further access will happen on the source node after 2 minutes. A client that did not act accordingly upon receiving the relocation notifications will then face errors when attempting to access the database. Therefore it is in the sole interest of the client to comply and migrate its connections in time.

5.2 Server-Side Migration Control

As an extension of our idea, migration control could be implemented in the server by explicitly disconnecting (idle) client connections one by one on the server side[10]. The connection pool could then, transparent for the application, reestablish them to the target node without failure of any transactions. If transactions were interrupted when a connection was terminated, features like *Application Continuity* [20] could transparently replay these transactions on the target node. A similar algorithm as presented in this paper to determine the rate at which to close connections could then be implemented in the server. Further enhancements could potentially eliminate any dependency on a client's connection pool if database connections including their TCP socket could be relocated to another host [4] without the need of reestablishing them. We leave the investigation of these possibilities for future work.

5.3 Long-Running Transactions

Our prototype only migrates connections that are currently not in use by a client, which avoids migration of connections that are within a transaction. This implies that workloads with long-running transactions will only migrate their connections after transactions have completed, causing potentially longer migration times. With the enhancements described in section 5.1, the provider can limit the maximum migration time regardless of transaction duration.

5.4 Other Workloads

Other workloads such as Decision Support Systems (DSS) and batch workloads often do not use connection pools but establish connections on demand. After the service has been relocated, their next request will be directed to the new node automatically. The previous section also applies to their long-running transactions.

[10] Oracle 12.1.0.2 will implement a *pluggable database relocate* command to accomplish this.

5.5 Failure Scenarios

Traditional VM live migration only addresses *planned* migrations and does not help in supporting unplanned outages where VMs fail unexpectedly. For high availability of virtualized database deployments, alternative solutions have been proposed [23]. Our technique is based on Oracle RAC which can handle both planned and unplanned events. While the migration of databases in case of a node failure will not be as seamless as in a controlled migration, services will fail-over and connections will be reestablished in a similar way. The presented technology can therefore cover both planned migration as well as failure scenarios and provide seamless live migration and high availability at the same time.

It is worth noting that our technique is therefore also robust against node failures *during* migration. A failure of the source node during migration will terminate connections to this node, which will then be reestablished to the target node. The migration therefore continues. A failure of the target node will cause the service to fail back to the source node (or another node in the cluster) and essentially abort or revert the migration. In both cases, no data is lost, and operation resumes after a short cluster reconfiguration phase.

6 Related Work

Live migration has become a popular technique, but few studies focus on the live migration of databases. For virtual machines in general, Hu et al. [12] quantify migration time using different hypervisors and demonstrate how both memory size as well as memory dirtying rate affect migration time. Their study shows that the amount of data to be transferred can in some cases exceed the VM size by a factor of 2 if pages are repeatedly updated, similar to results in [13]. Liu et al. [18] have migrated a database in a 1 GB VM running TPC-C in less than 20 seconds at a downtime of 25 ms using Xen.

Our migration technique is based on the idea of migrating a tenant's database connections and transferring database content rather than VM pages. A similar approach for shared-disk databases, named *Albatross*, has been proposed by Das et al. [7]. By copying database cache using an iterative pre-copy technique, they were able to migrate databases with downtimes as low as 300 ms. For a 1 GB TPC-C database, the downtime increased to slightly below 1 second with more requests failing when the transaction rate was increased to 2500 tpmC (equivalent to about 93 tps[11]). For shared nothing architectures, Elmore et al. [8] presented *Zephyr*, which migrates a database without downtime by copying database content using a combination of an on-demand pull phase similar to ours, followed by a push phase. In contrast to our implementation, Zephyr redirects requests to the target node abruptly, which requires frequently accessed data to be transferred quickly during the pull-phase. Using YCSB [6] as a workload, which like CRUD performs a mix of read, update, and insert operations on a table, they observe a 10-20% increase in query response times and some

[11] Based on a 45% share of NEWORDER transactions as quoted in the paper.

aborted transactions due to index modifications during migration. Both Albatross and Zephyr use a query router to direct traffic to the correct database node. Our technique is different as clients connect directly to the database and the migration of connections is handled in the client's connection pool rather than a router. *Slacker*, a database migration system presented by Barker et al. [2], migrates database content by taking an initial database snapshot followed by streaming of change records to the target database, and achieves downtimes of less than 1 second with response times increasing from 79 to 153 ms during migration for a 1 GB YCSB database.

7 Conclusions

Providing on-demand database service requires database consolidation to be elastic and scalable, while at the same time achieving a high density through resource sharing between tenants. In such a multitenant database environment, an efficient method to seamlessly migrate tenants from one set of physical resources to another is a crucial component to support dynamic changes in demand and implement load balancing. We have presented a technique that allows to migrate a tenant's database by only transferring database cache. Our prototype connection pool implements an algorithm to automatically adapt migration speed to workload and system behavior in order to minimize impact on the migrated tenant while keeping overall migration time low. It is completely transparent to the application and requires no modifications on the application side. To demonstrate the scalability and feasibility for real-world workloads, we evaluated our technique at much larger scale than other researchers with per-tenant database sizes of up to 100 GB and transaction rates up to 2500 tps. In a detailed analysis, we characterized the performance of our solution depending on various workload parameters and verified that also in an environment under load, with 33 tenants executing queries at a rate of 33,000 tps, our technique allows the migration of a tenant with no downtime, not a single failed transaction, and only a moderate increase of response times.

References

1. P. Barham, B. Dragovic, K. Fraser, S. Hand, et al. Xen and the art of virtualization. *ACM SIGOPS*, 37(5):164–177, 2003.
2. S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, et al. Cut me some slack: Latency-aware live migration for databases. In *EDBT*, pages 432–443. ACM, 2012.
3. D. Breitgand, G. Kutiel, and D. Raz. Cost-aware live migration of services in the cloud. In *SYSTOR*, 2010.
4. H. Chu, S. Kurakake, and Y. Song. Communication socket migration among different devices, 2001.
5. C. Clark, K. Fraser, S. Hand, J. G. Hansen, et al. Live migration of virtual machines. In *NSDI*, pages 273–286, 2005.
6. B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, et al. Benchmarking cloud serving systems with YCSB. In *ACM CLOUD*, pages 143–154. ACM, 2010.

7. S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.
8. A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *ACM SIGMOD*, pages 301–312. ACM, 2011.
9. J. Gelhausen. Oracle Database 12c product family. Oracle White Paper, 2013.
10. R. Hankins, T. Diep, M. Annavaram, B. Hirano, et al. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *IEEE/ACM MICRO*, page 151. IEEE Computer Society, 2003.
11. M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *ACM SIGOPS*, pages 51–60, 2009.
12. W. Hu, A. Hicks, L. Zhang, E. M. Dow, et al. A quantitative study of virtual machine live migration. In *ACM CLOUD*, page 11. ACM, 2013.
13. D. Huang, D. Ye, Q. He, J. Chen, et al. Virt-LM: A benchmark for live migration of virtual machine. In *ACM SIGSOFT*, volume 36, pages 307–316, 2011.
14. D. Jacobs, S. Aulbach, et al. Ruminations on multi-tenant databases. In *BTW*, volume 103, pages 514–521, 2007.
15. X. Jiang, F. Yan, and K. Ye. Performance influence of live migration on multi-tier workloads in virtualization environments. In *IARIA CLOUD*, pages 72–81, 2012.
16. A. Kivity, Y. Kamay, D. Laor, U. Lublin, et al. kvm: the Linux virtual machine monitor. In *Linux Symposium*, volume 1, pages 225–230, 2007.
17. T. Lahiri, V. Srihari, W. Chan, N. Macnaughton, et al. Cache fusion: Extending shared-disk clusters with shared caches. In *VLDB*, volume 1, pages 683–686, 2001.
18. H. Liu, H. Jin, C.-Z. Xu, and X. Liao. Performance and energy modeling for live migration of virtual machines. *Cluster computing*, 16(2):249–264, 2013.
19. B. Llewellyn. Oracle Multitenant. Oracle White Paper, 2013.
20. K. Mensah. Oracle Database 12c Application Continuity for Java. Oracle White Paper, 2013.
21. M. Michalewicz. Oracle Real Application Clusters (RAC). Oracle White Paper, 2013.
22. Microsoft. Server virtualization: Windows Server 2012, 2012.
23. U. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, et al. Remusdb: Transparent high availability for database systems. *PVLDB*, 22(1):29–45, 2013.
24. M. Nelson, B.-H. Lim, G. Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX*, pages 391–394, 2005.
25. Oracle. Best practices for building a virtualized SPARC computing environment. Oracle White Paper, 2012.
26. B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, volume 6, pages 18–18, 2006.
27. Y. Shen and N. Michael. Oracle Multitenant on SuperCluster T5-8: Scalability study. Oracle White Paper, 2014.
28. R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*, page 7. ACM, 2013.
29. The Transaction Processing Performance Council. TPC-C benchmark revision 5.11, 2010.
30. C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS*, 36(SI):181–194, 2002.