Leveraging Scale-Up Machines for Swift DBMS Replication on IaaS Platforms Using BalenaDB

Kaiho FUKUCHI†, Nonmember and Hiroshi YAMADA†a, Member

SUMMARY In infrastructure-as-a-service platforms, cloud users can adjust their database (DB) service scale to dynamic workloads by changing the number of virtual machines running a DB management system (DBMS), called DBMS instances. Replicating a DBMS instance is a non-trivial task since DBMS replication is time-consuming due to the trend that cloud vendors offer high-spec DBMS instances. This paper presents BalenaDB, which performs urgent DBMS replication for handling sudden workload increases. Unlike conventional replication schemes that implicitly assume DBMS replicas are generated on remote machines, BalenaDB generates a warmed-up DBMS replica on an instance running on the local machine where the master DBMS instance runs, by leveraging the master DBMS resources. We prototyped BalenaDB on MySQL 5.6.21, Linux 3.17.2, and Xen 4.4.1. The experimental results show that the time for generating the warmed-up DBMS replica instance on BalenaDB is up to 30× shorter than an existing DBMS instance replication scheme, achieving significantly efficient memory utilization.

key words: system virtualization, IaaS clouds, operating systems, databases

1. Introduction

Infrastructure-as-a-service (IaaS) platforms, such as Amazon EC2 [1] and Google Compute Engine [2], enable cloud users to dynamically change their database (DB) service scale as needed. To handle fluctuated workloads, the cloud users adjust the number of active virtual machines (VMs, or instances), on each of which a DB management system (DBMS) is running. We refer to such instances as DBMS instances. Cloud vendors typically offer supports to facilitate DBMS instance management. For example, Amazon RDS provides well-configured DBMS instances where a popular DBMS such as MySQL runs and generates DBMS instances with a read-only DB replicated from the master DBMS to handle read-heavy requests OLAP workloads.

DBMS replication is essential in increasing DBMS instances. Modern DBMSs have a replication mechanism that maintains the data consistency between the master and replicas. Creating another DBMS instance is typically achieved by launching a new instance from an uploaded image and then replicating a DBMS on the instance using a DBMS-level mechanism. For example, the MySQL replication mechanism takes a snapshot of the current DB states, transfers it to the new instance, and periodically sends write logs to propagate updates to the replica. A replica-side mechanism restores the DB from the received snapshot and applies updates to the DB using the write logs from the master.

However, replicating a DBMS is so time-consuming that the DB service disruption can occur without careful administration. The DBMS replication involves two non-trivial tasks; DB copy and buffer-pool warming. In the DB copy phase, a DBMS copies the DB snapshot on the target new-launched instance whose size is tens of GBs. Until the DB copying completes, the write requests are blocked to keep consistency among the DBs. In the buffer-pool warming phase, the throughput of the replica DBMS is low due to frequent buffer-pool misses. The buffer-pool misses involve disk accesses that are much slower than memory ones.

Time-consuming DBMS replication forces cloud administrators and/or users to precisely provision DBMS replica instances for the stable DB services even under bursty requests [3]. The under-provisioning of the DBMS instances leads to DB service disruption since cloud users cannot prepare additional DBMS instances soon. On the other hand, over-provisioning affects the inefficient utilization of resources. Cloud users have to pay extra administration costs; otherwise cloud vendors fail to high service consolidation.

BalenaDB, presented in this paper, allows us to perform urgent DBMS replication. Unlike conventional DBMS replication schemes that implicitly assume DBMS replicas are generated on remote hosts, BalenaDB swiftly generates a warmed-up DBMS replica in a newly launched instance on the local host where the master DBMS instance is running. Specifically, BalenaDB enforces a new replica to share the master DBMS buffer-pool and DB files while maintaining resource isolation between the DBMS instances. BalenaDB and the regular replication are complementary to each other for load balancing. BalenaDB creates warmed-up replicas more swiftly than the regular replication. However, it creates them on the only local host where the master DBMS is running, while the regular replication generates replicas on remote hosts. BalenaDB relies on the fact that scale-up machines are prevalent [4], [5] and the resource utilization is low in modern datacenters [6]–[8]. If the local host is underutilized, we can perform BalenaDB’s replication. Otherwise, the regular replication is better to be done to replicate DBMS instances on remote hosts. If we always use BalenaDB’s replication, the resource reservation for replicas on the local host is required.

This paper makes the following contributions:

Manuscript received April 28, 2021.
Manuscript revised September 7, 2021.
Manuscript publicized October 1, 2021.
†The authors are with Tokyo University of Agriculture and Technology, Koganei-shi, 184–8588 Japan.
a) E-mail: hiroshiy@cc.tuat.ac.jp
DOI: 10.1587/transinf.2020ZDP7505

Copyright © 2022 The Institute of Electronics, Information and Communication Engineers
We present BalenaDB, which offers an alternative form of DBMS replication. BalenaDB can be characterized as follows. First, BalenaDB is independent of the target DBMS internal functions, which means we can apply BalenaDB to a DBMS simply by hooking read and write operations to its DB file. Second, BalenaDB swiftly starts a replica transparently to the DBMS-level mechanism for data consistency. Third, BalenaDB allows us to replicate a DBMS on various sizes of instances while efficiently handling the coming workload. Finally, BalenaDB maintains isolation between DBMS instances that share memory pages with each other (Sects. 2 and 3).

To realize BalenaDB, we introduce two mechanisms: enlightened DBMS page sharing and lazy DB copying. Enlightened DBMS page sharing shares the memory page when a DBMS replica accesses the DB data that is already cached in the master. Lazy DB copying, inspired by DB live migration techniques [9], [10], enables us to start a DBMS replica in the newly launched instance without having to complete DB file copying (Sect. 4).

We implement a prototype of BalenaDB on MySQL 5.6.21, Linux 3.17.2, and Xen 4.4.1, discuss it, and conduct experiments to evaluate it. The experimental results show that BalenaDB successfully achieves up to 30x faster warmed-up DBMS replication than the default DBMS instance replication, delivering more efficient memory utilization significantly. The results also show that BalenaDB can flexibly handle our dynamic synthesis workload by generating appropriate sizes of DBMS instances (Sect. 7).

Note that BalenaDB focuses on facilitating DBMS replication for handling request increases, not for improving service reliability against hardware failures and natural disasters. Creating DBMS instances in other physical machines is adequate to protect services against unpredictable failures such as administrative mistakes, power failures, and natural disasters. BalenaDB replicates a DBMS in instances running on the machine where its master instance runs. To protect DB services against failures, we need to use other replication schemes that generate replicas on another physical machine and/or other datacenters [11], [12].

## 2. Background

### 2.1 Replication of DBMS Instances

The replication of DBMS instances is essential on IaaS platforms. Cloud users create and delete an appropriate size of DBMS instances to adjust the total performance of their DB service and avoid wasting resources such as CPU and disk bandwidth. Figure 1 shows a typical DBMS instance replication. Although we can scale up our service simply by increasing active instances where stateless applications such as Web servers are running, generating a DBMS replica instance is complicated due to its rigid data management. To prepare a DBMS instance, we first launch an instance and then invoke a master DBMS function to create a replica. The function takes a snapshot of the DB file, transfers it to a newer instance, and starts the replica after the DB transfer is complete. At this point, the load balancer adds the DBMS replica instance and redirects a part of the read queries to it. The function also periodically propagates DB updates of the master to the replicas to maintain data consistency between the same DBs in its policy.

In generating a DBMS replica instance, the time for the DBMS replication is dominant. Launching an instance takes anywhere from a few dozen seconds to a few minutes, and research efforts have explored ways to launch instances more quickly [13], [14]. With DBMS replication, on the other hand, the time needed is ten minutes or more due to the DB-copy and buffer-pool warming up. To guarantee the quality of their DB services under bursty requests from the Internet [3], cloud users have to prepare redundant active DBMS instances proactively. Since it is quite difficult and almost impossible to estimate the necessary number of the DBMS instances in advance precisely, the appropriate instance provisioning is hard [15]–[18]. The under-provisioning of the DBMS instances leads to DB service disruption due to slow DBMS replication. At the same time, over-provisioning affects redundant customer payments and the inefficient utilization of computational resources.

BalenaDB is designed for IaaS platforms, where cloud users enhance their service performance by increasing instances built from a specific image that has already been uploaded. They create and destroy the instances on demand. We also assume that the target DBMS supports a persistent DB, transactional processing, and data consistency management between the master and slaves. These functionalities are commonly supported by modern DBMSs such as MySQL and PostgreSQL.

### 2.2 Previous Approaches

Here, we give an overview of previous approaches to replicate DBMS instances. **DBMS-level Replication:** The modern DBMS offers a replication function that clones its DBMS on another host while maintaining data consistency. For example, mysqldump [19] exports the logical backup data into text files by reading from each table using SQL
with the REPEATABLE READ isolation level, while Mydumper [20] does so in parallel using multiple threads and XtraBackup [21] generates a binary snapshot of the target DB file while blocking the running MySQL.

Unfortunately, generating a DBMS replica is a time-consuming task and can be disruptive to the DB service. Our experiments (Sect. 7) revealed that it takes 18 to 28 minutes to generate and start a DBMS replica whose DB size is 16 GB, and the buffer-pool warming of MySQL using 16 GB of memory requires up to 30 minutes.

DB sharding tools [22]–[25] are useful for balancing write queries among multiple DBMSs. Typically, slave DBMSs have their partitioned DB and handle write queries to their DB elements while the master DBMS receives write queries and offloads them to an appropriate slave DBMS. Although these techniques shorten the DB copy phase since copying the whole DB is unnecessary, the slave DBMSs spawned by these techniques suffer from the warm-up buffer-pool problem.

**Amazon RDS:** Amazon AWS has a DB-specialized service called the Amazon Relational Database Service (RDS) that offers functionalities to manage DBMS instances flexibly. Amazon RDS performs DB backups and data consistency management of users’ DBMS instances. Amazon RDS performs DB backups and data consistency management of users’ DBMS instances. Amazon RDS/Aurora also offers MySQL-based DBMS instances. To help cloud users handle read query bursts, such as OLAP workloads, to their DB service, Amazon RDS/Aurora has a function that launches read-only DBMS instances called read replicas. We create read replicas from the original DB instance using this function instead of performing manual DBMS replication procedures.

In Amazon RDS/Aurora, the data consistency management between the original DBMS and read replicas depends on DBMS-based mechanisms. For example, if a user runs MySQL, MySQL’s mechanism propagates the updates of the original DB file to read replicas in its policy. To mitigate the large data transfers in DBMS replication, Amazon RDS automatically takes one snapshot of the DB per day and stores it on a virtual disk for the read-replica. By doing so, the master DBMS can transfer only incremental updates from the snapshot to the new replica. Read-replicas can handle requests just after they are launched because the DBMS function does not copy the whole DB file from the original DBMS instance to the read-replicas. Also, read-replicas suffer from buffer-pool miss problem.

**Google Cloud SQL:** The replication process of DBMS instances in Google Cloud SQL [27] is similar to that of Amazon RDS, which means it also suffers from the buffer-pool miss problem. The replication functionality in Google Cloud SQL supports MySQL only. It periodically takes a snapshot of the target MySQL, and its replication function generates a MySQL replica instance using the latest backup. And then, the MySQL replication mechanism maintains data consistency of the DB by repeatedly shipping a binary log of the master to the replica and applying it to the replica DB.

**DB2 for VSE & VM:** DB2 Server for VSE & VM [28] is a commercial solution for VM-based DBMS consolidation on IBM zSystems. It uses a function called VM Data Spaces Support (VMDSS) for efficient VM consolidation. VMDSS [29], [30] enables us to create virtual storage space and share it between DB2 instances, resulting in the reduction of storage space utilization and disk server accesses.

Disk sharing approaches achieve space efficiency but are not suitable for load balancing. Since the disk accesses of DB2 instance concentrate on a single disk volume on which shared blocks are, disk contention affects their performance. In addition, DB2 still relies on the DBMS-level replication named Q replication [31]. In this replication, the master identifies the changes in data from the committed transactions and sends the transactional messages to the replica. And then, it applies the changes to its DB as a single transactional unit.

**VM fork:** The semantics of VM fork families [32], [33] are similar to those of the process fork: namely, users issue a fork call to the hypervisor that creates a child VM. The child VM inherits the runtime state, such as the memory and registers of the parent VM, and then proceeds with an identical view of the system. The child VM has its independent copy of the OS, virtual disk, network interface card (NIC), and snapshot. The state updates of the child VM are not propagated to the parent.

Although the VM fork can quickly clone a target instance, the resource configurations of the child instances (the number of cores, the memory size) are always the same as the parent’s ones. This makes it impossible to prepare appropriate instances for the workload. For example, when a DBMS instance whose memory is 32 GB becomes slightly too heavy, the VM fork spawns a child instance with the same specs as the parent, even if adding a smaller DBMS instance would be enough to handle the workload. This over-provisioning leads to redundant payments for cloud users and inefficient resource utilization.

In addition, DB read queries can place a significant burden on one disk drive. Since the child instance shares the parent’s virtual disk in a copy-on-write (COW) manner, the VM fork mechanism directs the disk read requests of both parent and children to the virtual disk. The performance of the DB service cannot be improved by the VM fork when a workload that causes disk reads of the DBMS occurs. In some cases, the performance will degrade due to the disk contention.

Also, the VM fork does not cooperate with any DBMS-based replication mechanisms. Since the VM fork is a hypervisor-level mechanism, creating a child instance is done transparently to the running DBMS, which means no mechanism propagates updates on the master DB to the replica ones. To propagate the updates, we have to develop an update mechanism that appropriately updates replicas’ DB after the VM fork generates a clone instance. Developing such a mechanism is not easy since the mechanism is tightly coupled with the DBMS internals to update replicas while maintaining data consistency carefully.
3. BalenaDB

This paper presents BalenaDB that performs urgent DBMS replication for achieving stable DB services under sudden workload increases. Our approach is based on the fact that the modern data center machines are highly scale-up [4], [5]; equipped with numerous CPU cores, a large amount of memory, and multiple disk drives. We can host multiple instances on a single machine by assigning them slices of abundant resources such as cores, memory, and disks. The key insight behind BalenaDB is to exploit the runtime states of the master DBMS, such as the DB and buffer-pool, for quickly generating its warmed-up replica. The initial states of the replica are almost the same as those of the master DBMS. This fact motivates us to run both the master and replica DBMS instances on the same scale-up machine and replicate the master DBMS by leveraging its DB and buffer-pool in a sharing manner.

To effectively create warmed-up DBMS replicas, the design of BalenaDB is driven by the following goals:

- **Minimized dependency on DBMS internal functions:** When designing BalenaDB, to improve its applicability, we carefully avoid using specific functions of a DBMS. Although this paper presents a case study of using MySQL only, we believe that BalenaDB can be applied to other DBMSs such as PostgreSQL.

- **Transparency to DBMS-based data consistency functions:** BalenaDB enables administrators to use existing DBMS-based replication mechanisms. The target DBMS can manage data consistency between the master and slaves by using its policy.

- **Applicability to various instance sizes:** Unlike the VM fork series, BalenaDB is not limited to the creation of the same size of replica instances, thus leading to efficient resource utilization by preparing the appropriate instances.

- **Isolation between DBMS instances:** To retain multi-tenancy in IaaS platforms, BalenaDB maintains the isolation between the master and replica DBMS instances.

Figure 2 gives an overview of DBMS instance replication on BalenaDB. When creating a DBMS replica, BalenaDB can skip the DB copy phase by having the master DBMS share its DB with the replica. Specifically, BalenaDB copies only the DB file’s metadata to the new instance and then starts the DBMS replica. It transfers the rest of the DB blocks in the background to achieve a quick replica start. BalenaDB also warms the buffer-pool of the replica by sharing belonging to the master DBMS. By doing so, BalenaDB achieves rapid buffer-pool warming of the replica.

4. Design Details

To ensure transparency to the DBMS data consistency management and isolation between DBMS instances, BalenaDB’s mechanisms mainly work at the hypervisor-levels (outside the DBMS and OS kernel) and obtain information by intermediating interaction between the DBMS and its DB file. To replicate the DBMS on variously sized instances, BalenaDB does not inherit the instance-level states. Rather, it reuses the DBMS-level running states of the master DBMS by two mechanisms: enlightened DBMS page sharing and lazy DB copying.

4.1 Enlightened DBMS Page Sharing

When a DBMS replica spawned by the regular DBMS-integrated replication mechanisms starts, its buffer-pool is cold; thus, its performance might be much worse than that of the master DBMS due to having to fetch DB blocks from the disk. To address this issue, we introduce enlightened DBMS page sharing that helps generate a warmed-up replica DBMS. Briefly, enlightened DBMS page sharing reuses the buffer-pool of its master to warm up the buffer-pool of a replica DBMS.

To reuse the buffer-pool of the master DBMS for its replica, we make extensive use of an I/O-based page sharing approach for VMs [34], [35]. In this approach, instances share memory pages of their kernel-level buffer cache with each other in a copy-on-write (COW) manner. Since the hypervisor-level mechanism can handle page faults caused by modifying the shared buffer cache page, this page sharing is transparent to the instances, and their isolation is kept. The hypervisor monitors disk read requests from the running instances, all of which have the same shared disk, and manages a table that associates the accessed block numbers with the machine frame numbers (MFNs) of the kernel-level buffer cache pages. It registers both the accessed block number and cache page’s MFN to the table every disk reads. When the requested block number hits the table, the corresponding page is shareable among the VMs. However, this approach cannot share the buffer-pool pages among the master and replica DBMS because the hypervisor does not understand which pages are used as a buffer-pool, which is known as the semantic gap [36].

To overcome this challenge, we exploit DBMS semantics at the hypervisor so that it can associate DB blocks with
4.2 Lazy DB Copying

In the DBMS-integrated replication, a DBMS replica cannot start until the DB copy from the master has been completed. To address this issue, lazy DB copying, inspired by DB live migration techniques [9], [10], allows the DBMS replica to start handling requests even if the DB copy is not yet completed. Briefly, this mechanism first copies metadata of the DB and launches the DBMS replica. It is copying the rest of the DB blocks while the DBMS replica handles requests.

One challenge here is how to keep the original states of the DB. The DBMS replica requires DB data whose contents are at the point when the replication starts. The master DBMS updates its DB while the replication mechanism propagates the updates based on its policy. Unlike the DB live migration, the mechanism of lazy DB copying creates a delta DB in the master DBMS instance and redirects updates to it. When the DBMS replication begins, lazy DB copying makes the master DB read-only and creates a delta DB. In receiving write requests, the master DB stores the updates to the delta DB in a COW manner. The update propagation depends on the DBMS-integrated function.

Another challenge is how to capture the master DB at a point when the DBMS replication is ordered. Preserving only the DB file is not prudent since the master DB has dirty pages containing updated data in memory. To start a DBMS replica with the correct DB state, lazy DB copying exploits the crash recovery function of the DBMS. Specifically, our mechanism copies a transaction log as well as the DBMS metadata to the replica instance. The DBMS replica conducts its crash recovery with the transferred transaction log and thus builds the same DBMS state as naturally as it was when the DBMS replication began. Figure 4 gives an overview of lazy DB copying. When the DBMS replication begins, the master DBMS creates a delta DB, transfers the DB’s metadata, and a transaction log to the instance for the replica. The DBMS replica starts handling queries after building its DB state by conducting a crash recovery with the log. The hypervisor copies the DB blocks to the replica instance in the background while preferentially sending DB blocks accessed on demand. When the master receives an update query, the guest OS stores the updated blocks in the delta DB.

Lazy DB copying offers a mechanism against failures while copying the DB file. When copying the DB file stops due to failures, the replica does not have a complete set of the DB file and thus fails to handle a request for untransferred DB blocks. A mechanism of lazy DB copying conservatively stops the DBMS replica and reports it to the administrator. If necessary, he or she retries BalenaDB-based replication again if necessary. Since the DB file of the master DBMS is not affected by such a failure, the master continues to run.

5. Implementation

We implemented a prototype of BalenaDB on MySQL 5.6.21, Linux 3.17.2, and Xen 4.4.1. In implementing the prototype, we do not use specific features of the software, so BalenaDB is conceptually applicable to other software. Our prototype consists of three modules: a user-level module, kernel-level module, and hypervisor-level module. The user-level module hooks the read and write requests of MySQL and propagates their information to the kernel-level and hypervisor-level modules. These two modules then cooperate based on the user-level module trigger of lazy DB copying and enlightened DBMS page sharing. In our prototype, the data ID of MySQL, which is a unique ID assigned to every DB data block, can be represented by a combination of two values: space and offset.

User-level Module: The user-level module consists of two processes: a replication manager and a lazy cloner. Both are related to lazy DB copying. The replication manager, run-
ning on domain 0 (dom0), receives a request for replicating the DBMS from users via TCP connections and starts lazy DB copying and enlightened DBMS page sharing. The lazy cloner, running on domain U (domU) for a replica, generates a sparse DB file whose size is the same as the master DB file and launches MySQL as a replica. The lazy cloner then issues a hypercall to fetch data of the DB file in the master domU’s disk image and copies it to the sparse file. During lazy DB copying, the master MySQL stores updates in a delta file.

For ease of implementation, for lazy DB copying and enlightened DBMS page sharing, we slightly modify MySQL to issue system calls exposed by the kernel-level module (described in the next section). Our modification replaces `pread()` and `pwrite()` to the DB file with our system calls. We believe that implementing a library that hooks the `pread()` and `pwrite()` is an effective way to apply BalenaDB to MySQL without having to modify it.

**Kernel-level Module:** To perform lazy DB copying and enlightened DBMS page sharing, the kernel-level module exposes three system calls: `balena_read()`, `balena_write()`, and `balena_transfer()`. The `balena_read()` and `balena_write()` are used as alternatives of `pread()` and `pwrite()` to the DB file so that the kernel-level module can extract the read/write position of the DB file. The `balena_transfer()` is used by the lazy cloner to copy the original DB file. The `balena_transfer()` copies the fetched original DB blocks to the space file without copying it to the user-level buffer; thus, we can skip the two copies between the kernel- and user-level.

In lazy DB copying, the lazy cloner repeatedly issues the `balena_transfer()` until the master DB file copying is complete. To detect when that is, the kernel-level module manages and updates a block bitmap that shows which block in the DB file has been copied to memory pages. The `balena_read()` checks whether the requested data ID has already been copied or not. If it has not, the `balena_read()` fetches it from the master DB file by calling `balena_transfer()` and then copies it to the sparse file. Otherwise, the data is fetched from the replica DB file.

The kernel-level module also communicates with hypervisor-level module to make the buffer-pool pages of the master shareable. The `balena_read()` issues a hypercall exposed by the hypervisor-level module to pass the data ID and the physical frame number (PFN) of the buffer-pool page. The hypervisor-level module converts the passed PFN to MFN and registers it to the table for the enlightened DBMS page sharing.

**Hypervisor-level Module:** The hypervisor-level module manages page sharing information and shares pages between the master and replica DBMSs. It maintains a hash table for each domU that shows which pages are shareable. To create this table, the hypervisor-level module exposes a hypercall whose arguments include the data ID and PFN. The hypervisor-level module registers the MFN to the hash table using the data ID as its key. When a user sends a replication request to the replication manager, it notifies the hypervisor-level module of the request via a system call and hypercall, and the hypervisor-level module sets specified machine pages to be write-protected.

To share pages between the master and replica DBMSs, the hypervisor-level module also exposes a hypercall, `insert_shareable_cache()`, to check whether the passed pages are shareable or not by using the hash table. The `balena_read()` issues `insert_shareable_cache()`. If the passed pages are shareable, the hypervisor-level module updates P2M and M2P table entries of the instances to the MFS of the original DB data. For ease of implementation, the kernel-level module conducts these updates. Specifically, the kernel-level module requests the update of the P2M and M2P tables to the Xen hypervisor via an existing hypercall if the DB data to fetch are shareable. At this point, the kernel-level module returns the same number of shared pages to the hypervisor so that the total memory size of the domU is consistent.

6. Discussions

BalenaDB is friendly to live migration that moves the running instances between different physical hosts. Even if a replica instance shares the DB file and buffer-pool of the master instance by lazy DB copying and enlightened DBMS page sharing, the mechanism of live migration running inside the hypervisor identifies the replica instance’s memory and disk blocks and can thus successfully transfer them to the destination machine.

Enlightened DBMS page sharing is a complementary use of scan-based page sharing mechanisms [37], [38]. Scan-based page sharing, which detects pages whose contents are the same as a certain page and then maps them to that page, contributes to the same page deduplication on BalenaDB-based systems.

Our prototype currently does not have any mechanism to merge a delta DB file with the original DB one. Since BalenaDB forces the original DBMS to store updates in a delta DB file in order to maintain the original DB contents during lazy DB copying, we have to manage such additional files. There are several strategies for merging the stored updates on a DB file. For example, we can apply the updates to the DB file just after lazy DB copying is complete in batch, with disk I/O restriction, at DBMS idle time, and so forth. The best policy depends mainly upon the situations and/or the user’s preference. Exploring policies is outside the scope of this paper.

To quickly create warmed-up replicas, distributed memory cache systems such as memcached and Redis are helpful. The memory cache systems cache DB blocks and the newly-generated replicas warm up their buffer-pools by fetching the DB blocks from them without access to backend storage. This way is complementary to enlighten page sharing. Caching DB blocks is applicable to replicas generated by the regular replication. Compared to the cache approach, to create warmed-up replicas on the local machine, enlighten page sharing is an efficient solution due to no re-
mote access.

7. Experiments

We conducted several experiments to demonstrate the effectiveness of BalenaDB. The objective of the experiments is to answer the following questions: 1) How long does BalenaDB take to start the DBMS replica?, 2) How long does it take to finish lazy DB copying?, 3) What is the performance penalty caused by BalenaDB?, 4) How does BalenaDB handle a scale up workload?, and 5) How does BalenaDB handle a synthetic workload?. We conducted the experiments on an IBM System x3755 M3 server with 64 cores, 256 GB of memory, and eight 7,200 rpm hard disk drives. We run our prototype on it.

7.1 Replica Start Time

To demonstrate how long BalenaDB takes to start a replica of the target DBMS, we measure replica start time, which is time for making a new replica ready to handle queries. We set up MySQL with a 16 GB DB file on a domU with one virtual CPU and vary the dirty page sizes of the master MySQL from 0 to 256 MB. Since a new replica carries out its crash recovery to start with the same DB state as the master one, replica start time depends on the dirty page size in the transaction log. We also change the domU’s memory from 4 to 32 GB and configure the MySQL memory size to be domU’s memory size \(\times 0.75\). We replicate the MySQL on another domU whose resource configuration is the same as the already-started domU. For comparison, we also measure replica start time of the MySQL’s representative replication scheme, Xtrabackup-based replication, which takes a binary snapshot of the target DB file, transfers it to the replica instance and starts the replica after the DB file is restored from the snapshot.

The results are shown in Fig. 5, where the x-axis and y-axis are dirty page size and replica start time, respectively. As shown, BalenaDB takes longer as the dirty page size becomes bigger. When the dirty pages are zero to 256 MB, the replica start time of BalenaDB varies from 25 to 110 seconds. The crash recovery mechanism checks all dirty pages in the log and writes them to the DB file. The replica start time is also slightly longer as the buffer-pool size gets bigger. The replica start time in the 32 GB case is up to 8.7 seconds longer than that in the 8 GB case since the MySQL replica takes longer to initialize buffer-pool data objects such as the LRU list and the buffer-pool.

7.2 DB Copy Time

To show the effectiveness of lazy DB copying, we measure the time taken for the DB copy. We set up a domU with 1 VCPU and 16 GB of memory and runs MySQL with a 16 GB \(\times 0.75\) buffer-pool on it. The DB file size and the copy bandwidth are changed from 10 to 40 GB and from 2.5 to 10.0 MB/s, respectively. We also observe CPU utilization to show how each bandwidth affects domU’s service. We conduct the same experiment using Xtrabackup-based replication.

Figure 6 shows the results. Figure 6 (a) plots time for lazy DB copying, where the x-axis is DB size and y-axis is completion time. The x-axis is DB size and y-axis is completion time. The figure indicates that BalenaDB successfully controls the DB file copy based on the defined copy bandwidths. Completion time for the DB copy is inversely proportional to the copy bandwidths. In the 10 GB case, the completion time under 2.5 MB/s is 4221 seconds, which is 2x, 3x, and 4x longer than the 5.0, 7.5, and 10.0 MB/s cases. This situation is the same in the other DB size cases.

Figure 6 (b) shows average CPU utilization on domUs for the master and replica MySQL during each replication. The CPU utilization of Xtrabackup is much higher than that of BalenaDB. The CPU utilization on domUs for the master and replica MySQL in BalenaDB is 0.5 and 6.0% in the 2.5 MB/s case and 0.25 and 12.4% in the 10.0 MB/s case. In contrast, these CPU utilization on Xtrabackup are 13.0 and 15.9% for 2.5 MB/s and 32.6 and 41.6% for 10.0 MB/s. This is because Xtrabackup is a user-level mechanism and thus data copies from the kernel- to user-space are more than lazy DB copying which skips such copies.

7.3 Overhead

The first is overhead coming from DBMS replication.

---

**Fig. 5** Replica Start time.

**Fig. 6** Lazy DB copying.
We measure the master MySQL’s throughput during BalenaDB’s replication. We note that the total throughput of the master and its replica MySQL is shown in the next section. For this experiment, we run domU with 16 GB of memory and MySQL with a buffer-pool of 16 × 0.75 GB. We used three benchmarks: Sysbench-small, Sysbench-large, and TPC-C. Sysbench is configured to be a read-only benchmark, while TCP-C is an OLTP-based benchmark whose working set is 16 GB. The difference between Sysbench-small and -large is their working set size: Sysbench-small’s is 12 GB, which is small enough to be in MySQL’s buffer-pool, while Sysbench-large’s is 24 GB, which is bigger than the buffer-pool size and thus increases disk accesses. We start a workload and then replicate the master MySQL in 15,000 seconds. We also set the data transfer speeds to be 2.5, 5.0, and 10 MB/s.

The master MySQL throughput during replication is shown in Fig. 7. The figure reveals that the performance penalty depends mainly on the workload types. In the Sysbench-small case, the throughput is 3% lower than that of regular operation. In the Sysbench-large and TPC-C cases, the performance degradation is up to 47.3 and 47.8%, respectively. The reason the performance degradation is more severe in the latter two benchmarks is that these benchmarks and lazy DB copying fetch DB blocks from the same disk drive, thus causing disk contention. Also, the data transfer speeds change the performance penalty intervals.

Another reason for the performance degradation is runtime overhead incurred by enlightened DBMS page sharing. Enlightened DBMS page sharing registers the page address of DB blocks to its hash table when fetching them from the disk drive. To measure this overhead, we compare the throughput of MySQL under Sysbench-small and TPC-C, whose working set size is the same as the buffer-pool one. We change the MySQL buffer-pool size from 4 GB to 128 GB and start the benchmark with cold OS and buffer-pools.

The results, given in Fig. 8, show that runtime overhead incurred by BalenaDB is lower than 2%. In the Sysbench and TPC-C cases, the largest overheads are 0.26 and 1.52%, respectively. Since the enlightened DBMS page sharing’s check is accompanied by disk accesses for DB block fetches, its overhead becomes relatively small.

### 7.4 Scalability

To show the effectiveness of BalenaDB for a scaling workload, we measure the total throughput of the running DBMSs. We use Sysbench-small as a memory-intensive workload and TPC-C and Sysbench mixed workload as a disk-intensive one. The master MySQL handles the TPC-C workload, and its replicas handle the Sysbench that reads the TPC-C DB. The working set sizes of these benchmarks are 12 and 16 GB, respectively. We first run a master MySQL with a benchmark for a while, launch a new domU, start another Sysbench, and replicate the master MySQL. This is done seven times. We prepare four configurations: Xtrabackup-based replication, BalenaDB without lazy DB copying or enlightened DBMS page sharing, labeled Lazy and PGshare, and a full set of BalenaDB.

The results are shown in Fig. 9. From these figures, we can see that BalenaDB swiftly handle a scaling workload. In both benchmark cases, BalenaDB replicates MySQL faster.
than the other configurations. In the memory-intensive workload, the replica start time of BalenaDB is 47x faster (23 seconds) than one of Xtrabackup (18 minutes). In addition, the throughput of the replica spawned by BalenaDB is much higher since BalenaDB generates the already warmed up replica by enlightened DBMS page sharing. The replica start time of Lazy is the same as one of BalenaDB, but this configuration needs to warm up the buffer-pool of the replica MySQL, which takes more than 30 minutes. The throughputs of the PGshare and Xtrabackup cases are similar for the following reasons. One, neither replication can start the MySQL replica until the DB file copy finishes, and two, Xtrabackup’s DB file copy warms the buffer-pool of the replica, which make the MySQL replica’s performance high at its start point, similar to the effect of enlightened DBMS page sharing.

BalenaDB swiftly handles the disk-intensive workload, as shown in Fig. 9 (B). The replica start time is 136 seconds in BalenaDB and 29 minutes in Xtrabackup. The throughput in Lazy is almost the same as that of BalenaDB, since the workload is disk-intensive, and thus the buffer-pool is not effective. The throughput of PGshare increases more slowly than that of Xtrabackup because it efficiently propagates DB’s updates.

Figure 9 (A) and (B) show the memory usage of the running DBMS instances. In the Sysbench case, enlightened DBMS page sharing effectively shares the buffer-pool between the master and replicas because each Sysbench instance has the same DB set. The memory usage spikes just after the BalenaDB’s replication, and then each replica shares the buffer-pool with the master MySQL, resulting in much lower memory usage than the default. When the seven replicas are running, the memory usage in BalenaDB is 313% lower than the configuration without enlightened DBMS page sharing. Since TPC-C is write-intensive, enlightened DBMS page sharing is not as effective as in the Sysbench case.

7.5 Synthetic Workload

To demonstrate that BalenaDB can use different sizes of instances for replicas to handle workload intensity efficiently, we create MySQL replica instances with different resources using BalenaDB. We first vary the memory size of an instance from 8 to 24 GB. The master MySQL instance has 16 GB of memory, and the buffer-pool size of the running MySQL is 16 GB \times 0.75. We run Sysbench-large and TPC-C, both used in the previous section, as read- and write-intensive workloads. We measure their benchmark scores and the time for warming the replica’s buffer-pool.
For comparison, we conduct the same experiment using Xtrabackup-based replication.

The results are shown in Fig. 10. BalenaDB successfully spawns the different memory sizes of MySQL replica instances in a memory-efficient manner. In the Sysbench-large case (Fig. 10-(A)), the total throughput of BalenaDB is up to 37.1% higher than that of Xtrabackup. This is because the master and replica MySQL sometimes fetch non-cached DB blocks without disk accesses due to enlightened DBMS page sharing. When a buffer-pool miss occurs in the master MySQL, enlightened DBMS page sharing checks the hash table of the replica MySQL and shares the buffer-pool page of the target DB blocks if the replica MySQL has already cached the blocks. The same situation happens when a replica MySQL tries to fetch DB blocks.

BalenaDB also successfully generates different memory sizes of MySQL replica instances with a shorter buffer-pool warming time in the TPC-C case shown in Fig. 10-(B). The throughput of BalenaDB is slightly better than that of Xtrabackup. Even if the TPC-C workload updates the buffer-pool pages of the master MySQL, enlightened DBMS page sharing shares the unmodified pages with its replica, which leads to improved total throughput and shorter buffer-pool warming. In BalenaDB, the buffer-pool warming time of TPC-C is much slower than that of Sysbench-large since TPC-C modifies the buffer-pool pages of the master MySQL and thus reduces the number of shareable pages with the replica.

Next, we prepare a synthetic workload to investigate how BalenaDB handles workload changes by generating different CPU numbers of replica instances. The synthetic workload mixes different intensities of Sysbench-small, as shown in Fig. 11 (a). We prepare two types of replica instances: small and big. The small and big instances, both of which have 16 GB of memory, are equipped with 4 and 8 VCPUs, respectively. We execute a replication policy using these instances based on Amazon Auto Scaling [39], where we trigger replica instance starts and stops using the average CPU utilization during the last two minutes. Specifically, our policy launches one small, one big, and two big replica instances in 65 to 75%, 75 to 85%, and more than 85% of total CPU utilization while shutting down two big, one big, and one small replica instances in 35 to 40%, 25 to 35%, and less than 25% of total CPU utilization, respectively. We first launch an instance whose resource configuration is the same as the small replica instance and run MySQL on it as the master.

Figure 11 (b), (c), (d), and (e) show the experimental results. Figure (b) and (c) show total throughput using BalenaDB- and Xtrabackup-based replication. These figures show replica instance generation and destruction. BalenaDB follows workload changes while Xtrabackup fails to handle the workload in 1000 to 1720 seconds due to its MySQL replication, where Xtrabackup takes more than ten minutes to copy the DB file and warm the replica’s buffer-pool. In addition, the throughput of Xtrabackup becomes unstable in 7720 to 8458 seconds, which is when replication is triggered. The reason for the degradation is that generating a MySQL replica takes up the huge CPU resources of the master MySQL instance; thus, CPU contention occurs.

8. Related Work

BalenaDB mitigates over- and under-provisioning by swiftly generating a warm-cached DBMS replica instance in a memory-efficient manner. As described in Sect. 2, the VM fork families [32], [33] can quickly clone the target instance but has some limitations. The resource configuration of each
cloned instance must be the same as the parent one. Second, disk contention continues to occur under read-heavy workloads. Lastly, it is inherently difficult to use the VM fork in collaboration with the DBMS replication. Some research has aimed at rapidly building instances from special template images. For example, VM substrate [13] launches an instance by restoring its state from a small memory snapshot and attaching it to the template COW virtual disk. Potemkin [14] offers flash cloning that creates an instance from a VM snapshot placed in memory. While these approaches shorten the instance launch time, BalenaDB shortens the time for generating a warmed-up DBMS replica.

Numerous efforts have focused on page sharing for instances. Enlightened DBMS page sharing makes extensive use of the page-sharing mechanism using shared disks, inspired by Satori [35] and XLH [34]. These techniques associate a read block number with the address of the page caching it, while Enlightened DBMS page sharing focuses on sharing the buffer-pool pages using DBMS-level block information. Other page sharing mechanisms detect pages with the same or similar contents by periodically tracking all memory pages [37], [38], [40]. These are complementary to BalenaDB.

Several studies aim at efficient resource management by using a DBMS-level mechanism and OS- and/or hypervisor-level mechanisms. Application-level ballooning [41] allows us to return unused DBS memory to the hypervisor. RemusDB [42] is a VM-based hot standby system optimized by exploiting DBMS characteristics. COD [43] provides an OS-level mechanism to install hints for DBMS’s requested thread scheduling. BalenaDB rapidly creates a warmed-up DBMS replica.

BalenaDB swiftly creates a replica of the target DBMS independently of update propagation protocols. We can use BalenaDB in combination with an update propagation protocol such as query-based propagation [44]–[46] or log-based propagation [19], [21], [47].

While BalenaDB focuses on DBMS instance replication for load balances, the goal of other approaches is to attain the availability of the DB service. SHADOW [11] is an efficient hot-standby system for DBMS instances. It shares the transaction log between active and standby DBMSs and forces the standby DBMS to checkpoint itself to mitigate I/O overhead on the active DBMS. PipeCloud [12] performs an efficient synchronous replication of the DBMS on a remote host. BalenaDB aims at handling an increase in workloads.

Migrating a DB or DBMS from one to another physical machine facilitates the management of data center resources and satisfaction of service level agreements. Albatross [9], Zephyr [10], Madeus [48], and Slacker [49] all achieve live migration of a DB and DBMS. Live VM migration [50], [51] moves the running VM between different physical machines. ShuttleDB [52] appropriately selects DB- and VM-level migration to achieve efficient DBMS replacements. BalenaDB focuses on DBMS replication, not migration.

9. Conclusion
This paper presented BalenaDB, which swiftly replicates a DBMS to support swift replication of DBMS instances on modern scale-up machines. BalenaDB runs transparently to the DBMS-based mechanisms for update propagation and can generate a DBMS replica on an instance with various specs, maintaining isolation between the master and replica DBMS instances. We implemented a prototype of BalenaDB on MySQL 5.6.21, Linux 3.17.2, and Xen 4.4.1. The experimental results show that our prototype successfully delivers flexible performance for different workload intensities by generating a DBMS replica readily.

References

[1] “Amazon Elastic Compute Cloud.” http://aws.amazon.com/ec2/
[2] “Google Cloud Platform.” https://cloud.google.com/compute/
[3] P. Bodik, A. Fox, M.J. Franklin, M.I. Jordan, and D.A. Patterson, “Characterizing, Modeling, and Generating Workload Spikes for Stateful Services,” Proc. 1st ACM Symposium on Cloud Computing (SoCC ’10), pp.241–252, 2010.
[4] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, “Scale-up vs Scale-out for Hadoop: Time to Rethink?,” Proc. 4th ACM Annual Symposium on Cloud Computing (SoCC ’13), pp.20:1–20:13, 2013.
[5] T.H. Hethertong, M. O’Connor, and T.M. Aamodt, “MemcachedGPU: Scaling-up Scale-out Key-value Stores,” Proc. 6th ACM Symposium on Cloud Computing (SoCC ’15), pp.43–57, 2015.
[6] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang, “PerfIso: Performance Isolation for Commercial Latency-Sensitive Services,” 2018 USENIX Annual Technical Conference (USENIX ATC ’18), pp.519–532, 2018.
[7] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, “Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience,” Proc. 19th USENIX Conference on File and Storage Technologies (FAST ’21), vol.17, no.4, pp.33–49, 2021.
[8] J. Leverich and C. Kozyrakis, “Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service,” Proc. Ninth European Conference on Computer Systems (EuroSys ’14), pp.1–14, 2014.
[9] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, “Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration,” Proc. VLDB Endowment, vol.4, no.8, pp.494–505, 2011.
[10] A.J. Elmore, S. Das, D. Agrawal, and A.E. Abbadi, “Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms,” Proc. 2011 ACM International Conference on Management of Data (SIGMOD ’11), pp.301–312, 2011.
[11] J. Kim, K. Salem, K. Dandjee, A. Aboulnaga, and X. Pan, “Database High Availability Using SHADOW Systems,” Proc. 6th ACM Symposium on Cloud Computing (SoCC ’15), pp.209–211, 2015.
[12] T. Wood, H.A. Lagar-Cavilla, K. Ramakrishnan, P. Shenoy, and J.V. der Merwe, “PipeCloud: Using Causality to Overcome Speed-of-Light Delays in Cloud-Based Disaster Recovery,” Proceedings of the ACM Symposium on Cloud Computing (SOCC ’11), pp.17:1–17:13, Oct. 2011.
[13] K. Wang, J. Rao, and C.-Z. Xu, “Rethink the Virtual Machine Template,” Proc. 7th ACM International Conference on Virtual Execution Environments (VEE ’11), pp.39–50, 2011.
Kaiho Fukuchi received the B.E. and M.E. degrees from Tokyo University of Agriculture and Technology in 2014 and 2016, respectively. His interests are in operating systems, virtualization, and heterogeneous computing.

Hiroshi Yamada received his B.E. and M.E. degrees from the University of Electrocommunications in 2004 and 2006, respectively. He received his Ph.D. degree from Keio University in 2009. He is currently an associate professor of the Division of Advanced Information Technology and Computer Science at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, dependable systems, and cloud computing. He is a member of ACM, USENIX and IEEE/CS.