Adaptive Live VM Migration in Share-Nothing IaaS-Clouds with LiveFS

Nick R. Katsipoulakis, Konstantinos Tsakalozos and Alex Delis
University of Athens, 15784 Athens, Greece
{katsip, k.tsakalozos, ad}@di.uoa.gr

Abstract—Live migration is a versatile option when it comes to attain load-balancing in IaaS–cloud architectures. Liveness, reliability and conformance to SLAs may all be achieved by moving a VM that creates excessive work from its current physical machine (PM) to a less busy node. Despite its promising features, live migration is an expensive operation in terms of resources. The situation gets further exacerbated when the movement involves PMs working off different file-systems which is often the case in shared-nothing IaaS-cloud infrastructures. In this paper, we suggest an approach that adapts the migration operation based on the I/O activity of the originating-VM. We introduce LiveFS, a FUSE–file system which trapps all I/Os and helps determine how to best ship virtual disk segments across PMs in a share–nothing IaaS–cloud. Through prototyping and experimentation, we show that LiveFS can improve the shipment of VMs for diverse types of workloads. In particular, LiveFS succeeds in reducing the Total Migration Time by up to 64% compared to the “pre-copy” live migration technique. Furthermore during migration, we attain up to 19% less I/O-delay if compared to the “post-copy” live-migration approach.

I. INTRODUCTION

Contemporary IaaS-clouds are being designed as a means to offer sophisticated computing services without having users tackle complex maintenance and administration tasks. This is mostly accomplished by leasing virtual machines (VMs) running on networked clusters of physical machines (PMs). By and large, providers would be interested in maintaining load-sharing among their shared-nothing infrastructures. Inevitably however, the performance of a single PM will start to deteriorate due to multitenant applications hosted and/or the appearance of user flash-crowds. This does affect the co-existence of VMs in a single node. Live-migration has been proposed as a way to ameliorate such performance degradation by moving a busy tenant and its VM to another physical machine. In an initial approach for VM-migration [1], [2], [3], a VM was suspended and its main-memory image was transferred into the destination. Despite the fact that these approaches could achieve the load balancing among the tenants of a server, they would impose severe limitations on the migrating tenant by stopping its execution and making its service unavailable until the migration would complete.

In an effort to attain minimal downtime, the above “pure stop-and-copy” or “cold” approach was succeeded by the “live” or “hot” VM migration. This approach intends on transferring a VM while it is operating and its feasibility was first presented in [4]. The “pre-copy” live migration would first transfer the disk contents of the VM to its destination PM until only a small set of dirty pages would remain outstanding. Then, the VM execution was halted and these pages were pushed over to the target machine along with the CPU state. Finally, the operation of the VM would commence at the destination node. Although this method has the advantage that the I/O latency remains low, in a write-intensive workload the Total Migration Time has the potential to be significantly lengthy. The “post-copy” live migration approach [5] manages to reduce the Total Migration Time for VM memory images and diminish the VM downtime to near-zero level. In [6], it is shown however that live migration imposes a considerable amount of performance degradation. The migration of a VM may take up a large amount of the resources available to a machine and may affect other tenants’ response time.

In a shared-nothing IaaS-cloud where a common filesystem is not always feasible, a VM-migration may involve significant delays as multiple Gigabytes of a virtual disk (VD) have to be copied over the network. Moving voluminous disk segments even through Gbps–rated networks in a way that does not upset operations and imposes minimal overheads still remains a challenge. In this paper, we follow a different approach as we introduce a virtual disk I/O monitoring mechanism operating even before a migration commences. Over time, this I/O-monitoring allows for effective compilation of the “hot” disk segments and enables timely handling of long-term memory shipment. Our proposal combines the pros of the pre-/post-copy methods along with the identification of virtual disk (VD) segments that receive high traffic. In this respect, the VM operation at the destination PM gets to start as quickly as it occurs in the “post-copy” method yet with reduced I/O-request response–time. LiveFS embodies the above features through the realization of a user-space filesystem. We have developed a prototype and experimented with a file-system benchmarking suite as well as synthetic workloads. Our experimentation shows that LiveFS succeeds in reducing the Total Migration Time by up to 64% compared to “pre-copy” live migration method and lowering the I/O-delay during migration up to 19% compared to the “post-copy” migration approach. Our contributions are the:

- proposal of a hybrid–approach to handle VM live migration in an adaptive manner.
- exploitation of the data–access patterns by VM-tenants so that we alleviate the workload experienced by PMs cloud nodes.

II. OVERVIEW OF OUR LiveFS–BASED APPROACH

Fig. 1 depicts the organization of our IaaS-cloud and shows the key elements that help address the issues of VM
migrations. At the top, the Cloud-Middleware layer founded on either Openstack or OpenNebula [8], [9], oversees the operation of the shared-nothing infrastructure. Each PM-node maintains its own disk of which a number of virtual disks (VD) have been defined and are in use by corresponding VMs. Through the use of a hypervisor such as Xen [10], PMs can host a number of VMs with which the users of the system interact. We assume that the Cloud Middleware employs a Load Balancing Policy [14] capable of determining when a VM should be transported along with its requisite VD and to which PM. Typically when a VM’s resource utilization rates reach a level beyond which degradations appears imminent, the load-balancing policy selects a destination PM and has the originating node directly talk to the new PM to host the migrating VM. The middleware can then initiate a Migration Task. Every migration task (Fig. 1) entails the VM to be transported, source and target PMs involved and more importantly, the VD to be shipped.

Our approach takes action as soon as a migration task appears in the middleware layer. Our prime objectives are to: a) reduce VM-handover time; this is the elapsed time between the initiation of a migration until the respective VM becomes operational in the new host, and b) minimize the performance penalty inflicted to the VM’s operation due to the migration operation. The three salient elements of our approach are the: Migrations Daemon, LiveFS, and Migration Worker(s). Each migration task is handled by the Migrations Daemon that administers VM moves between physical machines. LiveFS is our special-purpose FUSE-based filesystem [17] that functions as an abstraction layer between the hypervisor and the local filesystem. Every PM features its own LiveFS instance that allows to intercept all I/O operations targeting the virtual disks on the move, while also maintain data-consistency. Finally, as Fig. 1 shows, the Migration Worker on each PM works in tandem with LiveFS to accommodate the required data shipment involved in the migration. In the course of a migration, the subsequent pieces of action have to be carried out: 1) transfer of the VD content, 2) transfer of the CPU-memory state, and finally, 3) the VM-handover from the source to the destination node has to occur. The VM-handover includes forwarding all application/user requests to the VM’s new host.

A. Migrations Daemon and Workers

The Migrations Daemon of the middleware handles migration tasks in FIFO fashion. For each job received, the Migrations Daemon contacts the Migration Worker of the PM hosting the VM that has to be moved. The daemon dispatches the ID of the VD-image that will be transported along with the information of the receiving PM.

There are a few factors that drive our approach and are defined at the initialization part of the migration. These factors are: monitoring time period, migration threshold, and handover-size. The monitoring time period defines the time length, in which the source node monitors I/O operations performed on the data segments of the VD-image to be transferred. This monitoring phase takes place so that we can identify the working–set of VD segments the VM uses. During this period, each segment is assigned a score produced by a ranking function. If a data segment’s score is higher than the migration threshold, this segment will be transferred before the VM-handover phase. As I/O-writes may occur before the handover of the VM we keep track of these updates and push them to the destination PM. The handover-size defines the maximum number of updates that remain to be sent before the system enters the VM-handover phase. All the above 3 factors are defined by the Migrations Daemon and are sent over to the source–PM’s Migration Worker to help guide the migration process.

The Migration Worker component of the source–PM contacts its counterpart at the destination–PM so as to establish a communication channel through which segments will be transported. The Monitoring Phase ensues at the source–PM. By the time Monitoring ends, all accessed segments of the VD under migration are ranked using our ranking function. All of the segments that are ranked above the migration threshold are shipped to the destination node, before the VM-handover phase. The transferred segments are kept in sync between the source and destination PMs by applying any update to both hosts. Finally, the migration enters its final stage where the remaining segments of the VD-image are sent over to the destination–PM.

B. LiveFS Design

LiveFS is a FUSE filesystem [17] and enables us to trap all I/O-operations. Fig. 2 depicts the route every I/O takes in LiveFS. All VD–images that reside in a PM are stored under a path termed the virtual disk repository. A VM is
deployed so that it does not directly access its virtual disk image, but to access the disks under LiveFS’s ultimate mount point. Every time an I/O occurs from the VM to its VD-image, this operation is routed to the local LiveFS instance and handled accordingly. The functionalities of LiveFS are to:

a) intercept I/O calls performed from the VM to LiveFSs mount point and forward them to the actual virtual disk repository,

b) monitor I/O operations on a soon-to-be-migrated VD-image,

c) re-send segments that have been updated before the handover phase commences.

d) at the destination node, if a read eventually asks for a segment that has not been sent yet, LiveFS has to notify its local Migration Worker to handle the request at hand.

III. MIGRATION PHASES AND THEIR FUNCTIONALITY

Our approach completes the live-migration procedure in 5 distinct operational phases. The collective goal of these phases is to ensure that the transfer of a VD-image occurs in a consistent manner. The LiveFS of a PM-node keeps track of all specifics of a VD under transfer. Such information include: identifier of the VD–image, IP and port number at the destination/source PM, the particular migration stage LiveFS finds itself in and whether the node acts as either sender or receiver of the VD. Fig. 3 depicts the various structures maintained in memory by LiveFS. We outline below the functionalities of the 5 phases and their interactions with the above structures.

1) Monitoring I/O-Requests: LiveFS enables the monitoring of all I/O-operations issued at the source, for time equal to the value of the monitoring time period parameter. As it is known [18], [19], keeping track of spatially overlapping I/Os, especially in the context of virtualized environments may become too complicated. Since we intended to employ a lightweight solution, we opted for a more coarse approach: we divide the VD–image in equal-length disk segments. The size of each such segment is configurable and presumed to be bigger than 4KB. This choice allows for easier decisions when it comes to determining segments that become “hot” due to multiple writes and reads, what has been transferred and whether there are outstanding operations for particular VD portions.

While experimenting with our prototype, we have come to the conclusion that segments of 64MB size present good performance of VM-image transfer and a readily manageable number of segments so that LiveFS’s in-memory structures grow modestly.

LiveFS features a comprehensive repertoire of calls including the live_read() and live_write() calls. As soon as the monitoring time period commences, LiveFS-calls constitute the mechanism to record I/O activity on the VD. To accomplish this, we maintain an in-memory monitoring hash table. The FUSE library offers the capability to translate an access to a specific VD address into an offset; this number provides the distance from the first byte of the virtual disk. Given a fixed (and configurable) size of segments, the segment–id can be computed using the above offset. We use the segment-id as a means to store/access records in the hash-table and in each such record we maintain read and write counters. Every time there is an invocation of the read/write calls and while in monitoring phase, the corresponding counter is augmented. If a call spans multiple segments, then counters of respective segments are properly adjusted.

At the end of the monitoring time period, the hash-table is scanned so that the traffic received by each VD-segment can be ranked. Our ranking function takes into account both read and write operations and computes a weighted average for both types:

\[
f(\text{segid}) = \frac{w_1 \cdot \text{nreads}(\text{segid}) + w_2 \cdot \text{nwrites}(\text{segid})}{2}
\]

where \text{nreads()} and \text{nwrites()} are functions that return the two counters for a specific segment identifier and \(w_1 + w_2 = 1\) (default values \(w_1 = w_2 = 0.5\)). This function involves all needed information for the popularity of a VD-segment after the migration process has started. We only track I/O operations that are forwarded to the filesystem (and in turn to the VD of a VM) as only these operations are crucial in the successful LiveFS migration. Every segment that scores above the configurable migration threshold is transferred during phases 2 and 3 while the remaining segments are transported during phase 5.

2) Pre-copy Phase: the scope of this phase involves the shipment of all those “hot” segments to the destination PM. The Migration Worker component of the source node places these segments in SendQueue for dispatch (Fig. 3). SendQueue consists of segment identifiers whose cardinality remains unchanged through the migration procedure. Segments referenced by SendQueue get transported to the appropriate PM in FIFO discipline as soon as the pre-copy stage starts. Knowledge of the set of segment-ids in SendQueue is also essential to settle matters after VM-handover occurs.

As updates can take place, some of the segments in SendQueue may be (re-)written while this phase progresses. We use SyncQueue to place ids of such dirty segments. Every time a write() call is executed, LiveFS examines if it refers to a segment that is sent during phase 2 (i.e., placed in SendQueue). Should the segment has been already sent out, the corresponding update is noted with a record on the SyncQueue. The write is subsequently carried out to VD.

At the destination–PM, every segment received during this phase is written on the respective VD and its id is placed at ReceiveQueue; the latter maintains the identifiers of all the received segments before the VM handover. ReceiveQueue is a key structure as it helps the local instance of LiveFS determine the segments that have not been received yet.

3) Pre-Copy Synchronization Phase: in this stage all the items at SyncQueue are dispatched to the destination–PM. Updates
that arrive for segments being part of SendQueue, are placed in SyncQueue (only if the just-written segment does not already exist in the latter).

The processing of segments continues until the number of elements in SyncQueue is less or equal to the handover-size; this represents a relaxation level from strict segment consistency before VM-handover.

4) CPU-Memory Transfer and Handover Phase: the hypervisor is instructed by the Migrations Daemon to perform the VM CPU and main-memory migration to the target PM. The segment identifiers found at SyncQueue are sent as a string to target PM. Next, the source PM’s Migration Worker informs the Migrations Daemon to perform the “handover” of the VM through the facilities of the hypervisor. In our implementation, we use Xen’s migrate command with its “--live” option.

5) Post Handover Phase: the source Migration Worker has to send the remaining segments to the destination–PM. This set termed need-to-be-sent, consists of all those “cold” segments of the VD ranked below the migration threshold as well as those found in SyncQueue. All VD segments except those in SendQueue make up the “cold” area of the virtual disk under shipment. The Migration Worker initiates the transfer of the segments in the need-to-be-sent set in an eager manner. Moreover, the Migration Worker of the original PM is ready to serve segment requests called on demand by the destination–PM.

A similar task to what we discuss above is performed by the LiveFS at the destination–PM. This LiveFS instance has to be aware of the VD segments that are not yet present in its local disk. The corresponding need-to-be-received set consists of all the segments in the VD except those in SendQueue as well as the segment ids dispatched by the source PM during phase 4. As soon as the Migration Worker at the destination–PM has noted a receipt of a segment, it removes the segment-id from the need-to-be-received set.

Any time an I/O targeting the VM occurs, the LiveFS has to first establish whether the sought segment has been already received. This can be readily determined with the help of the need-to-be-received set. Should the segment be already transported, the I/O proceeds unhindered to the local disk. Otherwise, the LiveFS asks for it on the fly (and possibly out of sequence) from the source PM. Read–I/Os to missing segments are synchronous since the target PM has to wait for the segment to arrive. Write–I/Os to received segments simply go through to the VD. On the other hand, write–I/Os to missing segments are stored in the writeQueue – an in-memory LiveFS structure that maintains segment-id, offset as well as content of the modification. By the time a segment ultimately arrives at destination–PM all the updates that refer to it (and are kept at writeQueue) have to be replayed.

IV. HANDLING OF ERRORS DURING MIGRATION

The Migrations Daemon does not only oversee the migration process but also monitors the liveness of the participating PMs at all times (with the use of “still-alive” polling messages). As senders maintain open connections with their receiver counterparts, it is also easy to determine the receivers’ status. If both PMs become incapacitated, then the Migrations Daemon detects the error and may decide to start the migration anew. Fig. 4 shows how matters progress timewise for both a sending and a receiving PM. In this timeline, we sketch possible errors that may occur at the LiveFS-level and outline ways to overcome such deficiencies. Regarding data–consistency, we follow the notion of logical clocks [20] for updates that happen on a VD under migration.

- Phase 1: If an error occurs in either sender or receiver, the I/O monitoring has to start anew as soon as the respective PMs become again operational.
- Phase 2: If the LiveFS of the sender fails, then the entire procedure has to start from phase 1 all over again. If on the other hand the receiver fails, the sender can help successfully restart the receiver from phase 2.
- Phases 3 and 4: should one of the two parties fail, the still–alive PM aids its counterpart to reestablish the content of its SendQueue ReceiveQueue. As soon as the sender populates its SyncQueue with the segment-ids of its SendQueue, phase 3 starts over again.
- Phase 5: if the receiver fails, the sender can selectively ship segments not passed over before the failure. Updates that have occurred at the receiver can be only facilitated through a logging mechanism. On the other hand should the sender fail, the receiver asks on demand for all missing segments that a re-established sender can now provide.

V. EVALUATION

During our evaluation, we measure the performance of LiveFS as we tune its migration parameters and we establish the operational overheads of our approach. LiveFS is implemented in C with pthreads and FUSE [17] framework v2.9.2. We migrate a VM between two PMs while a workload is being executed within the VM. We employ two separate workloads on different evaluation scenarios. The first workload is produced by the Bonnie++ benchmark [21] and the second by our own variation of AFS [22]. With Bonnie++, we examine how long it takes to perform the VM handover under heavy load, while with our AFS-like benchmark we assess the effect that the Monitoring Time Period parameter has on the average I/O-delay during phase 5. In our set up, the PMs are two Intel(R) Xeon Servers with 8GB of RAM, connected with a 1Gbps Ethernet–switch. The employed hypervisor is the Xen v4.0.1 while the VM under migration runs a Linux Debian v6.0 and is equipped with 512MB of RAM and a 6GB virtual disk image.

The effectiveness of our of VM live-migration approach is compared against the pre–copy and post–copy techniques. LiveFS can effectively emulate both of these approaches by properly setting its migration threshold and monitoring time period parameters. For the pre–copy operation, we set LiveFS’s migration threshold and monitoring time period to zero; for the post–copy approach, the length of the monitoring time period

Fig. 4. Timeline of phases for both sender and receiver PMs.
is set to zero and the migration threshold is set to its maximum value (the maximum value of an unsigned long integer).

- **Bonnie++ benchmark in LiveFS**: Bonnie++’s I/O footprint involves creation and deletion of files as well as a number of read and write operations. We execute the benchmark within the VM-under-migration and measure the migration completion time. In this experiment, the handover-size parameter ranges from 0 to 80 segments and the disk segment size is fixed at 64MB.

Fig. 5 shows that the LiveFS Total Migration Time remains unaffected. This is because the working set of Bonnie++ is fixed and the number of updates is less than or equal to the handover-size. When LiveFS is configured to operate similar to the pre-copy approach, any updates performed have to be continuously pushed to the destination machine. Hence, if the handover-size is small, the Total Migration Time increases dramatically. The decrease of Total Migration Time compared to the pre-copy approach is on average 30.1%, with a maximum value of 64%.

Fig. 6 shows how the VM-handover is affected by the handover-size and the monitoring time period length. As depicted, the handover time increases linearly along with the monitoring time period; also the VM-handover time is neither affected by the time LiveFS takes to send segments (in SyncQueue) that need synchronization nor by the time required by Xen to complete the live-migration of CPU and main memory state. In this setting and for all our experiments, the time required by LiveFS to go through to the end of phase 4 is dominated by the monitoring period (i.e., phase 1).

- **The AFS workload**: To monitor how LiveFS benefits from re-occurring data-access patterns, we implemented a synthetic workload inspired by the AFS benchmark. AFS benchmark constructs a directory tree, copies files in it, scans the directory recursively and gets its contents status. As a final step, the benchmark reads all the files and issues a make command. Each of the aforementioned operations corresponds to a number of Load Units [22]. A Load-Unit refers to the load placed on a server by a single workstation client. Hence, in order to emulate a multi-user environment, multiple threads are instantiated, each one performing its own set of operations for a predefined number of rounds $R$. The set of operations remains unchanged for each worker-thread in order to produce a recurring workload. In our implementation of the AFS benchmark, the client threads only read from the VD-image and they do not perform any updates. In this way, we are able to quantify the I/O-delay caused by fetching disk segments from the source PM immediately after the VM-handover.

In every execution round, each worker-thread accesses the same set of blocks within the file in question. Using our AFS benchmark, we examine the effectiveness of LiveFS in identifying a working set of segments and reducing the I/O-delay during phase 5. Here, the migration threshold is set to 1,000 accesses, the handover-size to 30 and we produce 200 worker-threads each one reading a 1MB block for a 1,000 times.

Our focus in this experiment is the effect of monitoring time period. This period reduces the I/O-delay on the destination machine during phase 5 of our approach. Figure 7 depicts how the I/O-delay decreases as we increase the monitoring time period. We represent the “post-copy” approach’s performance with 0 monitoring time period. The largest reduction occurs when the monitoring time period is set to 15 seconds. For this specific period length, the single most–accessed VD-segment scored above the migration threshold. This fact calls for the transfer of the segment in question to the target PM during phase 2. Therefore, the segment becomes available immediately after the VM-handover. The small increase in I/O-delay observed between 20 and 30 seconds (x-axis) is because not all accessed segments are dispatched to the target and the maximum delay for a few of those is high. As a result, a higher average I/O-delay is recorded. As soon as the monitoring time period nears and goes beyond the 70 seconds mark, all of
the segments in the working set of the worker–threads are sent over during phase 2. This is the main reason why we experience fairly stable I/O-delays at this range. The largest decrease on I/O-delay is monitored on monitoring time period 55 and at that point, the difference between the "post-copy" and our approach experiences 19% less I/O-delay.

- **LiveFS memory requirements**: LiveFS maintains a number of in-memory structures. During the evaluation of the prototype, the maximum amount of memory that LiveFS consumed has been noted to decrease as the segment size increases. In fact, we monitored that when the VD segment used is greater than 32MB, the memory consumption remains below 6KB. However, we note here that the memory consumption of the writeQueue depends entirely on the workload present. Further analysis on this matter is avoided due to page limitations.

VI. RELATED WORK

Process migration can be considered as preceding work to VM migration. A thorough survey can be found in [23]. Several approaches that employ the "stop-and-copy" paradigm have been introduced in [1], [2], [3]. In those, a VM would be suspended and its entire state would be moved to a destination node. Finally, its operation would commence on the new machine. Live migration of VMs is tackled in [4] where 3 distinct phases are introduced to help facilitate the efficient movement of CPU state and main memory pages. Despite the fact that downtime is greatly reduced, the total migration time remains high. In [5], an alternative approach is presented where only the CPU state is transferred before handover occurs. The dynamic self-ballooning mechanism is suggested as the means to deal with page faults and speed-up migration. The evolution of live-storage techniques is presented in [7] and the problem of remote disk migration is addressed. In our work, we attempt to avoid lengthy delays especially when multi-GByte VDs have to move in a shared-nothing environment. We accomplish this by monitoring I/O traffic to VD segments and adaptively handle recent modification of hot and cold segments. A lot of research has been conducted in database live migration [24], [25]. These approaches employ "pre-copy" migration methods on multi-tenant database nodes experiencing high load.

VII. CONCLUSIONS

We address the problem of efficiently transporting not only VMs but also their voluminous virtual disks (VDs) in an IaaS share-nothing infrastructure. We present the design and implementation of a novel and adaptive approach to VM Live Migration. Its main objectives are to rapidly move "hot" VD-segments across the network and enable rapid handover in the new PMs when the need arises. PMs have their own physical storage and run a VM hypervisor such as Xen. Our multi-phase approach is facilitated by an instance of LiveFS at each node. LiveFS is an abstraction layer between the hypervisor and the underlying local filesystem and features a number of tunable parameters that can help adapt the type of the migration (i.e., pre-/post-copy and/or strict/relaxed hot-segment consistency). In this way, the needs of clients can be more effectively addressed. Moreover, instances of LiveFS monitor I/O-traffic to segments of the VD-under migration so that recently-accessed segments can be shipped faster. Experimentation with our prototype using a number of workloads has shown that our approach can effectively complete the execution of VM live migration by up to 64% faster than "pre-copy" migration approaches. On workloads that present recurrent I/O-patterns, our approach can reduce its I/O-delay by up to 19% if compared with "post-copy" VM live-migration techniques.

Acknowledgements: this work has been partially supported by the Sucre and iMarine EU FP7 ICT projects.

REFERENCES