2017 IEEE 23rd International Conference on Parallel and Distributed Systems

A Fast, General Storage Replication Protocol for Active-Active Virtual Machine Fault Tolerance

Cheng Wang, Xusheng Chen, Zixu Wang, Youwei Zhu, Heming Cui
Department of Computer Science, University of Hong Kong
cwang2@cs.hku.hk, chenxus@hku.hk, layton@hku.hk, judehku@hku.hk, heming@cs.hku.hk

***Abstract*—Cloud computing enables more and more online services deployed in virtual machines (VMs), making fast VM fault tolerance particularly crucial. Unfortunately, despite much effort, achieving fast VM fault tolerance remains an open problem. A traditional way to provide VM fault tolerance is the active-passive approach, which frequently transfers tremendous updated states, including memory and storage, of a primary VM to a suspended secondary VM. The other emerging approach, namely the active-active approach, runs the secondary VM concurrently with the primary. Compared to active-passive, active-active is faster because it only performs the transfer when the externally visible states (e.g., network outputs) of the primary and secondary diverge. However, active-active aggravates the performance issue on I/O intensive workloads. In existing active-active systems, storage replication protocols hold updated storage states from both the primary and secondary on the secondary, incurring excessive I/O contention. For instance, both our evaluation and prior study show that a well-engineered active-active system, COLO, degrades the throughput of I/O intensive services by up to 61.6%.**

To tackle this open problem, this paper presents GANNET, a fast and general storage replication protocol for active-active VM fault tolerance systems. It greatly alleviates the I/O contention on the secondary's storage by efficiently buffering the updated disk states from both the primary and secondary VM in memory. GANNET carries a lightweight storage checkpoint algorithm to avoid consuming too much memory. GANNET is proved to be as reliable as existing storage replication protocols. We integrated GANNET into two popular active-active systems. Evaluation on six widely used services shows that GANNET incurred 15.9% overhead compared to the native executions and outperformed COLO's storage replication protocol by 1.2X$\sim$2.6X. GANNET's source code is available at `github.com/hku-systems/gannet`.

***Keywords*-Cloud Computing; Fault Tolerance; Virtual Machine; Storage Replication**

I. INTRODUCTION

The cloud computing paradigm enables a pervasive deployment of online services in virtualized infrastructures (e.g., Xen [1]). In a virtualized environment, online services run in virtual machines (VMs), and multiple VMs may often be consolidated [2] in a single physical host to enhance server utilization and reduce various operational costs. As a result, it is extremely crucial to make VMs tolerate hardware failures, and this fault tolerance must be fast so that it has little impact on the performance of services.

Unfortunately, despite much effort, achieving fast VM fault tolerance remains an open problem [3], [4], [5], [6], [7]. A traditional way to provide fault tolerance is the active-passive approach [4], [8]. It periodically checkpoints the updated execution states of a primary VM and propagates to a secondary VM. The secondary VM is passive in the sense that it is suspended until a primary failure passes control to it. The externally visible states (e.g., network outputs) of the primary need to be buffered until completion of a successful checkpoint to ensure *external consistency* [4]: primary and secondary have the same states and the external world will notice no interruption or inconsistency in the event of the primary failure. However, active-passive incurs high performance overhead because the release outputs of the services are greatly deferred due to a large amount of states to be transferred.

The other emerging approach, namely the active-active approach [9], [10], [11], executes the secondary VM concurrently with the primary. It is faster than active-passive because it only does a checkpoint when the externally visible states of the primary and secondary diverge. By doing so, it can immediately release the outputs to the external world as long as the outputs of two VMs remain identical. For example, COLO [10], a well-engineered active-active system developed by Huawei and Intel, can run almost as fast as the native executions (i.e., executions in an unreplicated VM) for services working purely in memory.

Although the active-active systems have greatly reduced the amount of transferred memory states, they aggravate the performance issue for online services when intensive I/O workloads exist. A key reason is that in existing storage replication protocols (e.g., COLO's), the secondary's storage device holds the updated disk states from the primary. Since the secondary VM is also active, there exists excessive disk I/O contention on the secondary. Therefore, the secondary VM runs much slower than the primary, causing much deferred comparison of network outputs. For instance, our evaluation shows that COLO decreases the throughput of I/O intensive services by up to 61.6%.

To tackle this open problem, this paper presents GANNET[1], a fast and general storage replication protocol for

[1]Northern gannet is a swift seabird living in pairs to tolerate bad weather.

978-1-5386-2129-5/17/31.00 ©2017 IEEE
DOI 10.1109/ICPADS.2017.00031

active-active VM fault tolerance systems. Our observation is that holding the updated disk states from the primary on the secondary's disk is overly-constrained. We can achieve external consistency by efficiently buffering both the primary and secondary's updated disk states on the secondary's memory, thus we can greatly alleviate I/O contention. During each checkpoint operation, to make the primary and secondary's disks identical, the secondary only needs to flush primary's disk states into the secondary storage.

Our new protocol comes with two practical challenges. The first one is that the size of the ever-growing in-memory buffer might exceed the host machine's capacity. The second challenge is that the amount of flushed disk states from the primary can still be enormous at checkpoints. To address these two challenges, we introduce a lightweight checkpoint algorithm. The algorithm periodically compares the disk contents modified by the primary and secondary VM, applies the identical modifications to the secondary storage and frees the memory space containing the modifications. By doing so, we not only prevent the in-memory buffer from growing too big, but also reduce the amount of disk flushes at checkpoints.

We implemented our protocol in KVM-QEMU [12] and integrated it into two popular active-active systems, COLO [10] and Synchronization Traffic Reduction (STR) [11]. We evaluated GANNET with six widely-used programs, including two database servers `PostgreSQL` [13] and `MySQL` [14], two disk I/O benchmarks `fio` [15] and `NPB-BTIO` [16], a key-value store library `Berkeley DB` [17], and a FTP server [18].

Our evaluation shows that

1) GANNET is general to active-active systems. It was easily integrated into COLO and STR with only 36 and 49 lines of code respectively.
2) GANNET is fast. Evaluation on six widely used services shows that GANNET incurred 15.9% overhead compared to the native executions and outperformed COLO's storage replication protocol by 1.2X$\sim$2.6X.
3) GANNET is robust to single point of VM failure.

The key contribution of this paper is a new protocol that makes the storage replication in active-active VM fault tolerance system fast. Other contributions include a comprehensive study on the performance of existing storage replication protocols in active-active systems.

In the remainder of this paper, §II introduces the background of virtual machine replication and storage replication. §III gives an overview of GANNET. §IV presents GANNET storage runtime protocol. §V describes the algorithm for optimizations. §VI shows the implementation details. §VII does evaluation. §VIII discusses related work, and §IX concludes.

II. BACKGROUND

This section introduces virtual machine replication and storage replication.

A. Virtual Machine Replication System

Currently there are mainly two approaches to providing fault tolerance for VM. The first one is the active-passive approach [4], [3], [8], in which the secondary VM is suspended until a primary VM failure passes control to it. It operates by checkpointing the entire execution state, including I/O devices, memory, and CPU, of the primary VM, and propagating to the secondary VM machine. To ensure the client sees a consistent view in the event of the primary failure, the response needs to be buffered until the next checkpoint operation completes. This approach consumes bandwidth due to a large amount of state to transfer and incurs high network latency because of the output buffering [4], [19], [20], [11], [10].

An alternative to the active-passive approach is the active-active approach [10], [11]. In this approach, the secondary VM runs concurrently with the primary and fed the same network inputs as the primary VM. Compared with the active-passive approach, the active-active approach has many advantages. First, it can transfer much less state during a checkpoint operation because the primary VM and secondary VM run the same code, receive the same network inputs, their states should largely be the same and do not need to be transferred. Second, the active-active design can offer much lower latency overhead. For instance, in COLO [10] if the outputs of the two virtual machines remain the same, the two virtual machines are considered identical and the output is released immediately.

B. Storage Replication

Traditional storage replication approaches [21], [22], [23], [24] assume the secondary only passively receives data from primary and focus on mirroring disk changes from primary to secondary. Therefore, these traditional storage replication approaches are not designed for active-active systems where the secondary is also actively running.

COLO [10] proposed a typical storage replication protocol [25] for active-active systems. This method replicates the storage device state by asynchronously forwarding the disk modifications in the primary VM to the secondary node and directly writing them to the secondary disk image. However, since the secondary VM is also actively running, it needs to access its secondary disk image as well. Due to this I/O contention between the forwarded disk modifications and the secondary VM disk access, the secondary VM can be slowed down severely. If the secondary VM runs much slower than the primary VM, there could be a large amount of divergent states to transfer during the checkpoint operation. Worse, this causes the primary VM in COLO to buffer its output for a longer time before releasing because it needs to wait for the output from the secondary, incurring high network latency. Our evaluation in Section VII shows that compared to the unreplicated system, COLO's throughput has dropped by 61.6% for I/O intensive programs.

III. Overview

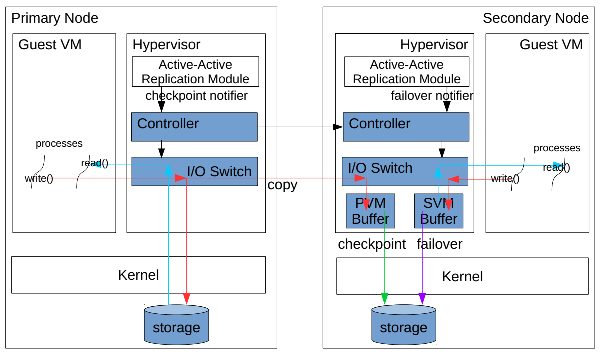

Figure 1: ***The*** **GANNET** ***Architecture.*** **GANNET components are shaded (and in blue).**

Figure 1 shows a GANNET instance deployed in an active-active virtual machine replication system. GANNET is integrated into the hypervisor because it needs to intercept the virtual I/O disk path of the disk requests from the virtual machine. It contains five main components, the controller which is responsible for the replication logic, the I/O switch that redirects the disk requests to different endpoints, and three endpoints: Primary Virtual Machine (PVM) buffer, Secondary Virtual Machine (SVM) buffer and the storage device.

The controller acts as an interface to the upper layer active-active replication module and controls the behavior of the I/O switch. It receives event notifications from the upper layer, communicates with the controller on the peer node, and instructs the I/O switch to take corresponding actions.

The I/O switch interposes on the disk requests from the virtual machine. Writes to disk from the primary VM are treated by the I/O switch as write-through: they are immediately applied to the primary disk image and asynchronously mirrored to the secondary. This approach provides two benefits. 1) writing directly to disk accurately accounts for the latency and throughput characteristics of the physical device. 2) it ensures the primary disk will reflect the crashed state of the externally visible VM at the time of failure. On the secondary side, at startup, the I/O switch creates two in-memory buffers, namely the PVM buffer and the SVM buffer. Upon receiving the mirrored disk write requests, the secondary I/O switch stores it in the PVM buffer. This approach eliminates the contention influence from the mirrored disk modifications. The secondary I/O switch buffers the write requests from the secondary VM as well because the modified disk contents need to be cleaned in the checkpoint to keep the primary and secondary storage in sync. Section IV-A and V describes the details about how I/O switch handles the requests.

The PVM buffer is used to store the mirrored write requests. It is applied to secondary disk image to keep the primary and secondary disk identical when a checkpoint is invoked. When the primary controller receives the checkpoint notification from the upper layer replication module, it informs the secondary controller of this message, asks the primary I/O switch to flush all the pending write requests on the connection and sends a commit message at the end to indicate it has finished forwarding all the primary write requests since the last checkpoint. When secondary controller receives the checkpoint notification, it sends a checkpoint command to the secondary I/O switch. The secondary I/O switch will wait for the commit message and then clean the SVM buffer and flush the PVM buffer into the secondary disk image.

The SVM buffer is used to hold the modified disk contents in the secondary VM. In the event of the primary failure, it is flushed into the disk image and discarded, which means the secondary I/O switch does not need to intercept the virtual I/O path any longer and all the disk requests from the secondary VM can directly manipulate on the secondary disk image afterwards. At checkpointing time, this buffer is cleaned. Section IV-B gives a correctness proof for this.

IV. Gannet Storage Runtime Protocol

A. Storage Runtime Protocol

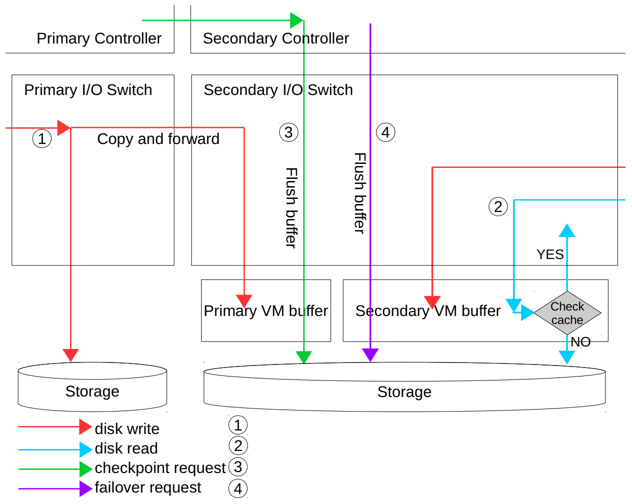

Figure 2: **GANNET** ***storage replication protocol.***

Write requests (labeled ① in Figure 2) from the primary VM are treated as write-through: they are immediately applied to the primary disk, and asynchronously mirrored to PVM buffer on the secondary. The secondary I/O switch holds the mirrored write request in PVM buffer. Specifically, it first reads the original content of the disk sector from the secondary storage, then manipulates the data according to the request, and finally stores the sector in the PVM buffer. For write requests from the secondary VM, the I/O switch reads the corresponding disk sector into SVM buffer and applies the write requests.

Read requests (labeled ②) from the primary VM will read the data directly from its own storage. For the read requests from the secondary VM, in order to avoid returning stale data, the disk manager will try to read the requested data from SVM buffer. If the requested data does not exist in SVM buffer, disk manager will then read the data from the secondary storage.

When the primary I/O switch receives the checkpoint notification (labeled ③), it flushes all the pending requests in the connection and sends a special commit message. When the secondary I/O switch receives this commit message, it cleans the SVM buffer and flushes the PVM buffer into the secondary storage to make the primary and secondary storage identical. Notice that all the checkpoint states should be well received by the secondary node first and then applied in a transactional (atomic) fashion, otherwise, a primary failure during the data transfer would leave the secondary VM in an inconsistent state. Therefore, the upper layer active-active replication module should only start GANNET's checkpoint operation when all the other states (e.g., memory and CPU) have been received successfully by the secondary node. Section V describes our optimization for accelerating this process.

Upon receiving the failover notification (labeled ④) from the secondary controller, the secondary I/O switch flushes the SVM buffer into the secondary storage. Although the secondary VM could continue to execute using the SVM buffer as an overlay on the physical disk, this would violate the disk semantics to the protected VM: if the secondary fails after activation but before data is completely flushed to disk, its on-disk state does not reflect the crashed state of the externally visible VM at the time of failure. Therefore, this flush action in the failover case should not return until the whole SVM buffer has been applied. If the secondary node fails, the primary controller simply asks the I/O switch to stop forwarding.

B. Fault-Tolerance Correctness

GANNET is designed to tolerate the same failure mode, i.e., fail-stop mode, as all the VM fault tolerance systems do [4], [10], [8], [26]. To ensure correctness, it is designed to satisfy two properties, external consistency and data durability. External consistency means after the secondary VM takes over after a failure of the primary, no externally visible state or data is lost and the external world will notice no interruption or inconsistency. Data durability means the storage device should reflect the crashed state of the externally visible VM at the time of the failure.

First of all, we provide a sketch of the proof of external consistency. If the primary fails while transferring states during a checkpoint operation, the secondary will receive an incomplete VM state, including the storage. Because the secondary will only apply the SVM buffer when the entire VM state has been received (IV-A), the secondary storage state is left consistent with all the other states of the secondary VM. Note that the the primary storage and secondary storage must have executed consistently to the external world since the last checkpoint, after the secondary node takes over, the secondary storage device will continue to behave consistently.

If the primary fails at the other moments, the external consistency is guaranteed as well because the primary storage and secondary storage must have behaved consistently from the last checkpoint, therefore the secondary node only needs to discard the buffered primary disk states and uses its own to keep the consistency to the clients.

Second, data durability property in GANNET is guaranteed by two mechanisms. On the primary node, writes to disk are write-through: they are immediately applied to the primary disk. This ensures that the primary storage device preserves data durability at all times. On the secondary side, before taking over, instead of using the SVM buffer as an overlay on the physical disk, the secondary will wait until the whole SVM buffer has been flushed to the storage. Afterwards, disk requests are directly reflected on the secondary storage. Hence, data durability is preserved on the secondary side as well.

V. GANNET OPTIMIZATION

There are two practical challenges imposed by our protocol. The first challenge is that PVM buffer and SVM buffer are growing bigger as the guest VM issues more disk write requests and may exceed the memory capacity of the host machine. The second challenge is the potential extended checkpoint duration for flushing PVM buffer into the secondary storage. To address these two challenges, we propose a lightweight storage checkpoint algorithm, as illustrated in algorithm 1. This algorithm works on the secondary I/O switch and is event driven. The core of this algorithm is to speculatively compare the contents in SVM buffer and PVM buffer and apply the identical modifications to the secondary storage.

The secondary I/O switch maintains PVM and SVM buffer as two arrays, $Sector_{PVM}$ and $Sector_{SVM}$. It also keeps two bitmaps, namely $DirtyBitmap_{PVM}$ and $DirtyBitmap_{SVM}$, for tracking the index of the modified sectors in PVM buffer and SVM buffer respectively.

Whenever the secondary I/O switch receives a disk write REQUEST($role$, $offset$, ...), where $offset$ is the starting position the request will work on and $role$ is the initiator of the request (either primary or secondary), it marks the corresponding sector dirty in the bitmap. Specifically, the

Algorithm 1 Lightweight Storage Checkpoint Algorithm

```
Secondary I/O Switch:

upon receiving REQUEST($role$, $offset$, ...) do
    $SectorIndex = offset$ / $SectorSize$
    $Apply(REQUEST)$
    $Set(DirtyBitmap_{role}[SectorIndex])$
end upon

upon receiving IDLE do
    $DBitmap_{inter} = DBitmap_{PVM}$ & $DBitmap_{SVM}$
    for $DirtyBit$ in $DBitmap_{inter}$ do
        $HValue_{PVM} = Hash(Sector_{PVM}[DirtyBit])$
        $HValue_{SVM} = Hash(Sector_{SVM}[DirtyBit])$
        if $HValue_{PVM} = HValue_{SVM}$ then
            $Flush(Sector_{PVM}[DirtyBit])$
            $Free(Sector_{PVM}[DirtyBit])$
            $Free(Sector_{SVM}[DirtyBit])$
            $Unset(DMap_{PVM}[DirtyBit])$
            $Unset(DMap_{SVM}[DirtyBit])$
        end if
    end for
end upon

upon receiving CHECKPOINT do
    for $DirtyBit$ in $DBitmap_{PVM}$ do
        $Flush(Sector_{PVM}[DirtyBit]$
    end for
    $Free(Sector_{PVM})$
    $Free(Sector_{SVM})$
    $Clean(DBitmap_{PVM})$
    $Clean(DBitmap_{SVM})$
end upon
```

I/O switch first calculates the sector index $SectorIndex$ according to $offset$. If this request comes from the primary, then $Set()$ the $SectorIndex$ bit in $DirtyBitmap_{PVM}$, otherwise, $Set()$ the $SectorIndex$ bit in $DirtyBitmap_{SVM}$.

When the VM is idle, i.e., it is not busy handling client requests, the secondary controller sends an IDLE notification to the I/O switch. Upon receiving this notification, the I/O switch performs a bitwise AND operation between $DirtyBitmap_{PVM}$ and $DirtyBitmap_{SVM}$ and gets $DirtyBitmap_{intersection}$. Then the I/O switch computes a hash for each sector found in $DirtyBitmap_{intersection}$. If the sector in the PVM buffer has the same hash value as the same-indexed sector in the SVM buffer, then we can safely flush this sector into the secondary storage and free the corresponding memory space in $Sector_{PVM}$ and $Sector_{SVM}$. Finally, the I/O switch $Unset$ this dirty bit in $DirtyBitmap_{PVM}$ and $DirtyBitmap_{SVM}$. Since the primary VM and secondary VM run the same code, receive the same network inputs, most of their modified disk contents in SVM buffer and PVM buffer should be identical and hence be flushed and freed in this event.

The last event occurs when the secondary disk I/O switch receives the $CHECKPOINT$ notification. In this case, the I/O switch only needs to flush each sector found in $DirtyBitmap_{PVM}$ into the secondary storage, free the corresponding memory occupied by $Sector_{PVM}$ and $Sector_{SVM}$, and finally $Clean$ the $DirtyBitmap_{PVM}$ and $DirtyBitmap_{SVM}$.

Through this algorithm, we not only prevent the buffer from consuming too much memory, but also reduce the amount of modifications to be applied at checkpoint. Our evaluation in Section VII-C shows that GANNET storage checkpoint duration is shortened by 51.9% with this optimization and is only 178.9% longer than COLO.

Although the above steps can already avoid occupying too much memory resource, it is still possible for the buffer to grow bigger than the host machine's capacity if there is no checkpoint involved for a long time. To prevent this rare case from happening, we set a fixed threshold for the size of the growing the buffer. If the size of the buffer exceeds this threshold, GANNET will proactively call the upper active-active replication module to invoke a checkpoint.

VI. Implementation Details

We implemented GANNET in KVM-QEMU [27] because of two reasons. First, KVM achieves nearly the same performance with the underlying physical machine [11]. Second, since COLO is built on top of KVM-QEMU, it is fair to compare its performance with GANNET.

A. Determining Virtual Machine Idle Status

The optimization technique introduced in Section V needs to determine the idle status to invoke the algorithm. To efficiently detect the VM is in idle status, we develop a simple, non-intrusive algorithm without modifying guest OS. This algorithm leverages the threading hierarchy of QEMU; all QEMU virtual threads (threads that emulate VCPUs) are spawned from the QEMU main process. GANNET runs an internal thread in QEMU to call `clock()` every 1ms and compare the increment of processor time with a threshold (default 1200, ticked by the guest OS's idle process in our hardware setting). If the increment is smaller than the threshold for five consecutive checkpointing, GANNET regards the guest VM as idle. Then the lightweight storage checkpoint algorithm is invoked.

B. Computing Dirty Sector Hashes Concurrently

We leveraged multi-core hardware and implemented a multi-threaded dirty sector hash computing mechanism for further accelerating the algorithm in Section V. The mechanism detects the number of CPU cores on local host and creates the same number of threads to compute hashes of dirty sector contents. One can leverage the emerging multi-GPU hardware to speedup our mechanism, and we leave it for future work. We used Google's CityHash [28] because it is fast and has low collision rate.

C. *Efficiently Propagating I/O stream between the primary and secondary*

COLO uses QEMU's embedded Network Block Device (NBD) server for forwarding the disk modification operations in the primary VM. The built-in NBD server on the secondary node serves each disk write request from the primary as write-through: the write request is immediately written to the secondary storage. Since GANNET does not need to apply each primary write request on the secondary storage, we discarded the use of NBD in COLO. GANNET used a TCP connection for forwarding the requests and a dedicated thread on the secondary node for caching the forwarded requests.

VII. EVALUATION

Our evaluation hosts were Dell R430 servers with Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. Each machine was equipped with a 40 Gbps NIC. The `ping` latency between two hosts was 84 μs. All client benchmarks and VMs were ran in these hosts. The guest VM ran Linux 3.13.0 with 16 GB memory and 16 VCPU.

We implemented GANNET in Linux KVM-QEMU [27] and integrated it into two popular active-active systems, COLO [10] and Synchronization Traffic Reduction (STR) [11]. We chose COLO because it is an industry-leading active-active system deployed in Huawei's cloud operating system FusionSphere [29].

We evaluated the same set of programs which have been tested by COLO [10] except Kernel Build and `WebBench` [30]. Kernel Build is not an online service. `WebBench` is a stateless web server which usually does not contain important in-memory executions states or storage and thus there is no need to provide fault tolerance to it. In addition to those programs from COLO, we have evaluated another wide range of programs, including `Berkeley DB`, a widely used key-value store library [17]; `MySQL`, an SQL database [14]; two widely used disk I/O benchmarks `NPB` [16] and `fio` [15]; and a FTP server [18].

We chose similar workloads which have been used in COLO. For `PostgreSQL`, we run the benchmark `pgbench` which is loosely based on TPC-B. For `FTP`, we tested the PUT and GET performance by transmitting a 300MB file. For `MySQL`, we used `SysBench` [31] to generate random update queries. For `Berkeley DB`, we used a popular benchmark bench3n [32], which does fine-grained, highly concurrent transactions.

The remaining of this section focuses on four questions:

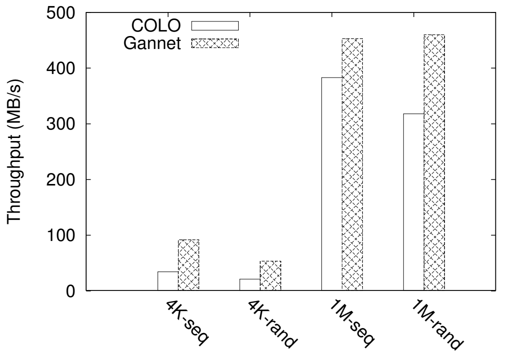

Figure 3: GANNET *write performance compared to COLO.* "seq" and "rand" are different I/O patterns, namely sequential and random. "4K" and "1M" are the unit in bytes for each I/O operation.

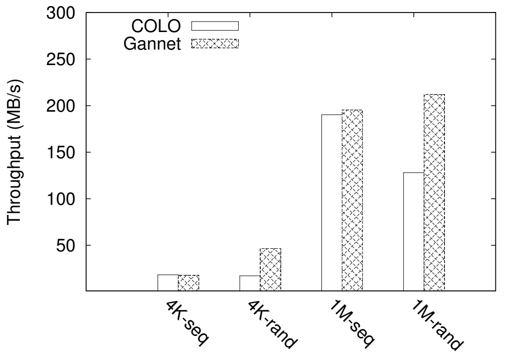

Figure 4: GANNET *read/write performance compared to COLO.*

§VII-A: How easy is GANNET to be integrated into existing active-active systems?
§VII-B: What is the performance overhead of running GANNET on real-world services and workloads? How faster is GANNET than existing storage replication protocols?
§VII-C: How effective is each GANNET's component?
§VII-D: How well does GANNET handle failover?

A. *Integrating* GANNET *into COLO and STR*

Since STR is not open source, we have implemented a version on our own and integrated GANNET into it.

GANNET is easily integrated into COLO and STR with only 36 and 49 lines of code respectively. We took advantage of the built-in mechanisms in QEMU (e.g., the copy-on-write disk driver [33]). Most of the extra code is for implementing the algorithm and optimization introduced in Section V and VI.

B. *Performance Overhead*

Figure 3 and 4 show the throughput comparison between COLO and GANNET under various workloads running the disk I/O benchmark `fio`. To stress the system, we spawned

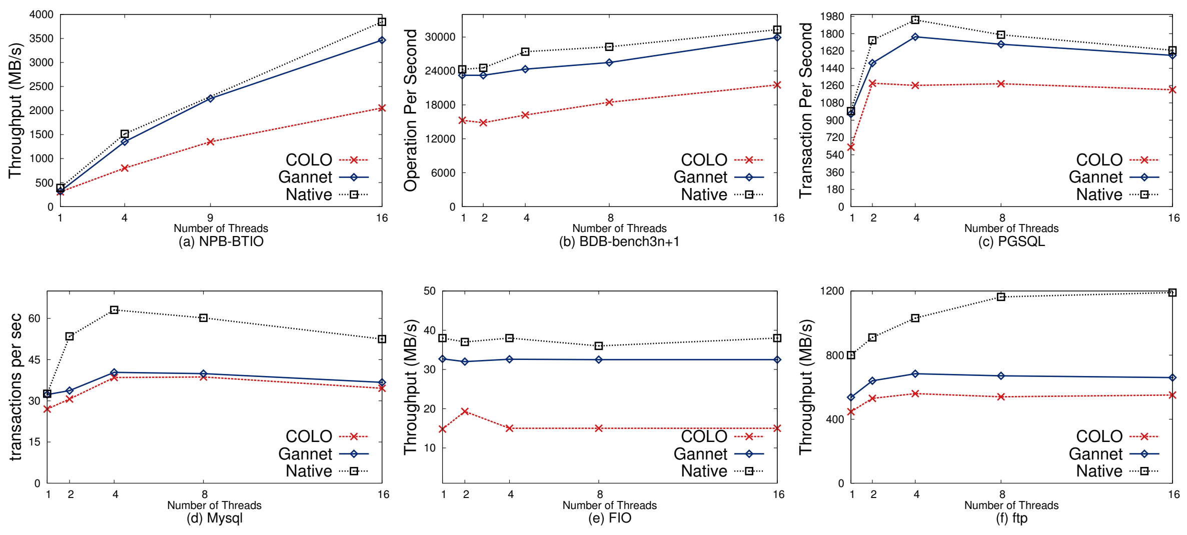

Figure 5: GANNET *performance compared to native executions and COLO*

Program	COLO Primary Lat.	COLO Secondary Lat.	GANNET Primary Lat.	GANNET Secondary Lat.
`NPB`	9.75ms	65.10ms	9.34ms	15.78ms
`MySQL`	18.21ms	53.97ms	19.66ms	27.37ms
`Berkeley DB`	0.23ms	0.33ms	0.29ms	0.30ms
`PostgreSQL`	0.23ms	1.53ms	0.20ms	0.28ms
`fio`	15.67ms	87.23ms	14.10ms	22.00ms
`FTP`	30.20ms	54.31ms	31.10ms	36.80ms

Table I: *Average latency per disk I/O request in COLO and* GANNET

16 threads in the benchmark. The improvement was less significant on mixed I/O tests than pure writes because disk reads do not incur any I/O contention on the secondary.

Figure 5 shows GANNET's throughput compared to COLO and unreplicated executions. Figure 6 shows GANNET's throughput compared to STR. We only list the `NPB-BTIO` application from the `NPB` benchmarks because all the other `NPB` applications work purely in memory and they incurred no performance difference between GANNET and COLO. The `NPB-BTIO` application requires a square number of processes for the execution, so we only collected the 1,4,9,16-thread results in Figure 5(a).

The results in Figure 5 show that GANNET scales better than COLO as the number of threads increases. With more threads issuing I/O requests concurrently, COLO's performance is limited by more excessive contention on the secondary storage device. On the other hand, GANNET fully exploits the concurrency since it has eliminated the I/O contention.

Overall, GANNET performed almost as well as the native executions on most of the programs except for `MySQL` and `FTP`. The network packet of `MySQL` contains physical timestamps, therefore the outputs of the two virtual machines are very likely to diverge, incurring high checkpoint frequency. In this case, the high checkpoint frequency dominates the overhead, rather than the storage replication. For `FTP`, we observed an average checkpoint interval of 605ms and hence the performance overhead mainly comes from the high checkpoint frequency as well.

To understand why GANNET outperforms COLO, we measured the average latency of each disk I/O request on the primary and secondary node. Table I shows these statistics. The secondary in COLO takes 318.2% longer to finish a disk request on average than the primary. Since the response to every incoming request can be returned to the client only when the slower of the primary and secondary successfully processes the request, the performance overhead becomes significant. On the other hand, while GANNET and COLO's primary process the I/O request at the same speed, the secondary in GANNET takes 53.8% shorter time to serve a disk request, which makes GANNET's throughput outperforms COLO.

Figure 6 shows that we achieved similar results with STR. However, we observed some differences. For example, `MySQL` achieved higher throughput. As discussed above, this is because COLO invokes the checkpoint frequently due to the divergence of the outputs, while STR's checkpoint frequency is not determined by the similarity of the outputs [11].

Because and COLO and STR show similar results, we only focus on comparing GANNET and COLO in the following subsections.

C. *Effectiveness of* GANNET *Lightweight Checkpoint Algorithm*

We ran the same performance benchmark as in Section VII-B and collected the storage checkpoint time to measure the effectiveness of our lightweight checkpoint algorithm. Table II shows the storage checkpoint duration in GANNET and COLO. This time varies because it depends on how I/O intensive the program is and the frequency of the checkpoint. For example, when running with 16 concurrent client requests, `MySQL` invokes the checkpoint so frequently that only a small amount of accumulated storage states need to be synchronized in the checkpoint.

Benefiting from the algorithm introduced in Section V, GANNET storage checkpoint duration is shortened by 51.9% and is only 178.9% longer than COLO. Since GANNET outperforms COLO a lot in the normal execution, this overhead is acceptable.

Program	**COLO ck.**	**GANNET ck. w/ opt.**	**GANNET ck. w/o opt.**
`NPB`	174.63ms	250.24ms	640.24ms
`MySQL`	121.90ms	213.76ms	740.00ms
`Berkeley DB`	198.46ms	556.37ms	692.10ms
`PostgreSQL`	38.46ms	113.21ms	679.81ms
`fio`	2035.19ms	2647.68ms	2670.18ms
`FTP`	37.86ms	246.25ms	1023.89ms

Table II: ***Effects of* GANNET*'s lightweight checkpoint algorithm.* "GANNET ck. w/o opt" means GANNET storage checkpoint duration without the optimization.**

D. *Handling Failover*

We measured the performance of GANNET when various failure happens. We manually killed primary and secondary in each experiment with a `PostgreSQL` server running in guest VM and monitored the real time throughput from its clients. When either the primary or the secondary was killed, `PostgreSQL` run as fast as the native execution because it is unreplicated.

Figure 7 shows the fluctuation of the throughput when we killed the leader. When the primary fails, the secondary first needs to detect the primary failure, then flushes the SVM buffer into the secondary storage device and finally resumes execution. This took about 360ms.

VIII. RELATED WORK

Virtual Machine Fault Tolerance. VM replication enables application-agnostic, non-stop service, surviving hardware fail-stop failures.

Log-replay [34], [35], [36], [37], [38] records all input and non-deterministic events of the primary machine so that

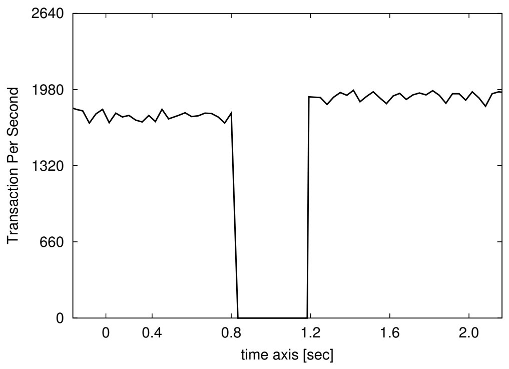

Figure 7: GANNET ***throughput on primary's failure.***

it can replay them deterministically on the backup node in case the primary machine fails. While this solution provides high efficacy to uni-processor virtual machines, its adaption to multi-core CPU environment is cumbersome, because it requires determining and reproducing the exact order in which CPU cores access the shared memory. It has been shown that this approach imposes superlinear performance degradation with the number of virtual CPUs on several workloads when applied to multi-core VM setups.

Active-passive checkpointing [4], [8] is then proposed to address the excessive overhead in log-replay. In this approach, the secondary is passive in the sense that the secondary VM is only activated after failover, when a hardware failure is detected in the primary node. Active-passive checkpointing operates by periodically checkpointing the entire execution state of the primary VM and propagating to the secondary machine. Additional optimizations [20], [11] are conducted to reduce the overhead of memory checkpoints, however they still suffered from extra network latency due to output packet buffering and the overhead of frequent checkpointing.

Active-active checkpointing [10], [11] is an alternative to active-passive design, in which the secondary VM concurrently with the primary. This design provide more flexibility. Lu et al. [11] reduced the checkpoint time by comparing the memory states of the primary and secondary VM and only transferring the divergent memory pages at checkpoint. COLO [10] improved the network latency by comparing the network outputs from the VMs and release the packets to the external world as long as they are the identical.

Storage Replication. Storage replication is important because hardware disk usually contains critical data whose damage can cause great losses. Storage replication can be done in different layers, including application level [39], [21], block level [22], hardware level [24], and file level [23], [40].

All these traditional works are orthogonal to GANNET. They focus on replicating disk data from primary to secondary, where the secondary only passively receives data

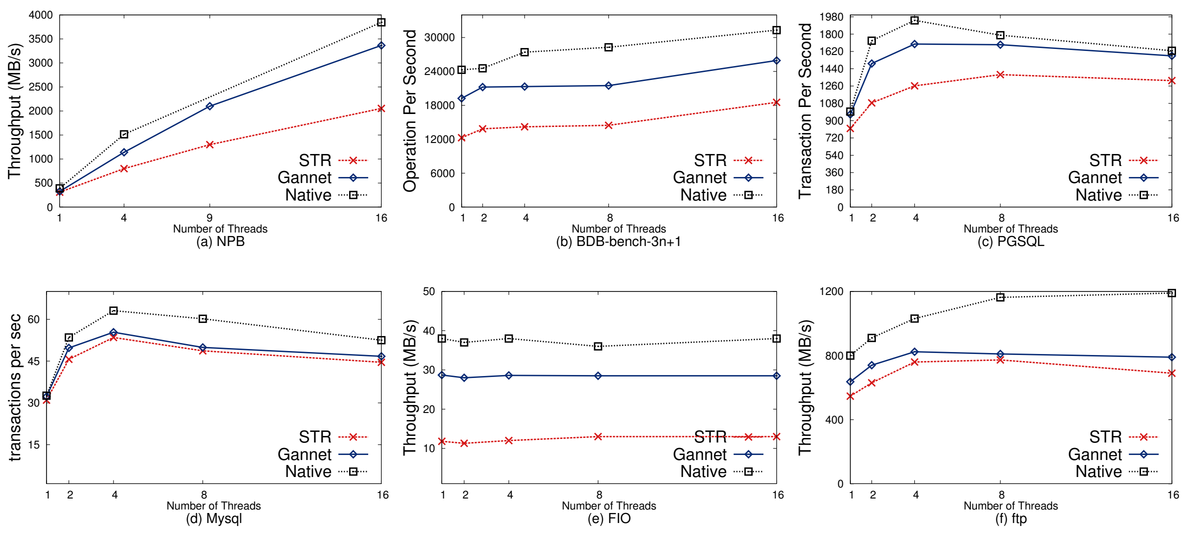

Figure 6: Gannet *performance compared to native executions and STR*

from primary. However, Gannet aims to remove the contention on an active-active setting where the primary and secondary make changes simultaneously and the secondary's disk contents need to be synced with the primary at each checkpoint.

IX. Conclusion

We have presented Gannet, a novel protocol that makes the storage replication in active-active virtual machine fault tolerance systems fast. We have integrated Gannet into two popular active-active systems and evaluation on six I/O intensive programs shows Gannet runs several times faster than the existing storage replication protocol and almost achieves the same performance as the native system. Gannet has the potential to greatly improve the reliability and performance of real-world online services deployed in clouds.

Acknowledgment

This paper is funded in part by a research grant from the Huawei Innovation Research Program (HIRP) 2017, HK RGC ECS (No. 27200916), HK RGC GRF (No. 17207117), and a Croucher innovation award.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 164–177.

[2] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, 2005.

[3] "RDMA migration and rdma fault tolerance for QEMU," http://www.linux-kvm.org/images/0/09/Kvm-forum-2013-rdma.pdf.

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. San Francisco, 2008, pp. 161–174.

[5] B. Gerofi and Y. Ishikawa, "Rdma based replication of multiprocessor virtual machines over high-performance interconnects," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, ser. CLUSTER '11, 2011.

[6] M. Lu and T.-c. Chiueh, "Speculative memory state transfer for active-active fault tolerance," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12, 2012.

[7] V. A. Sartakov and R. Kapitza, "Multi-site synchronous vm replication for persistent systems with asymmetric read/write latencies."

[8] Y. Du and H. Yu, "Paratus: Instantaneous failover via virtual machine replication," in *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*. IEEE, 2009, pp. 307–312.

[9] S. Sharma, J. Chen, W. Li, K. Gopalan, and T.-c. Chiueh, "Duplex: A reusable fault tolerance extension framework for network access devices," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*. IEEE, 2003, pp. 501–510.

[10] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "Colo: Coarse-grained lock-stepping virtual machines for non-stop service," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, 2013.

[11] M. Lu and T.-c. Chiueh, "Fast memory state synchronization for virtualization-based fault tolerance," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 534–543.

[12] http://www.qemu.org.

[13] "Postgresql," https://www.postgresql.org, 2012.

[14] "MySQL Database," http://www.mysql.com/, 2014.

[15] J. Axboe, "Fio-flexible io tester," *URL http://freemeat.net/projects/fio*, 2014.

[16] http://www.nas.nasa.gov/publications/npb.html.

[17] http://www.sleepycat.com.

[18] "Very secure ftp daemon (vsftpd)." http://www.nlm.nih.gov/mesh/jablonski/syndrome_title.html., 2014.

[19] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield, "Remusdb: Transparent high availability for database systems," *The VLDB Journal*, vol. 22, no. 1, pp. 29–45, 2013.

[20] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the performance of hypervisor-based fault tolerance," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[21] "Redis Replication," https://redis.io/topics/replication.

[22] L. Ellenberg, "Drbd 9 and device-mapper: Linux block level storage replication," in *Proceedings of the 15th International Linux System Technology Conference*, 2008.

[23] "The Hadoop Distributed File System," http://hadoop.apache.org/hdfs/.

[24] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks," *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988.

[25] "QEMU file system block replication," https://github.com/qemu/qemu/blob/master/docs/block-replication.txt.

[26] D. Shah and S. Lokray, "Xsft: Next generation ha," Technical Report, Symantec Research Labs, 2008, http://engweb. ges. symantec. com/cuttingedge/2007/presentations/SHAH1003 rev0. ppt, Tech. Rep., 2008.

[27] http://www.linux-kvm.org/.

[28] https://github.com/google/cityhash.

[29] "Huawei FusionSphere." www.huawei.com/ilink/enenterprise/download/HW_193449, 2014.

[30] "Web bench." http://home.tiscali.cz/cz210552/webbench.html, 2014.

[31] "SysBench: a system performance benchmark," http://sysbench.sourceforge.net, 2004.

[32] http://libdb.wordpress.com/3n1/.

[33] "The qcow2 image format," https://people.gnome.org/~markmc/qcow-image-format.html.

[34] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[35] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *SIGOPS Oper. Syst. Rev.*, Dec. 2010.

[36] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen, "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002, pp. 211–224.

[37] G. Altekar and I. Stoica, "ODR: output-deterministic replay for multicore debugging," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009, pp. 193–206.

[38] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: global comprehension for distributed replay," in *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

[39] "MySQL Replication," https://dev.mysql.com/doc/refman/5.0/en/replication.html.

[40] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Oct. 2003.