AutoReplica: Automatic Data Replica Manager in Distributed Caching and Data Processing Systems

Zhengyu Yang*, Jiayin Wang†, David Evans‡, and Ningfang Mi*
* Dept. of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
‡Dept. of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
‡ Samsung Semiconductor Inc., Storage Software Group, San Diego, CA 92121

***Abstract*—Nowadays, replication technique is widely used in data center storage systems for large scale Cyber-physical Systems (CPS) to prevent data loss. However, side-effect of replication is mainly the overhead of extra network and I/O traffics, which inevitably downgrades the overall I/O performance of the cluster. To effectively balance the trade-off between I/O performance and fault tolerance, in this paper, we propose a complete solution called "AutoReplica" – a replica manager in distributed caching and data processing systems with SSD-HDD tier storages. In detail, AutoReplica utilizes the remote SSDs (connected by high speed fibers) to replicate local SSD caches to protect data. In order to conduct load balancing among nodes and reduce the network overhead, we propose three approaches (i.e., ring, network, and multiple-SLA network) to automatically setup the cross-node replica structure with the consideration of network traffic, I/O speed and SLAs. To improve the performance during migrations triggered by load balance and failure recovery, we propose the a migrate-on-write technique called "fusion cache" to seamlessly migrate and prefetch among local and remote replicas without pausing the subsystem. Moreover, AutoReplica can also recover from different failure scenarios, while limits the performance downgrading degree. Lastly, AutoReplica supports parallel prefetching from multiple nodes with a new dynamic optimizing streaming technique to improve I/O performance. We are currently in the process of implementing AutoReplica to be easily plugged into commonly used distributed caching systems, and solidifying our design and implementation details.**

***Keywords*—Replica, Backup, Fault Tolerance, Device Failure Recovery, Distributed Storage System, Parallel I/O, SLA, Cache and Replacement Policy, Cluster Migration, VM Crash, Consistency, Atomicity**

I. Introduction

With the rise of Cloud Computing and Internet of Things, as an enabling technology, Cyber-physical systems (CPS) is increasingly reaching almost everywhere nowadays [1]. As a fundamental infrastructure of parallel and distributed computing for large scale CPS design, distributed data process and storage system is an important CPS components [2]. In those distributed systems, replication technique – a process of synchronizing data across multiple storage nodes – is often used to provides redundancy and increases data availability from the loss of a single storage node [3], [4].

However, there is a problem related to the replication overhead and tiering storage I/O performance. In a SSD-HDD tier storage based cluster, SSDs are usually used as the write back cache to improve to I/O speed, since writing through to HDD will dramatically slow down the I/O path. However, there is only one update-to-date copy cached in the SSD under this policy, which is not acceptable, However, as revealed in study [5], SSD is relatively not a "safe destination" though it can preserve the data after power off, and thus having only one up-to-date copy on SSD is not acceptable for high SLA (Service-Level Agreement) demand use cases such as bank, stock market, and military databases. Therefore, the crucial problem is *"Where to store replicas of those datasets cached in the SSD while not downgrading the performance?"*

Motivated by this, we propose a complete solution called "AutoReplica", which is a data replica manager designed for distributed caching and data processing systems using SSD-HDD tier storage systems. AutoReplica maintains replicas of local SSD cache in the remote SSD(s) connected by high speed fibers, since the access speed of remote SSDs can be way faster than local HDD's. AutoReplica also has three approaches (i.e., ring, network, and multiple-SLA network) to automatically build the cross-node replica structure. We also develop a lazy migrate-on-write technique called "fusion cache" that can conduct seamlessly online migration operation to balance loads among nodes, instead of pausing the subsystem and copying the entire dataset from one node to the other. AutoReplica can efficiently recover from different disaster scenarios (covers VM crash, device failures and communication failures) with limited and controllable performance downgrades. Finally, AutoReplica supports parallel prefetching from both primary node and replica node(s) with a new dynamic optimizing streaming technique to improve I/O performance. We are currently in the process of implementing AutoReplica and solidifying our design and implementation details, and we are also working hard to make AutoReplica to be easily plugged into other popular distributed caching and data processing systems.

The remainder of this paper is organized as follows. Sec. II presents the topological structure of datacenter cluster of AutoReplica. Sec. III introduces AutoReplica's cache and replacement policy, including the new "fusion cache" technique. Sec. IV describes recovery policy under different scenarios. Sec. V discusses the parallel prefetching scheme. Sec. VII talks the related work. Finally, we summarize the paper in Sec. VIII.

II. Topological Structure Of Datacenter

We first introduce the datacenter cluster topological structure of AutoReplica. As illustrated in Fig. 1, there are multiple nodes in the cluster, and each node is a physical host running multiple virtual machines (VMs) upper on either type 1 or type 2 hypervisors.In our prototype, we use VMware's ESXi [6] to host VMs, which is a type 1 implementation. Inside each node,

This work was completed during Zhengyu Yang and Jiayin Wang's internship at Samsung Semiconductor Inc., and was partially supported by National Science Foundation Career Award CNS-1452751 and AFOSR grant FA9550-14-1-0160.

978-1-5090-5252-3/16$31.00

there are two tiers of storage devices: SSD tier and HDD tier. The former tier is used as the cache and the latter tier is used as the backend storage. Each storage tier contains one or more SSDs or HDDs, respectively. RAID mode disks can also be adopted in each tier. SSD and HDD tiers in each node are shared by VMs and managed by the hypervisor.

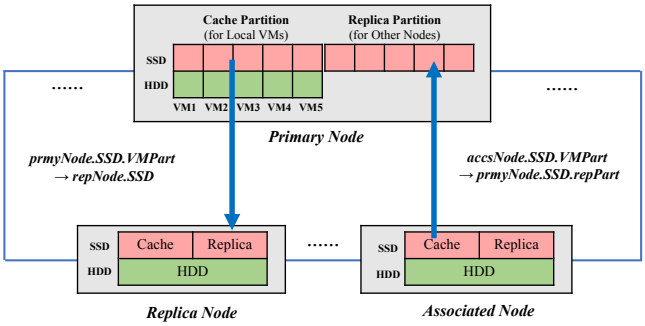

Fig. 1: An example of the structure of AutoReplica's datacenter.

Inside the SSD tier, there are two partitions: "Cache Partition" (for local VMs), and "Replica Partition" (for storing replica datasets from other nodes). As mentioned, AutoReplica uses write back cache policy to maximize I/O performance, since writing through to HDD will slow down the I/O path. However, SSD is relatively vulnerable and not cannot be equally trusted as a "safe destination" like HDD, though SSD can preserve the data after power off. Therefore, AutoReplica maintains additional replicas in the remote SSDs to prepare for recoveries for failures. In fact, we still can use local HDD as the second replica device for those extremely high SLA nodes, which will be discussed in Sec. II-A. Based on these facts, we propose three approaches to setup the topological structure of the datacenter clusters, focusing on *"how to select replica nodes?"*, *"how many replicas nodes do we need?"*, and *"how to assign replicas?"*.

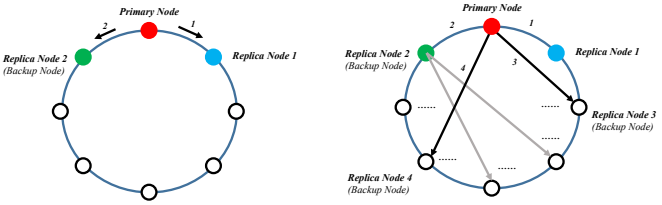

Fig. 2: Examples of (a) Ring and (b) Network approaches.

A. Ring Approach

Our first approach is a directed logical "Ring" structure, which can be either user-defined or system-defined. A system-defined ring is based on geographic distance parameters (e.g., I/O latency and network delay). As shown in Fig. 2(a), this logical ring defines an order of preference between the primary and replica nodes. Caching is performed using the local SSD with a copy replicated to another node in the cluster. Each node consists of two neighbors, storing replicas on both/one of them. The node walks in the ring until it can find a replica to use if unsuccessful during the process of building the ring cluster. Once it has a replica, it can begin to write caching independently of what the other nodes are doing.

B. Network Approach

As a "linear" approach, the "Ring" structure has a drawback during searching and building replicas, since it has only one or two directions (e.g., previous and next neighbors). In order to improve system robustness and flexibility, we further proposed the "network" approach – a symmetric or asymmetric network, see Fig. 2(b), which is based on each node's preference ranking list of all its connected nodes (i.e., not limited to two nodes).

From \ To	1	2	3	4
1	-	1	3	2
2	3	-	2	1
3	3	2	-	1
4	1	3	2	-

TABLE I: Example of the "distance matrix" used in Network approach, which shows the ranking of each path.

In our implementation, we introduce a "distance matrix" (an example is shown in Table I) to maintain each node's preference list ranked by a customized "score" calculated based on multiple parameters such as network delay, I/O access speed, space/throughput utilization ratio, etc. This matrix is periodically updated through runtime measurement (e.g., heartbeat). The main procedure of how to assign the replica nodes for each node is as following: Each node simply lookup the matrix and selects its "closest" node as its replica node if possible. To avoid the "starvation" case that lots of nodes are choosing one single node or a small range of nodes as their replica nodes (i.e., the "starvation and overheat" problem), AutoReplica also limits the maximum replica number per node. Lastly, each node can also have more than one replica node.

C. Multiple-SLA Network Approach

In real environment, rather than treating different nodes equally, the administration is often required to differentiate the quality of service based their SLAs (and even workload characteristics). To support this requirement, we further develop the "multiple-SLA network" approach to allow each node to have more than one replica node with different configurations based on a replica configuration decision table.

Case	Workload		Destination				#Reps.
	SLA	Temp.	SSD_P	SSD_{R1}	SSD_{R2}	HDD_P	
1		✓	✓	✓			1
2			✓		✓		1
3	✓	✓	✓	✓	(✓)		1(2)
4	✓		✓	✓		(✓)	1(2)

TABLE II: Replica configuration table for mulitple SLAs.

An example of replica configuration table is shown in Table II, where SSD_P, SSD_{R1}, SSD_{R2} and HDD_P stand for the SSD tier of the primary node, the SSD tier of the first replica node, the SSD tier of the second replica node, and the HDD tier of the primary node, respectively. It also considers:

- **SLA:** Related with importance of each node. Multiple SLAs is supported by utilizing multiple replica configurations. Although our example has only two degrees: "important" and "not important", AutoReplica supports more fine-grained degrees (even online-varying) SLAs.
- **Temperature:** Similar to [7], we use "data temperature" as an indicator to classify data into two categories according

to their access frequency: "hot data" has a frequent access pattern, and "cold data" is occasionally queried.

Although local HDD ($prmyNode.HDD$) can also be used as a replica destination (case 4 in table II), AutoReplica will reduce the priorities of those write-to-HDD replica operations in order not to affect those SSD-to-HDD write back and HDD-to-SDD fetch operations in the I/O path. Technique [8] is adopted to improve the performance of the write-to-HDD queue in the I/O path.

III. Cache and Replacement Policies

AutoReplica uses write back cache policy. In detail, when the SSD tier (i.e., cache) is full, SSD-to-HDD eviction operations will be triggered in the primary node ($prmyNode$), while in the replica node ($repNode$), the corresponding data set will simply be removed from the SSD of $repNode$ without any additional I/O operations to HDDs of $repNode$. Alg. 3 shows a two-replica-node implementation. In fact, it can have any number of SSD replica nodes to support more fine-grained SLAs. AutoReplica switches between two modes, namely "runtime mode" (line 19) and "online migration mode" (line 3 to 11) by periodically checks the $migTrigger$ condition (which considers runtime states such as load balancing and bandwidth utilization). If $migTrigger$ returns true, AutoReplica will select the "$overheat$" replica node (line 6) and is replaced with the next available replica node (line 7). After that, AutoReplica begins to run under the "migration mode" (line 14). If the "migrate out" replica node ($repNodeOut$) has no more "out-of-date" replica datasets (i.e., the migration is done), AutoReplica then stops the migration by setting $migModeFlag$ to $false$ (line 16), and goes back to the runtime mode (line 19). We describe the details of the runtime mode and the migration mode cache policy in Sec. III-A and III-B.

Main Procedure of AutoReplica's Cache Policy

Note: (1) $prmyNode$ is the primary node. $repNodeRem$ and $repNodeOut$ are the original two replica node. $repNodeIn$ is the destination of migration which replaces $replicaNodeOut$.
(2) $write(inputData, inputDirtyFlag)$: A function that writes the *inputData* into the device. If the device is "SSD", then this function sets *dirtyFlag* of the *inputData* as *True*. If the device is "HDD", then $inputDirtyFlag$ can be ignored.
(3) $migrateTrigger()$: A function returns *True* if the subsystem needs to migrate due to imbalance load.
(4) T_W: window size (i.e., frequency) of migration condition checking.

```
Procedure cache (prmyNode, repNode1, repNode2)
1    migModeFlag = Flase
2    for each new I/O request newData ∈ IOStream on prmyNode do
3        /* check load balance */
4        if currTime mod T_W == 0 and migModeFlag ≠ True and migTrigger() ≠ False then
5            migModeFlag = True
6            repNodeOut = selectOverheatNode(repNode1, repNode2)
7            repNodeIn = selectNextReplica()
8            if repNodeIn == repNode1 then
9                repNodeRem = repNode2
10           else
11               repNodeRem = repNode1
12       /* online migrate mode cache policy */
13       if migModeFlag == True then
14           cacheOnlineMigrateMode(newData, prmyNode, repNodeRem, repNodeOut, repNodeIn)
15           if repNodeOut.SSD.sizeOfRepForPrmy() == 0 then
16               migModeFlag = False
17       /* runtime mode cache policy */
18       else
19           cacheRuntimeMode(newData, prmyNode, repNode1, repNode2)
20   return
```

Fig. 3: AutoReplica's cache policy.

A. Runtime Mode Cache Policy

Under the runtime mode, AutoReplica searches the new I/O request in the local SSD VM partition (i.e., $prmyNode.SSD$). If it returns a cache hit, then AutoReplica either fetches it from the $prmyNode.SSD$ for a read I/O, or updates the new data to its existing cached copies in $prmyNode.SSD$ and the correspond replica node(s) for a write I/O. For the cache miss case, AutoReplica first selects a victim to evict from the $prmyNode.SSD$ and all its $repNode.SSD$(s), and only write updated (with "dirty" flag) evicted dataset into $prmyNode.HDD$. AutoReplica supports other replacement algorithms to implement the victim selection function, such as Multi-LRU [9], CLOCK [10], ARC [11], CAR [12], VFRM [13], Glb-VFRM [7] and GREM [14]. AutoReplica then inserts the new dataset into both $prmyNode$ and all its $repNode.SSD$(s). If it is a read I/O, AutoReplica fetches it from $prmyNode.HDD$ to SSDs of the $prmyNode.HDD$ and SSDs of all its $repNode(s)$. Additionally, it also returns the fetched $cacheData$ to the user buffer in the memory. If it is a write I/O, AutoReplica writes it to SSDs of $prmyNode$ and all its $repNode$(s) with $dirtyFlag$ as "dirty", since it is updated new data.

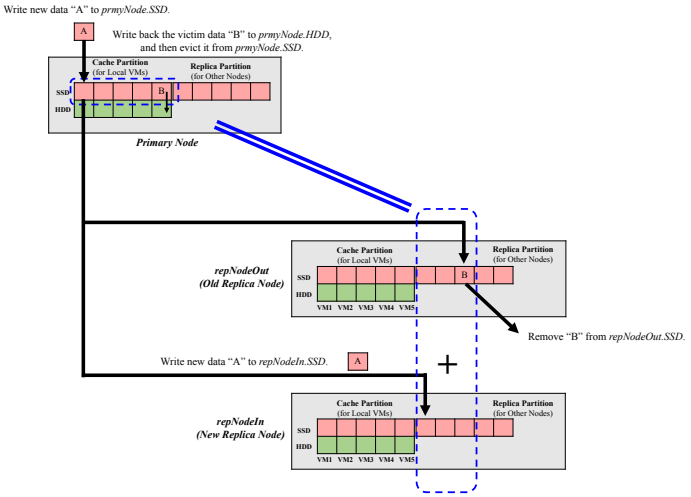

Fig. 4: Example of Online Migration Cache Policy.

B. Online Migration Mode Cache Policy

AutoReplica uses a cost-efficient migrate-on-write scheme called "fusion cache" to migrate replicas from one $repNode$ to the other. The main idea is instead of pausing the subsystem, and copying all existing replicas from the old node ($repNodeOut$) to the new node ($repNodeIn$), regardless of whether these data pieces are necessary or not, "fusion cache" keeps the subsystem alive and only writes new incoming datasets to $repNodeIn$ and keeps those "unchanged" cached data on $repNodeOut$. Eventually, $repNodeIn$ will replace $repNodeOut$. In other words, AutoReplica mirrors the $prmyNode.SSD$ by using the *unibody* of $repNodeOut$ and $repNodeIn$ to save lots of bandwidth. An example is depicted in Fig. 4, where only one replica node is needed to be "migrated out" and one new replica node is needed to take over those cached datasets. When a new dataset "A" comes to the $prmyNode$, AutoReplica first evicts the victim "B" on $prmyNode.SSD$ if the cache is full, and writes "A" to its $prmyNode.HDD$ and the new $repNodeIn.SSD$. Meantime, AutoReplica remove the "B" from the old replica node $repNodeOut.SSD$. In fact, this policy also works in the case where exist more replica nodes. Furthermore, in our implementation, users can also configure the laziness degree of migration from the traditional pro-active mode to "fusion cache" style lazy mode.

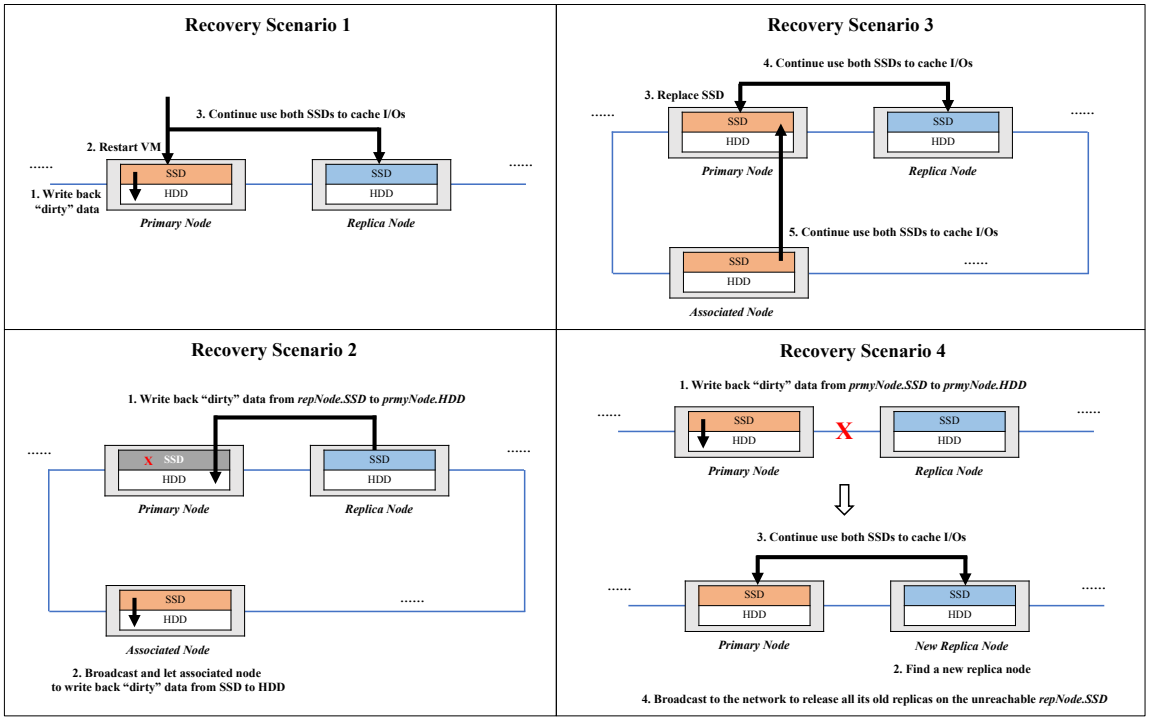

Fig. 5: Example of AutoReplica's recovery scenario.

IV. Recovery Policy

AutoReplica maintains additional replicas in the remote SSDs to prepare for recoveries for different failures. Specifically, it has different procedures to recover from failures covering the following four scenarios:

A. VM Crash on Primary Node

A very common failure is that a VM crash on the primary node. As shown Fig. 5(1), AutoReplica first closes out the VMDK. It then writes back "dirty" datasets from $prmyNode.SSD$ to $prmyNode.HDD$, and keeps them in $repNode.SSD$ with "nondirty" flag. After that, it restarts crashed VM on $prmyNode$, and continues to forward incoming I/O requests on both $prmyNode.SSD$ and $repNode.SSD$.

B. Primary Node Cache Device Failure

A primary node cache storage device failure will result in its inability to continue to write caching. As shown in Fig. 5(2), AutoReplica first writes back "dirty" datasets from $repNode.SSD$ to $prmyNode.HDD$, and keeps them in $repNode.SSD$ with "nondirty" flags. It then broadcasts this "unavailable" information to notify those nodes having replicas of this failure $prmyNode$ (called "associated nodes") to write back "dirty" datasets from their own SSD to HDD. These replicated datasets with "nondirty" flags are still kept in the $repNode.SSD$(s). AutoReplica further finds and replaces the SSD on $prmyNode$. After that, it continues to write incoming I/O requests on both $prmyNode.SSD$ and $repNode.SSD$. It also continues to let those "associated nodes" to write new replicas to $prmyNode.SSD$.

C. Replica Node Cache Device Failure

When a replica node detects a cache device (i.e., SSD) failure, it will disconnect from the primary node and reject any future connection attempts from that node with an error response. As shown in Fig. 5(3), AutoReplica then writes back "dirty" dataset from $prmyNode.SSD$ to $prmyNode.HDD$, but still keeps them in $prmyNode.SSD$ with "nondirty" flag. After a new replica node is found by using dynamic evaluation process, AutoReplica continues to write incoming I/O requests on both $prmyNode.SSD$ and new $repNode.SSD$. Notice that policy in Sec. IV-B takes responsibility for recovering this failure device.

D. Communication Failure Between Primary & Replica Node

When the primary node detects a non-recoverable communication failure between the primary and replica hosts, AutoReplica will be unable to continue to write caching. To recover from this failure, as shown in Fig. 5(4), AutoReplica daemon will write back "dirty" data from $prmyNode.SSD$ to $prmyNode.HDD$ to ensure all cached data are updated to the backend HDD. Next, it will start the dynamic evaluation process to find a new replica node to replace the unreachable replica node. It will then continue to use both SSDs to cache I/Os following the "fusion cache" design in migration policy. Finally, it will broadcast to the network to release all its old replicas on the unreachable $repNode.SSD$(s).

V. Parallel Prefetching

Lastly, replicates can also be used to enable parallel prefetching from multiple nodes (similar like parallel stripping in RAID [15]), especially for read operations. An example is shown in Fig. 6, where we split the dataset (with size of C) to perfetch (e.g., a file) into two parts (with sizes of αC and C), and load each part from the primary and replica node. Assume the access speed of $prmyNode.SSD$ is λ_1 (GB/Sec) and the access speed of $repNode.SSD$ (including the network delay) is λ_2 (GB/Sec). Since the main target for parallel prefetching is to reduce the total I/O time, i.e., makespan of

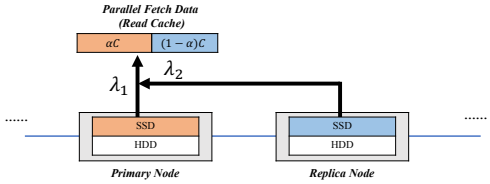

Fig. 6: Example of Parallel-fetch-enabled Read Cache.

each I/O request, we convert our problem into the following optimization framework:

Minimize:

$$max(\frac{\alpha C}{\lambda_1}, \frac{(1-\alpha)C}{\lambda_2}) \tag{1}$$

Subject to:

$$\alpha \in [0, 1] \tag{2}$$

$$\lambda_1 \geq \lambda_2 > 0 \tag{3}$$

$$\frac{C}{\lambda_1} \geq max(\frac{\alpha C}{\lambda_1}, \frac{(1-\alpha)C}{\lambda_2}) \tag{4}$$

Eq. 1 is the objective function which minimizes the overall makespan of an I/O request. The makespan is determined by the maximum value of the I/O operating time of each side (i.e., $prmyNode.SSD$ and $repNode.SSD$). Eq. 2 ensures that the branching ratio of two streams should be meaningful. Eq. 3 reflects that the local I/O speed (i.e., from $prmyNode$) is usually greater than remote (i.e., $repNode$) I/O speed including network delay. Notice that this constraint can be relaxed as "$\lambda_1 > 0$ and $\lambda_2 > 0$", if the remote I/O speed is higher (which is true in some rare cases), but the optimization framework remains the same. Eq. 4 further ensures that the parallel perfecting operation should only be triggered when it *can* help to reduce the I/O makespan.

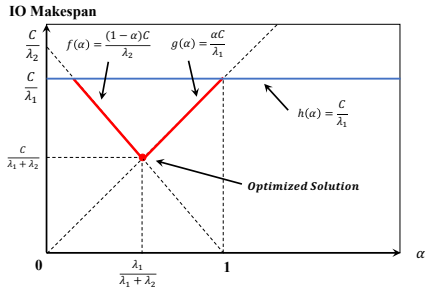

Fig. 7: Finding the optimized solution of prefetching stream division.

We then plot these functions and constraints into Fig. 7, where the red line is the objective function curve, and the blue line is the constraint of Eq. 4. We can see that there exists a minimum point at the cross point of $f(\alpha) = \frac{(1-\alpha)C}{\lambda_2}$ and $g(\alpha) = \frac{\alpha C}{\lambda_1}$. In order to calculate this sweet spot, we let:

$$\frac{\alpha C}{\lambda_1} = \frac{(1-\alpha)C}{\lambda_2} \tag{5}$$

Then, we can get the minimum makespan ($\frac{\alpha C}{\lambda_1+\lambda_2}$) when:

$$\alpha = \frac{\lambda_1}{\lambda_1 + \lambda_2} \tag{6}$$

Based on this result, we develop the parallel prefetching policy. Alg. 8 first describes the main procedure of parallel fetching daemon, which triggers the parallel perfecting by periodically checking whether the access speed of $repNode.SSD$ (including the network delay) is close enough to the access speed of local $prmyNode.SSD$ (by comparing their difference with a preset threshold ε), and the current utilization ratio of throughput of the $repNode.SSD$ is less than a threshold Thr. Notice that the time window T_W does not necessarily to be same as the sliding window previously mentioned in Alg. 3. The parallel prefetching will be approved if all these conditions are satisfied, and the parallel fetching policy then calculates and assigns the branching ratio of dataset to be loaded from each node. By using the same technique, AutoReplica supports parallel prefetching from more-than-one replica nodes feature. This idea can also be extended to support parallel write I/O operations, which need some additional sync and lock schemes to ensure data consistencies.

Main Procedure of Parallel Fetching Daemon

Note: (1) $currUtil_{IOPS}$: current throughput utilization rate of a node.
(2) ϵ: preset threshold to compare the difference between local and remote access speed.
(3) Thr_{IOPS}: preset threshold of upper bound of throughput utilization rate of a node to be perfected from.

Procedure *parallelFetchDaemon*()
1 $parallelReadFlag = False$
2 **while** *True* **do**
if ($currTime \bmod T_W == 0$) **and** ($\lambda(prmyNode.SSD) - \lambda(repNode.SSD) \leq \epsilon$) **and**
3 ($repNode.currUtil_{IOPS} \leq Thr_{IOPS}$) **then**
4 $parallelReadFlag = True$
5 **else**
6 $parallelReadFlag = False$
7 **return**

Parallel Fetching Policy

Note: (1) $\lambda(prmyNode.SSD)$: the IO speed of $prmyNode.SSD$.
(2) $\lambda(repNode.SSD)$: the end-to-end IO speed of $repNode.SSD$ (including network delay).

Procedure *parallelFetch*($data$)
1 **if** $parallelReadFlag == True$ **then**
2 $\alpha = \frac{\lambda(prmyNode.SSD)}{\lambda(prmyNode.SSD)+\lambda(repNode.SSD)}$
3 fetch size of $\alpha \cdot |data|$ part of data from $prmyNode.SSD$
fetch size of $(1-\alpha) \cdot |data|$ part of data from $repNode.SSD$
4 **return**

Fig. 8: Parallel Prefetching Procedure.

VI. Work In-Progress

We are currently building the AutoReplica in a multi-nodes cluster consists of NVMe SSDs and RAID mode HDDs to improve the performance. Hosts are connected by high speed fiber cables. In terms of software implementation, AutoReplica works on the VMWare's ESXi in the "user mode". With our customizations, AutoReplica is very close to a "pseudo-kernel mode" application. AutoReplica consists of 4 components: a vSphere web client plug-in, a CIM provider, multiple I/O filter library instances and a daemon process on each VM. Our plans of validation and evaluations are:

- Different replacement algorithms will be tested under AutoReplica. Evaluation metrics will be I/O hit ratio and average I/O latency.
- We plan to test the performance difference of using SSD as a read-only, a write-only and a read-write cache.
- Both homogeneous and heterogeneous VMs are planned to be installed onto AutoReplica to measure the performance interference among different combinations.
- We will test the recovery correctness and efficiency of different failure scenarios that are manually triggered and following well-tuned temporal interval distributions.
- We will measure the benefit and overhead of multiple replicas under different SLA configurations. We will focus on the consistency and atomicity of I/O operations.
- We plan to measure the I/O speed improvement brought by parallel prefetching, with the consideration of the corre-

sponding overhead, such as I/O path and network bandwidth.
- We plan to work on AutoReplica's compatibility with other hypervisors such as KVM/Xen and Virtual Box.

VII. Related Work

Replications are widely used in the big data and cloud computing era. Facebook's proprietary HDFS implementation [4] constrains the placement of replicas to smaller groups in order to protect against concurrent failures. MongoDB [3] is a NoSQL database system that uses replicas to protect data. Its recovery scheme is based on the election among live nodes. Copyset [16] is a general-purpose replication technique that reduces the frequency of data loss. [17] designed a novel distributed layered cache system built on the top of the Hadoop Distributed File System. Studies [7], [14], [18]–[21] investigated SSD and NVMe storage-related resource management problems, in order to reduce the total cost of ownership and increase the Flash device utilization to improve the overall I/O performance. Based on the frequency of data operation, [22] is proposed to solve the problem of uneven distribution of data in auto-sharding. [23] developed an automated method for identifying and repairing logical data discrepancies between database replicas in a database cluster. [24] proposed a replication strategy based on the access pattern of tile in order to optimize load balancing for large-scale user access in cloud-based WebGISs. Triple-H [25] is a hybrid design to minimize the I/O bottlenecks in HDFS and ensure efficient utilization of heterogeneous storage devices on HPC clusters.

VIII. Conclusion

We proposed a complete data replica manager solution called "AutoReplica', working in distributed caching and data processing systems using SSD-HDD tier storages. AutoReplica balances the trade-off between the performance and fault tolerance by storing caches in replica nodes' SSDs. It has three approaches to build the replica cluster in order to support multiple SLAs, based on an abstract "distance matrix" which considers preset priorities, workload temperature, network delay, storage access latency, and etc. AutoReplica can automatically balance loads among nodes, and can conduct seamlessly online migration operation (i.e., migrate-on-write scheme), instead of pausing the subsystem and copying the entire dataset from one node to the other. AutoReplica further supports parallel prefetching from both primary node and replica node(s) with a new dynamic optimizing streaming technique to improve I/O performance.

References

[1] J. Lee, B. Bagheri, and H.-A. Kao, "A cyber-physical systems architecture for industry 4.0-based manufacturing systems," *Manufacturing Letters*, vol. 3, pp. 18–23, 2015.

[2] L. Parolini, B. Sinopoli, B. H. Krogh, and Z. Wang, "A cyber–physical systems approach to data center modeling and control for energy efficiency," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 254–268, 2012.

[3] K. Chodorow, *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[4] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS under HBase: A Facebook messages case study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 199–212.

[5] T. Hatanaka, R. Yajima, T. Horiuchi, S. Wang, X. Zhang, M. Takahashi, S. Sakai, and K. Takeuchi, "Ferroelectric (fe)-nand flash memory with non-volatile page buffer for data center application enterprise solid-state drives (ssd)," in *2009 Symposium on VLSI Circuits*. IEEE, 2009, pp. 78–79.

[6] "vSphere Hypervisor," www.vmware.com/products/vsphere-hypervisor.html.

[7] J. Tai, D. Liu, Z. Yang, X. Zhu, J. Lo, and N. Mi, "Improving flash resource utilization at minimal management cost in virtualized flash-based storage systems," *IEEE Transactions on Cloud Computing*, vol. 7, no. 99, p. 1, 2015.

[8] J. Tai, B. Sheng, Y. Yao, and N. Mi, "SLA-aware data migration in a shared hybrid storage cluster," *Cluster Computing*, vol. 18, no. 4, pp. 1581–1593, 2015.

[9] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.

[10] F. J. Corbato, "A paging experiment with the multics system," DTIC Document, Tech. Rep., 1968.

[11] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, 2003, pp. 115–130.

[12] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement." in *FAST*, vol. 4, 2004, pp. 187–200.

[13] D. Liu, N. Mi, J. Tai, X. Zhu, and J. Lo, "VFRM: Flash resource manager in vmware esx server," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–7.

[14] Z. Yang, J. Tai, J. Bhimani, and N. Mi, "GREM: Dynamic SSD Resource Allocation In Virtualized Storage Systems With Heterogeneous VMs," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[15] "Understanding Penalty of Utilizing RAID," theithollow.com/2012/03/21/understanding-raid-penalty/.

[16] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 37–48.

[17] J. Zhang, G. Wu, X. Hu, and X. Wu, "A distributed cache for hadoop distributed file system in real-time cloud services," in *2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE, 2012, pp. 12–21.

[18] Z. Yang, M. Awasthi, M. Ghosh, and N. Mi, "A fresh perspective on total cost of ownership models for flash storage," in *8th IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2016.

[19] Z. Yang and M. Awasthi, "Online flash resource migration, allocation, retire and replacement manager based on multiple workloads waf tco model," 2015, uS Patent, US15/094971.

[20] J. Roemer, M. Groman, Z. Yang, Y. Wang, C. C. Tan, and N. Mi, "Improving virtual machine migration via deduplication," in *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems*. IEEE, 2014, pp. 702–707.

[21] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[22] Y. Liu, Y. Wang, and Y. Jin, "Research on the improvement of MongoDB Auto-Sharding in cloud environment," in *Computer Science & Education (ICCSE), 2012 7th International Conference on*. IEEE, 2012, pp. 851–854.

[23] R. E. Wagner, "Automated method for identifying and repairing logical data discrepancies between database replicas in a database cluster," Feb. 28 2012, uS Patent 8,126,848.

[24] R. Li, W. Feng, H. Wu, and Q. Huang, "A replication strategy for a distributed high-speed caching system based on spatiotemporal access patterns of geospatial data," *Computers, Environment and Urban Systems*, 2014.

[25] N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar, and D. K. Panda, "Triple-H: A hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 101–110.