12MAP: Cloud Disaster Recovery Based on Image-Instance Mapping

Shripad Nadgowda, Praveen Jayachandran, Akshat Verma

I2Map: Cloud Disaster Recovery based on Image-Instance Mapping

Shripad Nadgowda, Praveen Jayachandran, and Akshat Verma

IBM Research, India
{shnadgow,praveen.j,akshatverma}@in.ibm.com

Abstract. Virtual machines (VMs) in a cloud use standardized 'golden master' images, standard software catalog and management tools. This facilitates quick provisioning of VMs and helps reduce the cost of managing the cloud by reducing the need for specialized software skills. However, knowledge of this similarity is lost post-provisioning, as VMs could experience different changes and may drift away from one another. In this work, we propose the I2Map system, which maintains a mapping between each instance and the golden master image from which it was created, consisting of a record of all changes to the instance since provisioning. We motivate that this mapping can aid several cloud management activities such as disaster recovery, system administration, and troubleshooting. We build a host-based disaster recovery solution based on *I2Map*, which is ideally suited for low cost cloud VMs that do not have access to dedicated block-based storage recovery solutions. Our solution deduplicates changes across VMs and needs to replicate only the unique changes, significantly reducing replication traffic on end hosts. We demonstrate that *I2Map* is able to deliver on tight recovery time and recovery point objectives of the order of minutes with low overhead. Compared to state-of-the-art host-based recovery solutions, *I2Map* is able to save 50-87% network bandwidth on the primary data center.

1 Introduction

Enterprises are moving their IT infrastructure to cloud in order to gain greater flexibility in acquiring and relinquishing resources on demand, focus on core capabilities, reduce costs and avoid capital lock-in. Despite the obvious advantages of the cloud delivery model, CIOs remain skeptical about its potential with concerns primarily regarding availability. A survey [7] attributes increasing customer reluctance to move to the cloud to the problem of poor performance. For instance, outages in Amazon's EC2 and AWS [2] have costed companies millions of dollars. While most cloud providers offer high availability services [3, 11, 22, 8], these come at a premium that is unaffordable for many enterprises. Block-based storage replication solutions such as [15, 13] require expensive specialized storage controllers, storage area networks, or other hardware. Network-based replication is performed by a separate component from SAN/NAS or the hosts and can work between multi-vendor products. However, these solutions require intelligent switches which are expensive. Highly available cloud services at an affordable cost remains a distant dream. There is a need for solutions that can work with commodity hardware, is cheap, and yet provides good recovery performance.

Host-based solutions [4, 16] are cheaper and less complex than storage-based or network-based solutions as they can be implemented completely in software. They do not require any specialized hardware. They are usually file-based and asynchronous,

and work by trapping and forwarding write changes to the replication target. Their overheads and performance are also typically worse than the other two approaches.

The core idea of this work is to leverage the similarity of virtual machines (VMs) in a data center to provide a low-cost host-based disaster recovery solution. A few standardized 'golden master' images are used to provision VMs in a cloud, to ensure quick provisioning and to reduce management costs. Hence, VMs which are provisioned from the same golden master tend to be similar to one another. However, knowledge of this similarity is lost post-provisioning as instances could be used for different purposes and may drift away from one another. We build *I2Map*, which maintains a record of all changes to an instance, as a mapping between the instance and the golden master image from which it was provisioned. A light-weight agent running on each VM records all changes and transmits them to a set of aggregators. The aggregators deduplicate these changes across VMs, store only the unique changes, and maintain the mapping for each VM. A snapshot-mirroring technique can then be applied to backup the aggregators on to a remote site. This recovery process allows us to trade-off recovery performance for cost. We evaluate *I2Map* on representative activities such as installing new software, patching the operating system, and running hadoop-based applications. We demonstrate that individual VMs can receive good recovery performance of a few minutes without having to invest in dedicated and specialized hardware. We conduct a 24-hour high-load case study experiment where we recover a failed VM within a recovery time of 20 minutes and having a recovery point of less than 4 minutes. We show that *I2Map* uses 50-87% lesser network bandwidth on the primary data center compared to the state-of-the-art host-based recovery solutions.

The image-instance mapping can potentially be used for other applications such as system administration or troubleshooting failures. We discuss these as part of future work in Section 7. For the rest of this paper, we focus on the disaster recovery solution.

The rest of this paper is organized as follows. We provide some background and motivate our problem and solution in Section 2. We present the design of *I2Map* in Section 3. Section 4 describes our implementation of *I2Map* and certain optimizations we performed. We evaluate *I2Map* and report the results in Section 5. Section 6 discusses related work, and Section 7 highlights the limitations and other potential applications of *I2Map*. We finally conclude this paper in Section 8.

2 Background and Motivation

The motivation for our work stems from two important trends in cloud computing.

The first trend is increased standardization and automation of IT services delivery in clouds. Virtual servers are created automatically from virtual image templates and managed via standard tools and processes [21]. Applications are deployed from standard software catalog, which contain standardized version of popular middleware and application software. Cloud computing providers use standardization to drive automation of IT delivery as well as to keep the costs down. It has been widely reported that standardization allows system management costs to be significantly reduced [1]. Standardization of virtual image templates, software catalog and management tools lead to high similarity in cloud managed servers. Servers in a data center are already known to exhibit high similarity [5, 14] and standardization increases content similarity in virtual machines even further.

The second trend that drives our work is the increasing use of commodity servers and storage in infrastructure clouds. This helps cloud present a low-cost IT model by achieving significant cost-savings. While there are several block-based disaster recovery solutions [3, 13, 11, 8, 15] that provide replication across different availability zones, they all require high-end servers and storage technologies like SAN. By contrast, clouds often use commodity servers with attached storage. Commodity hardware does not contain enterprise features like high-availability, block-level replication, fault tolerance and these features need to be implemented at cluster management layer. Disaster recovery is a popular system management functionality, which is impacted by use of commodity hardware. Disaster recovery is often characterized by the Recovery Time Objective (RTO) or the time taken to recover a protected server, the Recovery Point Objective (RPO) or the maximum period of data loss, and the impact of replication on application performance. In a low-cost cloud, disaster recovery depends on host-based replication, which leads to high network traffic and impacts application performance.

In this work, we conjecture that standardization-induced similarity in cloud managed servers can significantly improve system management. Virtual servers instantiated from common image templates and with software deployed from a common catalog, share common content. However, current system management technologies work within virtual server boundaries and are unable to leverage this similarity. If we can create a succinct representation of instances as they evolve from a common image template and software catalog, this representation captures the similarity between instances in an instantly usable format. We call this representation an image-instance mapping called *I2Map* and capture it as a tree, where the master image is the root and each leaf node is a virtual server instance.

We use the *I2Map* tree to implement improved disaster recovery in low cost clouds. Replicating the *I2Map* tree to a secondary site is enough to recreate all the virtual servers on the secondary site. Hence, disaster recovery is trivially supported using the *I2Map* tree. The *I2Map* tree keeps only one copy of an update even if the update is made across a large number of cloud instances. Hence, the tree automatically eliminates redundancy and reduces network traffic, while replicating the tree on a secondary site. We also design techniques to ensure that the tree can be created without propagating updates from end hosts, leading to low traffic overhead even within the primary site.

3 Design

In this work, we design and build *I2Map*, a host-based disaster recovery solution that leverages redundancy in operations across VMs in a data center. We first outline the challenges we faced, describe the *I2Map* architecture, and then highlight the key design ideas that helped overcome the challenges.

3.1 Design Challenges

Identify duplicates across VMs without transferring data: Every write operation incurs an overhead with storage-based deduplication. Network-based techniques deduplicate only across data centers after bytes have been transmitted over the internal network. Host-based replication techniques deduplicate changes only on the backup server. Can we exclude duplicates across VMs without transferring the actual data?

Control overhead: Although, host-based techniques have a low cost and complexity of implementation, they have the disadvantage of having an agent running on each of the VMs. The agent uses network, CPU and memory resources leading to application impact. It is critical to control the overhead incurred by this agent and ensure that running applications are not affected.
Handle large files with small changes: Large files (e.g., log files) could undergo few minor changes or additions. How do we avoid transmitting the entire file each time they are modified?
Handle high load: When the load is high, the agent should not consume more resources. The overhead needs to be bounded.
Handle rapid updates to the same file: Rapid updates are typically correlated with high load. How can we handle rapid updates within the bounded resources?
Scaling deduplication: Deduplication often requires centralization. How do we scale deduplication in a cloud with thousands of servers?

3.2 *I2Map* Architecture

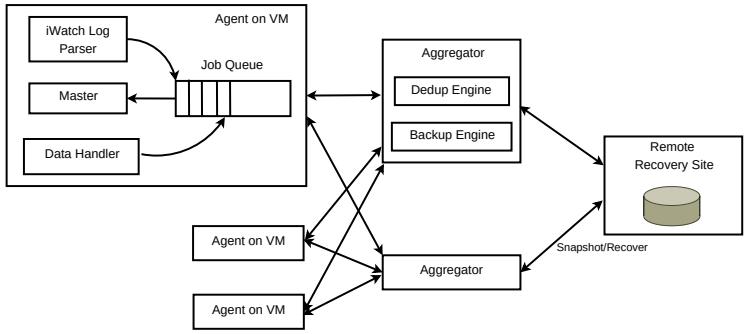


Fig. 1. Architecture of *I2Map*.

In this section, we present the architecture and main components of our *I2Map* disaster recovery solution. A light-weight agent runs on each VM that continuously monitors and reports meta-information regarding any changes on the VM. Dedicated aggregator machines collect reports from all the VMs, identify duplicates, request for and retrieve unique data from the agents running on the VMs. The aggregators also maintain information regarding which files are contained in each VM. Snapshots of the aggregators are backed up on to a remote site periodically. As copies of the aggregators are available on the remote site, any VM can be easily retrieved using its golden master image and the catalog of changes to the file system for the VM.

The *I2Map* agent comprises of three components. A file system monitor such as iWatch in Linux, monitors all changes to the file system. A `Parser` identifies files with changes to their meta-information (such as permissions and ownership) or to their content. A report of these changes (without file content) is sent to the aggregators. Files with changes in content are added to a job queue. The `Master` module picks up files from the job queue and computes Rabin fingerprints [18]. Rabin fingerprinting creates cryptographic hashes for variable size shift-resistant blocks. The hashes for each block are then transmitted to the aggregators. The `Dedup Engine` on the aggregator identifies blocks that are unique and requests one of the VMs holding each block to transmit

the contents. The `Data Handler` on the agent receives these requests and transmits all blocks requested from it. The `Backup engine` on the aggregator maintains the *I2Map* tree and records the changes to the file system for each VM. Periodically, snapshots of the aggregator are transmitted to a remote recovery site. Recovery for any VM can then be performed on-demand on the remote site. We describe the functioning of each of these modules in detail in Section 4.

3.3 Key Design Ideas

We next highlight and discuss certain key design ideas that enabled us overcome the challenges identified in Section 3.1.

Two-round data transfer: In order to ensure that duplicate data is not transferred from a protected host, we transfer data from agent to aggregator in a two round scheme. In the first round, fingerprints of file segments are sent to the aggregator. The aggregator maintains a hash index of fingerprints for all data that it has aggregated and requests data only for segments that are not already present in its hash index.

Separating deduplication and replication: We separate duplicate identification from data replication. Duplicate identification and aggregation of unique data is handled using a two round protocol between agent and aggregator. Data replication to secondary site is performed independently by the aggregators. This allows duplicate elimination to work at LAN speed, while replication can be performed at WAN speed.

Variable size duplicate identification: For each file that is written, the agent computes a Rabin fingerprint. Rabin fingerprints are shift-resistant, as the division into variable-size blocks and hash computation is based on the content of each block rather than any fixed offset. The hashes for the various blocks are sent to the aggregator. Therefore, even if a small change is made to a large file, only a small block of data around the change will need to be transmitted to the aggregator. The remaining blocks will be identified by the aggregator as duplicates of blocks already present and will not have to be transmitted. This helps us to reduce the overhead of file-based replication significantly.

Pipelining fingerprint computation with data transfer: A two-round protocol can lead to high Recovery Point Objective (RPO), if the rounds were serialized. Further, both rounds may need to perform disk I/O for a file segment. In order to speed up the process of identifying duplicates and aggregating unique data, we pipeline the different operations performed on the agent. The `Parser`, `Master`, and `Data Handler` are implemented as separate threads in order to allow them to progress parallelly acting on different data elements. Further, Master and Data Handler use a producer-consumer cache to minimize disk I/O. Master populates the cache with file segments, when it computes the fingerprints in the first round. In the second round, when `Data Handler` needs to send data, it reads the file segment from the cache, instead of reading it again from the disk.

Change coalescing: The agent's parser module looks ahead and identifies multiple successive writes to a file within a short duration and transmits information only once to the aggregator. This change coalescing helps *I2Map* deal with rapid updates to a file even during periods of high load.

Agent throttling: We allow the agent to be configured with a throttling parameter, in order to ensure that it does not consume too much of the CPU and memory resources of

the VM. This further reduces the overhead and ensures that applications that generate high I/O load operate smoothly.

Stable aggregator mapping: In order to scale to large data centers, we allow multiple aggregators to be created in *I2Map*. If multiple agents find common content, duplicate elimination requires all (or at least most) agents to send this content to the same aggregator. Clearly, this requires aggregator mapping to be defined based on content. Further, if a part of the file changes, we would like to send only the changed content to the aggregator. This places a restriction that the aggregator mapping should not change due to a small change in content. In order to deal with these conflicting requirements, we use hash of the first $4KB$ of a file to map its aggregator. Changes in other parts of the file do not lead to change in aggregator. Also, files across VMs with the same content, map to the same aggregator most of the time, meeting both our requirements.

4 Implementation

In this section, we describe the implementation of *I2Map* and highlight important optimizations that helped us keep overhead in check.

4.1 *I2Map* Agent

The agent comprises of three modules as depicted in Figure 1, implemented in Python.

iWatch and Parser The agent, at its core uses iWatch, a real-time file monitoring utility written in perl, based on iNotify, a file change notification system in the linux kernel. We monitor the entire file system excluding device files and temporary files in folders such as /dev. Whenever a file's contents or metadata (including permissions and ownership) is modified, iWatch generates a log. The `Parser` module thread checkpoints the log, processes all entries up to the checkpoint, and then zeroes all lines up to the checkpoint. This ensures that the iWatch log file is never too large, and *I2Map* does not incur the overhead of opening a large file to read.

The `Parser` computes a hash of the first $4KB$ of each file that is written. This hash is used to decide which aggregator is in charge of holding the file (we choose the first $4KB$ only, to ensure content-based stable mapping). The vector space of all the hash values is evenly split among all the aggregators and each aggregator is responsible for managing files with hash values in its vector space. The `Parser` creates reports, one for each aggregator, with meta-information regarding all changes to files managed by that aggregator. Information regarding files that are deleted are sent to all aggregators, and the aggregator to which the file is relevant can then delete the file. Files that are modified are treated as a delete followed by a create.

The `Parser` also adds any files that are newly created or have changed in content on to a job queue. The job queue contains the ID of the aggregator responsible for the file along with certain file meta-data. However, before adding the file entry, the `Parser` takes a peek at the job queue. If an entry already exists for the file, then the `Parser` skips entering the file again into the job queue. This allows the agent to optimize during periods of rapid writes, when a file is written multiple times in quick succession.

Master The `Parser` and the `Master` share a producer-consumer relationship with respect to the elements in the job queue (Figure 2). The job queue is controlled by a lock and both the `Parser` and the `Master` need to acquire the lock before writing to

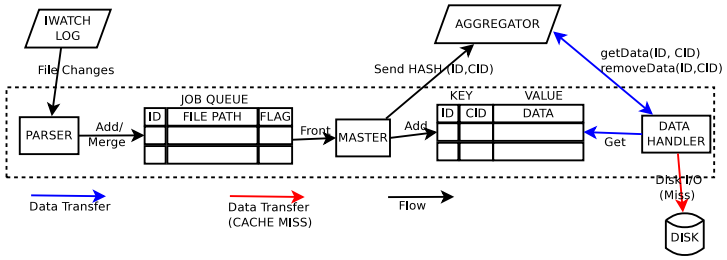


Fig. 2. Detailed Design of *I2Map* Agent

it. The `Master` picks up each file entry written by the `Parser`, and computes a Rabin fingerprint for the file, implemented in C. The Rabin fingerprint divides each file into variable-sized blocks based on the content rather than any fixed offset. This makes these blocks shift-resistant, that is, for example changes to the start of a file will not affect all the blocks. The fingerprint consists of cryptographic hash values for each of the blocks. The `Master` sends the hash values for blocks of each file to the corresponding aggregator, as identified in the job queue. In our implementation, we did not limit the size of the job queue, as we observed that the size never grew beyond 700 entries, with each entry being less than 30 bytes.

In order to ensure that even under high I/O load the agent does not consume too much of the VM's resources, *I2Map* can be configured with a throttling parameter. For instance, a throttling parameter of 67% would run the agent for 5*s* and then sleep for the next 10*s*. In our implementation, we kept the awake time to be constant at 5*s*, and alter the sleep time based on the throttling parameter. We observed that the Rabin fingerprinting and the actual data transfer were the costliest operations, while the parser was extremely light-weight. A crucial design choice was to selectively throttle the `Master` and the `Data Handler`, but not the `Parser`. Apart from throttling the most time-consuming tasks of the agent, this had the added advantage that any duplicate file writes within the VM would be safely omitted by the parser, as the file has not yet been read by the `Master`. Separating these modules into parallel threads and having the job queue as a shared resource between them was important to achieve this.

Data Handler The `Data Handler` responds to requests from each aggregator, with the content of the specific blocks of files requested by that aggregator. We noticed that we were reading the file twice, once for computing the Rabin fingerprint and a second time for the block transfer. This was adversely affecting performance. To alleviate this problem, we introduced a key-value store cache (Figure 2). When computing the Rabin fingerprint, the `Master` would write the blocks onto this cache. The `Data Handler` will look into the cache first for each block, and will read the file system only if it is unable to find the block in the cache. If found, the `Data Handler` would remove the entry from cache, after use. The `Data Handler` also removes those entries from the cache, which the aggregator already has and does not require. This ensures that only file segments not yet processed by the `Data Handler` stay in cache.

This considerably helped improve performance. From our experiments, we observed that typically the cache had about 400-500 entries (even for data-intensive hadoop experiments), with each entry holding a Rabin fingerprint for a block. The lag between computing the Rabin fingerprint and the data transfer was always small enough to keep the cache small. We limited the cache to 1000 entries and Rabin fingerprint blocks were restricted to a maximum size of 64*KB*, ensuring that the cache had a maximum size of

64*MB*. Since the workload is scan-based, we never replace unprocessed entries from the cache. Instead, the `Master` waits till a cache block is made available by the `Data Handler`. We also conducted a few overload experiments where the cache was fully utilized and tested a few replacement algorithms. We observed that not replacing entries in the cache, if the cache was full performed the best.

When transmitting blocks of requested files to an aggregator, the `Data Handler` sends at most 500 blocks at a time, to ensure that packet sizes aren't too big. We used a base64 encoding for file transfer as some files, especially those written by hadoop, had certain special characters.

4.2 Aggregator

The aggregator consists of two main modules. The `Dedup Engine` communicates with the agents and identifies blocks that are unique. For each unique block, it requests the contents of the block from one of the VMs holding it. The `Backup Engine` maintains a record of all the files and blocks (among those managed by this aggregator) contained in each VM. An *I2Map* tree is constructed for each golden-master image, where the master image is the root, and each leaf node is a virtual machine instance. Edges in the tree represent changes to files. If a set of VMs experience the same changes to files (e.g., a patch is applied), the `Dedup Engine` would ensure that only one copy of the change is stored. Replicating this tree on a remote site is sufficient for disaster recovery, as any VM can now be recreated by starting with its golden master image.

Interestingly, for the disaster recovery use case, *I2Map* does not even need to create the entire tree. Instead, what we maintain is a list of instances that are relevant for each update to the tree. For example, if a file got overwritten in 5 instances, we store the change along with the 5 instances, whose I2Map tree contain the change. The I2Map tree is thus stored as a set of nodes (one node for the golden master and one node for each instance). The intermediate nodes, which capture the transition from a golden master to an instance are not stored. Instead, a list of all changes are stored along with the impacted instances. Multiple updates to the same file segment are merged leading to a compact I2Map tree, whose size is proportional to the minimum set of changes needed to convert a golden master to any required instance.

4.3 Remote Recovery

The remote recovery site maintains periodic incremental snapshots of all aggregators. Snapshots are taken at 5 minute intervals (a configurable parameter) for each aggregator. We use Linux rsync [20] to transmit the snapshots to the recovery site. Prior to taking a snapshot, the aggregator waits for any packets sent on the wire, freezes all operations for an instant, and takes a filesystem dump of the database and tree. This operation, including taking the snapshot, takes less than a second. In the event of a site failure, the aggregators are recreated and the *I2Map* tree with record of changes to files is used to recreate VMs. We perform incremental recovery by merging updates for each VM periodically (default is 24 hours). This does not require a live VM at the recovery site as only the updates are gathered and stored in an offline VM image. When recovery is triggered, the latest updates are merged and an instance is provisioned from this image.

The Recovery Point Objective (RPO), or the worst-case duration for which recovery cannot be guaranteed, that *I2Map* can support depends on two factors - the maximum

time lag of all agents to send data to the aggregators, and the snapshot interval. The RPO for host failure is the lag between agents and aggregator, whereas the RPO for site failure is the sum of the two lags. We show in our evaluation that *I2Map* can guarantee an RPO of less than 4 minutes for host failures and an RPO of less than 9 minutes for site failure, sufficient for most non-critical applications (assuming 70 MBps within the primary site, 700-3300 KBps over WAN, and 1 GB data generated per minute).

Aggregators maintain a heartbeat among one another. In the event of an aggregator failure, one of the live aggregators (e.g., a chosen leader) takes up the responsibility of storing content on behalf of the failed aggregator. Agents are intimated accordingly. Recovery is initiated for the aggregator using snapshots on the remote site. Until recovery for the aggregator is complete, the acting aggregator may not be able to perform efficient deduplication. This, however, does not compromise safety of the system.

5 Evaluation

We evaluate *I2Map* on a heterogeneous set of 6 VMs running Ubuntu 10.04.2 64-bit. The VMs were hosted on 2 IBM BladeCenter servers, one with 4 cores 2.33GHz and 8GB memory, and the other with 8 cores 3GHz and 16GB memory. The memory and CPU specifications of the VMs are shown in Table 5. We did not set an upper limit to the CPU available for a VM, and it was bounded only by the availability of resources on the server hosting it. The aggregator was run on a physical server with an 8-core 2.27GHz Xeon processor and 16GB memory. Recovery is performed on a server with 24-core 3.07GHz Xeon(R) processor and 64GB memory. The primary site was located in New Delhi and the remote recovery site was located in Bangalore, over 2000kms away. We observed speeds of about 70$MBps$ within the primary site (between the agents and the aggregators). The WAN bandwidth between New Delhi and Bangalore varied with time of the day and was between 700$KBps$ to 3.3$MBps$. With better network speeds, our recovery performance will only improve.

VM-ID	Memory (MB)	vCPUs	CPU Reservation
vm-1	1024	4	0
vm-2	2048	2	684
vm-3	2048	2	684
vm-4	2048	2	1500
vm-5	3072	4	2300
vm-6	3072	2	0

Table 1. Virtual Machine Specifications

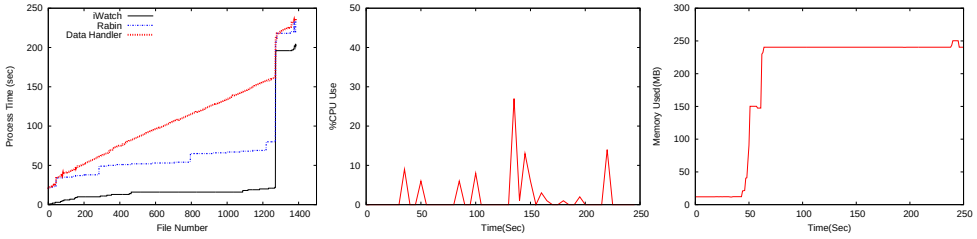
We evaluate *I2Map* based on several metrics. We define *dedup* as the ratio of the total bytes written on a VM to the bytes transferred from the VM to the aggregators. $1 - 1/dedup$ captures the reduction in network traffic for each VM by *I2Map* over state-of-the-art host-based replication solutions. We also define *dedup_aggr*, which is the aggregated measure of the total bytes written across all VMs to the total bytes transferred from all VMs to the aggregator. This is a measure of savings in storage and network transfer within the data center due to *I2Map* over state-of-the-art techniques. We measure the (*time lag*) between when a file is written and when it is transmitted to the aggregator (if requested by the aggregator). This measure captures the RPO for host failure and together with the time taken to transfer from the aggregators to the remote recovery node, represents the RPO for site failure. We also measure the CPU and memory utilization (*CPU_Util* and *Mem_Util*) of our agent running on each VM to quantify

the overhead of our approach. Finally, in the event of an actual failure of a VM, we measure the recovery time objective (RTO) achieved by *I2Map*.

5.1 Micro Experiments

In this section, we describe micro experiments that we conducted to analyze the performance of *I2Map*. We chose three common activities in a cloud - namely, software installation from a software catalog, patching VMs in a change window, and running clustered applications, for these experiments.

Software Installation For this experiment we downloaded and installed two software along with all their dependencies, with a 150*s* sleep time between the two. This process was repeated in sequence on 6 VMs. We chose freecad, an open source autocad software, and avgscan, an anti-virus scanning software, for their relatively large size and the number of dependencies with other software and libraries. It is possible that some VMs already had the dependent software and didn't need them to be installed. Freecad had a download size of 60.6*MB* and an install size of 197*MB*. Avgscan had a download size of about 100*MB* and a similar install size. The software installation scenario is one where a large number of files are written within a very short duration of time.



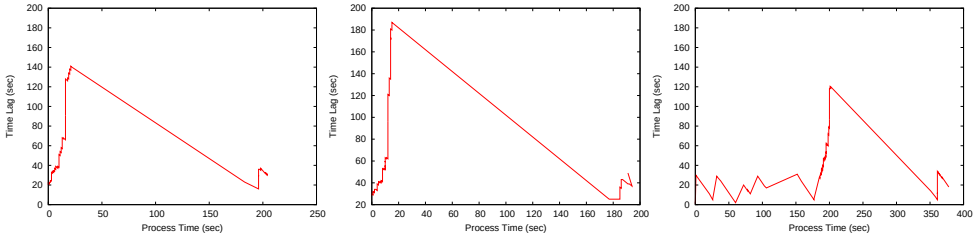
(a) (b) (c)

Fig. 3. (a)Time plot showing the split between iWatch, Rabin, and data transfer on vm-1 (b) CPU Utilization (c) Memory Usage

For each file modified or added on a VM, we monitored the time at which the `Parser` (iWatch), `Master` (Rabin), and `Data Handler` handled the file. We plot this in Figure 3(a) for one of the VMs (other VMs were similar). The difference between the successive operations shows the time lag in executing these steps. Observe that the Freecad installation wrote about 1250 files in about 20*s* (the iWatch curve), and the data handler was able to catch up with this load at around 170*s*. The figure also shows that the Rabin fingerprinting and the data transfer took nearly equal amount of time. Avgscan, on the other hand, writes only about 150 files (has bigger files than Freecad). *I2Map* is able to handle this load better and has a lag of only about 30s, with most of the delay being due to the data transfer. Hence, *I2Map* is able to achieve an RPO for host failure of less than 3 mins.

Figures 3(b) and (c) show the CPU and memory usage of *I2Map* during this experiment for one VM. We notice that CPU utilization is below 10% except for a couple of brief spikes and the memory usage is less than 250MB, which for servers today is less than 10% of total available memory. Note that this experiment was run without throttling and the resource consumption can be made even lower with throttling.

We next take a closer look at the time lag of *I2Map* over the course of the experiment. At any given time instant, the time lag is measured as the amount of time required



(a) vm-1 (b) vm-3 (c) vm-6

Fig. 4. Time lag vs process time plot for three VMs.

by *I2Map* to process and transmit files written up to that time instant, and plotted in Figure 4 for three VMs. The lag increases in spurts when files are written, but then tapers down as *I2Map* catches up. Unlike vm-1 and vm-3, vm-6 does not have a large spike at the start of the experiment as files were quickly identified as duplicates. However, it has a sharp increase in time lag for the avgscan installation (it was the first to perform it), where it had several files to transmit to the aggregators. These files were identified as duplicates for the other VMs and the experiment concluded earlier for them. Overall, the time lag never increased beyond 160*s*, which is sufficiently low compared to the aggregator snapshot period and the recovery point objective.

VM-ID	Avg Lag (sec)	Avg %CPU	Avg Mem (MB)
vm-1	78	4.97	170
vm-2	52	1.30	33
vm-3	100	2.81	148
vm-4	88	3.80	114
vm-5	82	1.65	67
vm-6	64	2.41	170

Table 2. Average time lag and overhead for each VM.

We summarize the average lag and the average CPU and memory usage for each of the six VMs in Table 5.1. The average lag is less than 100*s*, and the average CPU and memory utilization is less than 5% and 170*MB* respectively. Note that the installation was performed on vm-1 first before the other VMs, and so vm-1 was responsible for transferring most of the data to the aggregators. This was the reason why vm-1 consumed more CPU and memory compared to the other VMs. This is also corroborated in Table 3, which shows for each VM the number of file system change notifications, the number of notifications processed after the `Parser` eliminated intra-VM duplicates, the number of bytes written to disk and the number of bytes transferred to aggregators.

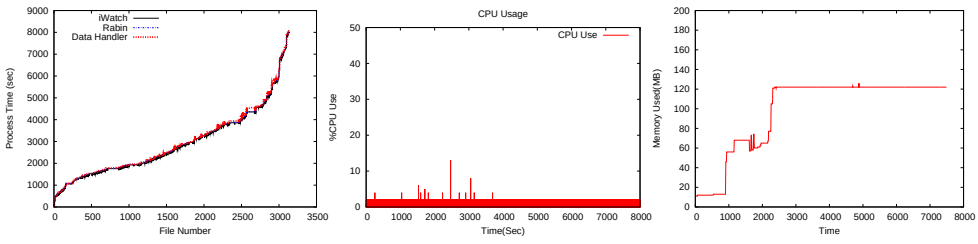
VM-ID	Total FS Notifications	Processed Notifications	Total Data Change(MB)	Transferred Data (MB)	Dedup
vm-1	21893	2357	447	276	1.62
vm-2	14643	847	337	0.11	3063.6
vm-3	30811	2350	450	1.2	375
vm-4	20041	2802	438	0.54	811.1
vm-5	33320	2335	437	0.09	4855.6
vm-6	49668	1526	551	80.32	6.86
Total			2660	358.26	7.42

Table 3. Comparison of dedup values with and without removing duplicate writes for each VM.

We make several interesting observations. First, while around 1400 unique files were written during the experiment (from Figure 3), 20000-40000 file writes were generated on each VM. However, *I2Map* only processed less than 3000 notifications for each VM. This justifies our design choice of *pipelining* and *change coalescing* for multiple changes to the same file. Any storage-based deduplication technique such as [15,

8] incurs an overhead for each of the 20000-40000 file writes. In terms of the number of bytes, observe that vm-1 transmitted only 276*MB*, compared to the 447*MB* written to disk. This is primarily due to the division into blocks performed by Rabin fingerprinting and any blocks that did not change would not be transmitted to the aggregators, justifying the use of *variable size* blocks. The other VMs transfer negligible amounts of data to the aggregator as they were similar to vm-1 and our *two-round data transfer* protocol helped them eliminate transfer of duplicate data. vm-6 was an exception as it was the first to have avgscan installed. It therefore transmitted an additional 80*MB*. If every write were to be captured and replicated, like in other host-based solutions, without the 'deduplication before data transfer' feature of *I2Map*, a total of 2660*MB* of data would need to be transferred. In comparison, *I2Map* transfers only 358.6*MB* of data between the agents and the aggregators, a reduction by a factor of over 7 (*dedup_aggr*).

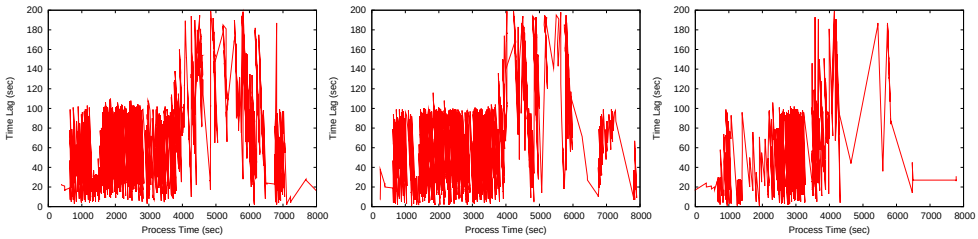
Patching In our next micro experiment, we apply a set of 55 security patches for Ubuntu on 6 VMs. For one of the VMs, vm-6, 19 of these patches were relevant, while for the other VMs, 49-52 patches were relevant (10 patches were not relevant for at least one VM excluding vm-6). The total download size of the 55 patches was about 480*MB*, with 4 patches each about 100*MB*, and 30 patches each less than 500*KB*. We used Tivoli Endpoint Manager [12], an endpoint management tool, to apply the patches in an automated fashion. The experiment took between 130 and 170 minutes to complete on each of the VMs.



(a) (b) (c)

Fig. 5. (a) Time plot showing the split between iWatch, Rabin, and data transfer on vm-2 (b) CPU utilization (c) Memory usage

In Figure 5(a) we present the split between when `Parser` (iWatch), `Master` (Rabin), and `Data Handler` process each file for one sample VM, vm-2. Over 3000 files are modified in about 140 minutes. We observe that the time lag is less than 100*s* most of the time. The CPU utilization is less than 10%, except for a couple of spikes, and the memory usage is less than 120*MB* as shown in Figures 5(b) and (c).



(a) vm-4 (b) vm-5 (c) vm-6

Fig. 6. Time lag vs process time plot for 3 VMs.

Similar to the software install experiment, Figure 6 depicts the time lag of *I2Map* over the course of the experiment. As most patches are small in size, we see many small spikes in the time lag. The larger patches take longer to download and are processed in the second half of the experiment with fewer spikes.

VM-ID	Avg Lag (sec)	Avg %CPU	Avg Mem (MB)
vm-1	58	1.14	106
vm-2	69	1.24	148
vm-3	75	0.98	94
vm-4	65	1.19	143
vm-5	46	1.80	207
vm-6	72	1.05	112

Table 4. Average time lag, and overhead for each VM.

A summary of the average lag and the average CPU and memory usage for each VM is presented in Table 4. The average time lag is less than 80*s* for all the VMs. Unlike the software installation experiment, where the first VM transferred all the data to the aggregators, the patches were applied in different orders on the VMs. Hence, the overhead is more or less uniform across all the VMs.

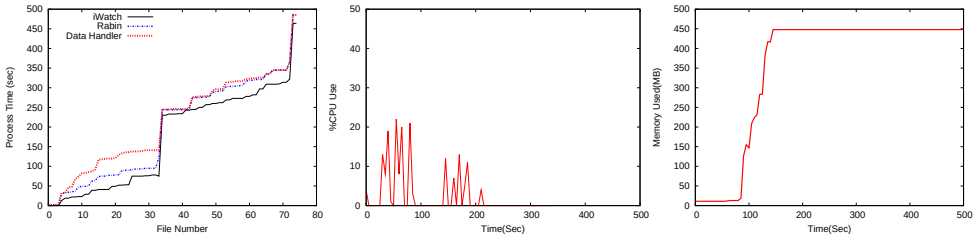
VM-ID	Total FS Notifications	Processed Notifications	Unique Data (MB)	Dedup Data (MB)	Transferred Data (MB)	Dedup
vm-1	63017	16990	10	212	38	5.84
vm-2	32877	10038	10	124	62	2.16
vm-3	62533	16789	13	209	101	2.20
vm-4	63064	16916	10	199	36	5.81
vm-5	55000	15112	10	184	74	2.62
vm-6	58323	3890	106	353	397	1.16
Total			159	1281	708	2.03

Table 5. Comparison of dedup values with and without removing duplicate writes for each VM.

Table 5 summarizes the deduplication information for the patch experiment similar to Table 3. The intra-VM deduplication and change coalescing of *I2Map* reduces the number of writes that need to be processed to about 16000 from about 60000 total writes. This is not as significant a reduction as in the software install case, as the same files are not rewritten multiple times and the time between file writes is longer reducing the amount of intra-VM deduplication possible. The deduplication achieved is mainly due to multiple patches writing the same files. The total data change is split into unique data and dedup data in Table 5. Unique data represents the amount of data that is unique to that VM, and dedup data represents the amount of data that is found in at least one other VM. Observe that a large fraction of the data written on each VM has duplicates. vm-6 is an exception with a larger fraction of unique data. Since, it had only 19 patches relevant, TEM got to apply patches on vm-6 ahead of the other VMs (patches on all 6 VMs were started simultaneously). Hence, a bulk of the data got transferred from vm-6 on to the aggregators, serendipitously achieving load-balancing. The total amount of data transferred to the aggregators was $708MB$, only a half of the $1440MB$ ($1281MB + 159MB$) of total data written across all the VMs ($dedup\_aggr = 2.03$).

Hadoop Sort Our third and final micro experiment is with running the Hadoop Teragen-Terasort application on a cluster of 5 VMs. Terasort is a distributed sort algorithm on $1GB$ data, that is created by Teragen. The sorted data is written separately from the input data. Hadoop uses a distributed file system that is append-only. This creates a challenge for *I2Map* as it creates and appends data on to large files. It waits until a block reaches

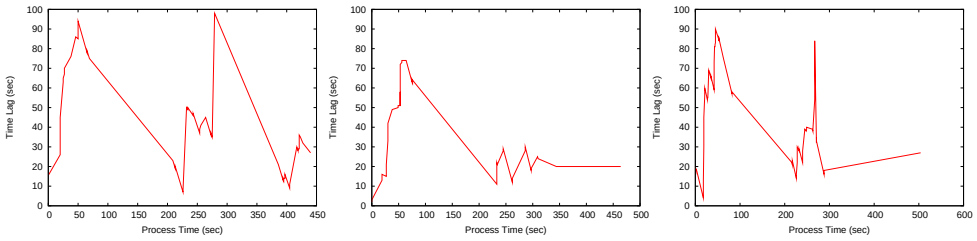
64*MB* and then writes the block to disk. Further, we employ triple replication of data within Hadoop's file system, so the experiment wrote 6*GB* of data in all by the end of the experiment. Identifying and leveraging this replication is critically dependent on the shift-resistant blocks created by the fingerprint. The triple replication also means that a tremendous amount of data is written within a very short amount of time, stress testing both the disk as well as *I2Map*. If ineffective, we may end up transferring a lot of duplicate data. Unless specified otherwise, we use a default value of 67% throttling for this experiment, where the agent is awake for 5*s* and then sleeps for 10*s*.



(a) (b) (c)

Fig. 7. (a) Time plot showing the split between iWatch, Rabin, and data transfer on vm-2 (b) CPU Utilization (c) Memory Utilization

We observe from Figure 7(a) that the time lag never exceeds 100*s*. This is better compared to the install and patch experiments as data writes are more or less uniform and don't happen in a burst. However, CPU and memory usage are higher as observed in Figures 7(b) and (c). This can be attributed to the larger file sizes, the append-only behavior of hadoop, and the triple replication (the total amount of data written during this experiment is 6*GB* in about 450*s*).



(a) Throttle 50% (b) Throttle 67% (c) Throttle 75%

Fig. 8. Time lag vs process time plot for different values of throttling agent process.

Figure 8 shows the time lag as a function of the process time for one VM for different values of the throttling parameter. The lag uniformly increases till about 80*s*. Then, we observe a pause in file writes till about 225*s*, as hadoop gathers data till it can write 64*MB* blocks. This is followed by another set of writes. More strikingly, increasing the throttling does not discernably increase the time lag, suggesting that much of the lag is due to the network transfer between the agent and the aggregators. This is encouraging as a better network would help reduce the overhead of *I2Map*.

Table 6 shows the average lag and overhead for each of the VMs and the aggregator for the above experiment. The average lag is more or less uniform and less than 60*s* for all the VMs. vm-4 handled the maximum data transfer (as we show in Table 8), which explains the higher lag and overhead seen. The aggregator doesn't perform any

VM-ID	Avg Lag (sec)	Avg %CPU	Avg Mem (MB)
vm-1	41	2.46	360
vm-2	33	1.33	257
vm-3	38	2.02	363
vm-4	59	8.91	361
vm-5	50	2.86	257

Avg %CPU	Avg Mem (MB)
36.86	103

Table 6. Overhead for each VM and on aggregator.

file based computations, and only needs to dedup file blocks and receive data from the agents, leaving it with a relatively low memory footprint, but a higher CPU consumption. Even at such a high load, one aggregator can handle 17 agents without throttling and up to 25 agents with throttling, which is an acceptable management overhead (the system can easily scale by adding more aggregators as needed).

	Throttle 50%		**Throttle 75%**	
VM-ID	Avg Lag (sec)	%CPU	Avg Lag (sec)	%CPU
vm-1	35	2.64	43	1.43
vm-2	37	1.91	39	1.80
vm-3	46	5.95	64	7.78
vm-4	51	3.36	62	1.67
vm-5	53	3.30	61	2.03

Table 7. Average time lag for different values of throttling agent process.

The above experiment was conducted with the default throttling value of 67%. We ran the experiment with 50% (5*s* wake and 5*s* sleep) and 75% (5*s* wake and 15*s* sleep) throttling, the results of which are presented in Table 7. The average overhead values typically decrease as we increase throttling, but is not strictly the case. This aberration is an artifact of how hadoop assigns jobs to nodes and is not something we can explicitly control. While the overall CPU and memory utilization can be reduced using throttling, we observe that the time lag increases only marginally for increasing throttling values.

VM-ID	Total FS Notifications	Processed Notifications	Unique Data (MB)	Dedup Data (MB)	Transferred Data (MB)	Dedup
vm-1	93	67	0.66	910	90	10.11
vm-2	135	79	0.02	1136	270	4.20
vm-3	132	85	0.02	1617	67	24.13
vm-4	90	69	0.03	1174	669	1.75
vm-5	188	139	0.02	3326	1007	3.30
Total			0.7	8163	2103	3.88

Table 8. Comparison of dedup values with and without removing duplicate file writes on VM.

We summarize the deduplication information for the 5 VMs in Table 8. As noted earlier, this experiment has significantly fewer file writes, but each write is for a large chunk of data. This would mean that we will need to process most of the file writes as they are sufficiently separated in time from one another. The effectiveness of *I2Map* is demonstrated by the high volume of data in each VM identified as duplicate with at least one other VM. Further, compared to the total amount of data generated, 8163.7*MB*, the amount of data actually transferred is only 2103*MB* across all VMs, which is a reduction by a factor of 3.88 ($dedup\_aggr$).

Summary Our micro-benchmark experiments establish the effectiveness of *I2Map*. We are able to ensure an RPO of less than 3 minutes for VM and host failure, reduce the replication traffic by a factor of 2 to 7.5 ($dedup\_aggr$), while using less than 5% CPU and 400*MB* memory during periods of intense I/O loads. The reduction in replication traffic translates into network bandwidth savings of $50-87\%$ ($1-1/dedup\_aggr$) in the primary data center, compared to state-of-the-art host-based recovery solutions. We are able to reduce the number of file changes we process by a factor of 2 to 10 due

to change coalescing and need only 1 aggregator per 25 managed VMs. While our experiments were conducted with 6 VMs, having a larger pool of VMs using *I2Map* will only increase the deduplication possible. Under normal operation, we believe we can achieve even better performance at lower resource overheads.

5.2 Case Study

We conducted a 24 hour case study where we mimicked a real-world scenario where an application is running continuously at high load, is then brought down, the operating system is patched, rebooted, and the application is restored once again. At the end of the 24 hours we artificially failed one VM, which triggered recovery. In this section, we report results from this experiment, including *I2Map*'s recovery performance.

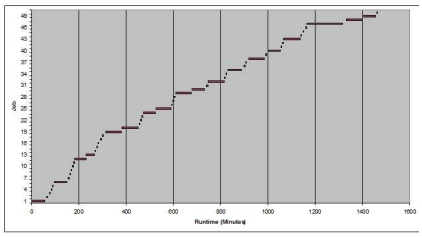


Fig. 9. Gannt chart showing duration of each hadoop run during the case study experiment.

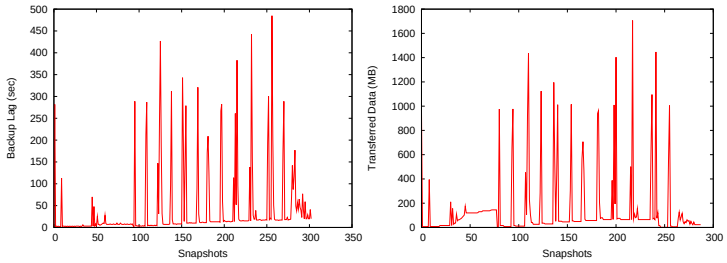
VM-ID	Avg Lag	Max Lag	Avg %CPU	Avg Mem
vm-1	49s	210s	0.50	115 MB
vm-2	62s	196s	0.65	138 MB
vm-3	52s	243s	0.65	57 MB
vm-4	76s	189s	0.71	113 MB
vm-5	58s	227s	0.69	113 MB
vm-6	75s	192s	1.01	112 MB

Fig. 10. Average time lag, and overhead for each VM.

We successively ran Teragen-Terasort on hadoop on the 6 VMs. Between every two runs of Teragen-Terasort we added a think time derived from a lognormal distribution with a mean of 120*s*. About 19 hours into the experiment, we brought down the hadoop application and started patching the VMs. This patch experiment was similar to the micro-experiment that we conducted, and lasted about 3 hours. Once patching of all VMs completed, we rebooted the VMs and restored the hadoop application. A Gannt chart showing the duration of each hadoop run and the think time between them is plotted in Figure 5.2. Overall, the hadoop-based application was running for 87.6% of the time. This is a very high load (as most enterprise systems run at a load of about 25%), created to stress-test *I2Map*. Notice the long sleep time between about 1150 and 1330 minutes, which was when the patching experiment was conducted.

Figure 10 shows the average and maximum time lag for agents to transfer files to the aggregator, as well as the average CPU and memory usage for each VM. We observe that the average lag is less than 1.5 minutes and the maximum lag at any instant is about 4 minutes. Thus, *I2Map* is able to achieve an RPO for host failures of approximately 4 minutes, even during periods of high write load. Including the time to transmit snapshots to the recovery site, the RPO for site failures is less than 9 minutes. This is very competitive compared to a best guarantee of 15 minutes provided by many commercial disaster recovery solutions [22, 8]. The average CPU usage was under 1% for all the VMs, and the memory usage was less than 140*MB*. Despite heavy load from the hadoop application, *I2Map* was able to operate with minimal overhead on the agents.

Snapshots of the aggregator were taken every 5 minutes and transmitted to the remote recovery site using Linux rsync [20]. Figure 11(a) shows the snapshot lag, the duration of time between when a snapshot was taken and when it was fully saved on the remote site, for each snapshot. This is primarily the delay over the network between



(a) Remote Backup Lag (b) Remote Data Transfer

Fig. 11. Snapshot Backup Lag and Data Transfer.

the primary site and the remote backup site. Observe that the maximum lag is about 7.5 minutes, which happens whenever a new run of hadoop is started and Teragen generates new data. For most snapshots the lag is negligible. Figure 11(b) shows the amount of data transferred for each incremental snapshot, which is about $1-1.7$ GB for the large spikes. Most snapshots transmit only about $100MB$ of data. The total data transferred across the Delhi-Bangalore WAN during the course of the experiment was about $26GB$. In comparison, the total data written on all the 5 VMs taken together was about $70GB$. The total aggregated deduplication $dedup\_aggr$ can be calculated as $70/26 = 2.69$.

At the end of 24 hours, we artificially failed a VM, which triggered recovery on the remote site. We mounted a copy of the golden master image corresponding to the VM, identified files belonging to the VM, and wrote their current version on to the mounted copy. This recovered the VM to its last known state. This process involved writing $14.88GB$ of data and took 719s, at $21.19MBps$. If a snapshot was being transferred to the remote site when the VM failed, then recovery may be delayed until completion of the transfer, adding up to 8 minutes to the recovery time. Hence, the total recovery time achieved by *I2Map* for this experiment can be estimated as $12-20$ minutes for a VM with plenty of writes, which is highly competitive with commercial DR solutions.

6 Related Work

Disaster recovery, as a concept, has existed for over three decades. Today, it forms the cornerstone of business continuity and every major IT service provider has a disaster recovery solution. These solutions require NAS/SAN arrays, storage controllers, smart network switches, or other specialized hardware. With the gaining popularity of the cloud, enterprises are looking to reduce their IT-spend and disinvest in hardware. In a bid to meet the expectations of their clients, cloud service providers are building low-cost clouds using commodity off-the-shelf hardware. In this section, we discuss the pros and cons of various replication and disaster recovery technologies. They can be broadly classified into storage-based, network-based, and host-based solutions.

There are several storage-based recovery solutions for cloud. Block-based storage replication requires expensive NAS/SAN arrays and storage controllers. But, they have the advantage of being independent of the operating system running on the server. Amazon's AWS provides multiple disaster recovery solutions that use Amazon S3 for backup [3]. These are either snapshot-based or storage replication solutions and do not perform any deduplication across VMs. IBM's GlobalMirror [13] provides an extremely high-end disaster recovery solution. It replicates all updates over a SAN and provides an RPO of 3-5 seconds. Other examples are VMware's Site Recovery Man-

ager [22] and IBM's SmartCloud Virtualized Server Recovery [11]. Notably, the lowest RPO guarantee provided by VMware's Site Recovery Manager [22] is 15 minutes. We have demonstrated that *I2Map*'s host-based solution can provide a comparable RPO, perform deduplication, and work using commodity hardware.

There are several disaster recovery solutions that do perform different kinds of deduplication. Dell's AppAssure [8] deduplicates and compresses data on the WAN while replicating storage disks. We argue that deduplicating data on the WAN is still too late as costly storage and network resources are consumed within the primary data center to support disaster recovery. NetApp's storage solutions [15] are specialized storage devices that perform deduplication using the Data ONTAP operating environment and the WAFL file system. They report that each write operation incurs a 7% additional overhead, in return for considerable savings in storage, which also translates into lower network bandwidth consumed when replicating the data across a WAN, using their SnapMirror solution [17]. However, deduplication can only be performed across VMs stored on the same storage device and comes with the cost burden of additional specialized hardware.

Network-based disaster recovery solutions perform deduplication on the bytes transmitted over the network. While useful, they do not leverage deduplication within the primary data center. Individuals VMs or servers are still required to transmit all their data across the local network. Some examples include Riverbed [19] and EMC's RecoverPoint [9]. Citrix cloud solution for disaster recovery [6] uses a combination of storage-based and network-based optimizations.

Host-based solutions have the advantage of not requiring specialized hardware and not locking the user into using a specific kind of storage device. Disadvantages include solution being dependent on the operating system used and having an agent running on the host and using its computing resources. Examples of existing solutions include CA's ARCserve [4] and Neverfail [16]. Neither of them perform deduplication on the primary data center. While ARCserve performs deduplication of data on the backup server (after the individual VMs have transmitted all their data), Neverfail uses what they call WANsmart in-line data deduplication, a form of network-based deduplication.

The concept of transmitting only the incremental changes relative to a base VM and dynamically synthesizing them at the time of provisioning has been used in the context of Cloudlets [10]. VM-based cloudlets have been proposed as offload sites for resource intensive or latency sensitive computations for mobile multimedia applications. The technique in [10] works by creating a binary difference between VM images, which is computed only on-demand when required. This is not a disaster recovery solution where continuous monitoring and data replication is desired.

In summary, there are a wide range of disaster recovery solutions that use a variety of technologies, have different requirements, and support different RTO and RPO guarantees. However, these solutions do not cater to the express need of low-cost clouds to support an efficient disaster recovery solution that can perform effective and early deduplication within and across VMs without transferring data, and work with commodity hardware. The *I2Map* disaster recovery solution presented in this paper addresses this concern, and its various optimizations ensure a competitive RPO and RTO guarantee along with low overhead on the VMs.

7 Limitations and Future Work

The disaster recovery solution presented in this paper caters to a specific need for having a low-cost, low-overhead solution that can work with commodity hardware. However, it does have its limitations. As with other host-based solutions, it requires an agent to be running on each VM, using up its computing resources. While we have demonstrated that the overhead can be contained to less than 5%, for many security-critical applications it may be inadmissible to have an agent (trusted as it might be) running on the VM. *I2Map* is not suitable for such applications. A majority of system management tools require agents (e.g., for monitoring, patching, backup) and we believe that having a well-tested agent with minimal performance impact may be acceptable to a large fraction of customers. Also, if the data is encrypted in the file system, *I2Map* will be unable to perform deduplication across VMs effectively. Security over the network is another issue faced by all DR solutions. This can be overcome by adding a layer of encryption before transmitting over network. Further, our current implementation of *I2Map* works only for linux-based VMs and new agents need to be developed for supporting any other operating systems.

Another issue that we have not investigated in this paper is the requirement and load on aggregators. If we were to scale up our disaster recovery solution to several hundred VMs, we may need more aggregators. This is an additional cost burden and we need ways to reduce the number of aggregators needed. The two round data transfer using aggregators does have its advantages, as it ensures that duplicate data is not transferred from protected hosts. Further, the aggregators separate the protected hosts from any WAN overheads, in case the transfer over the WAN were to be slow. As part of future work, we intend to study the costs and benefits of aggregators, especially at scale.

A limitation of all DR solutions (including *I2Map*) is that they only recover the state of the disk and not the memory. The state of memory is far more dynamic and one would have to quiesce any running applications in order to get a snapshot of memory. This is done in certain scenarios (e.g., live migration of a VM), but would be prohibitively expensive to perform on a regular basis and is not required by *I2Map*.

The notion of similarity captured by the *I2Map* tree can be used for performing other data center management tasks as well. The first is in troubleshooting software failures. For instance, system administrators routinely apply software upgrades and patches on a large set of VMs in the data center. If some of these upgrades fail, they have no clue to the cause of the failure. Analyzing the *I2Map* tree for similarities and differences between VMs could provide crucial insight into why the upgrade might have failed, and could even provide clues to how the situation can be remedied. Second, similarity between VMs as captured by the *I2Map* tree can also be used in assigning admins to VMs in a data center. Each admin could manage their VMs better, if they were all similar and had the same software. We intend to explore these applications of *I2Map* in our future work.

8 Conclusion

In this paper, we present *I2Map*, a host-based disaster recovery solution. *I2Map* leverages similarity across VMs in a data center and performs intra- and inter-VM deduplication to reduce the overhead of the solution. It maintains a mapping between instances

and the golden master image from which it was created as an *I2Map* tree, which captures all the changes to the instance with respect to the master image. Unlike existing disaster recovery solutions, *I2Map* does not require any expensive specialized storage devices or hardware. It separates deduplication and replication, allowing deduplication to be performed even before any data is transferred from a protected host. Extensive evaluation demonstrates that *I2Map* provides competitive recovery point and recovery time objective of the order of minutes, with low overhead.

References

1. IDC Linux Standardization White Paper: Executive Summary. http://www.redhat.com/f/pdf/IDC_Standard- ize_RHEL_1118_Exec_summary.pdf (2011)
2. Amazon: Summary of the AWS Service Event in the US East Region (2012), http://aws.amazon.com/message/67457/
3. Amazon: Using Amazon Web Services for Disaster Recovery. White paper (2012)
4. Associates, C.: ARCServe, http://www.arcserve.com
5. Campello, D., Crespo, C., Verma, A., Rangaswami, R., Jayachandran, P.: Coriolis: Scalable VM Clustering in Clouds. In: USENIX ICAC (2013)
6. Citrix: Citrix Cloud Solution for Disaster Recovery. White paper (2010)
7. Compuware: Performance in the cloud. White paper (2011)
8. Dell: AppAssure (2012), http://www.appassure.com/downloads/Dell_AppAssure_Specsheet.pdf
9. EMC, C.: EMC RecoverPoint Support for Cisco MDS 9000 SANTap Service: Intelligent Fabric-based Data Replication. White paper (2007)
10. Ha, K., Pillai, P., Richter, W., Abe, Y., Satyanarayanan, M.: Just-in-time provisioning for cyber foraging. In: ACM Mobisys. pp. 153–166 (2013)
11. IBM: SmartCloud Virtualized Server Recovery, http://www-935.ibm.com/services/in/en/it-services/smartcloud-virtualized-server-recovery-service.html
12. IBM: Tivoli Endpoint Manager. Online, http://www-01.ibm.com/software/tivoli/solutions/endpoint/
13. IBM: Global Mirror Whitepaper. White paper (2008)
14. Jayaram, K.R., Peng, C., Zhang, Z., Kim, M., Chen, H., Lei, H.: An Empirical Analysis of Similarity in Virtual Machine Images. In: ACM Middleware (Industrial Track) (2011)
15. NetApp: Back to Basics: Deduplication (2012), https://communities.netapp.com/docs/DOC-9949
16. NeverFail: Continuous Availability Suite: Neverfail Solution Architecture. White paper (2012)
17. Patterson, H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S., Owara, S.: Snapmirror: File system based asynchronous mirroring for disaster recovery. In: USENIX FAST (2002)
18. Rabin, M.O.: Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University (1981)
19. Riverbed Technologies: Riverbed Whitewater WAN Optimization and Steelhead Cloud Storage, http://www.riverbed.com
20. Tridgell, A., Mackerras, P.: The rsync algorithm. Tech. Rep. TR-CS-96-05, Australian National University (1996)
21. Viswanathan, B., Verma, A., Krishnamurthy, B., Jayachandran, P., Bhattacharya, K., Ananthanarayanan, R.: Rapid adjustment and adoption to MIaaS clouds. In: ACM Middleware, Industry track (2012)
22. VMware: vSphere Site Recovery Manager (2012), http://www.vmware.com/files/pdf/products/SRM/VMware-vCenter-SRM-Datasheet.pdf