Tradeoffs in Compressing Virtual Machine Checkpoints

Kai-Yuan Hou and Kang G. Shin
University of Michigan
{karenhou,kgshin}@eecs.umich.edu

Yoshio Turner and Sharad Singhal
HP Labs
{yoshio_turner,sharad.singhal}@hp.com

ABSTRACT

Checkpoint replication is a prevalent way of maintaining virtual machine availability in the presence of host failures. Since checkpoint replication can impose heavy load on network resources, checkpoint compression has been suggested to reduce network usage. This paper presents the first detailed evaluation and characterization of the effectiveness and overheads of checkpoint compression methods for various workloads frequently seen in high-availability systems. We propose a lightweight compression method that exploits similarities in checkpoints to eliminate redundant network traffic, and compare it with two well-known methods, gzip and delta compression. Our results show that gzip and delta compression reduce network traffic significantly for various workloads, but incur high CPU and memory overheads, respectively. The proposed similarity compression is most effective for VM clusters running homogeneous workloads, while using both CPU and memory efficiently. Based on our extensive evaluation, we suggest guidelines for selecting and using these compression methods.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

Keywords

Virtualization; Virtual Machine Checkpoints; Compression

1. INTRODUCTION

Physical host failures are common in large, virtualized data centers built with commodity hardware [29]. To maintain highly available virtual machines (VM) despite the occurrences of host failures, continuous checkpoint replication has been proposed [13, 28, 8]. It periodically captures the state of a VM in checkpoints, and replicates the checkpoints to a backup host. If the physical host of the VM fails, the VM can be restored from the most recent checkpoint available in the backup.

Checkpoint replication is more widely applicable to different hardware/software configurations in virtualized data centers, comparing to the other type of high-availability (HA) approach based on logging and replaying VM instructions [12, 6]; log-and-replay is limited to specific architectures and single-processor VMs, due to the complexity of deterministically replaying low-level VM events. However, checkpoint replication protects VMs at the expense of significant network traffic. Large amounts of checkpoint data are replicated over the network, especially when frequent checkpointing is used by client-facing, latency-sensitive applications to checkpoint network packets before sending them out. For example, if a checkpoint is taken every 25 ms, replication can consume more than 3 Gb/s of network bandwidth for a single VM. If multiple VMs must be protected at the same time, even dedicated GbE links cannot provide the aggregate bandwidth required for checkpoint replication. Reducing replication traffic is therefore crucial to using checkpoint replication in real-world systems for HA.

One way of reducing replication traffic is to "compress" checkpoints before sending them over the network. Checkpoint compression requires no modifications to the VM, and can be applied regardless of the workloads running in the VM. It can be done by a general-purpose tool, such as gzip. Alternatively, for each dirty memory page in a checkpoint, the bits that are actually changed (called the page *delta*) may be identified, and only the delta is replicated [21, 24, 27, 31]. These compression methods are available, but they have not been compared systematically. There are few guidelines for selecting and using them under different workloads and operating conditions in a HA system.

The primary goal of this paper is to quantify the tradeoffs between the effectiveness and overheads of various checkpoint compression methods, and provide insights that could guide selection decisions. We compare three compression methods, including gzip, delta compression, and a new method we propose, called *similarity compression*. Similarity compression exploits the inherent content redundancy in VM memory [30, 16]. It finds and eliminates redundant contents particularly in the *changed* set of VM pages, *i.e.*, the VM checkpoints, to reduce replication traffic.

We evaluated the three methods using workloads chosen from types frequently seen in HA systems, including server workloads that constantly interact with external clients and

The work reported in this paper was supported in part by the Air Force Office of Scientific Research under Grant No. FA9550-10-1-0393 and by HP Labs.

long-running computation jobs. Our results show that no single compression method is best suited for all types of workloads and operating conditions. gzip reduces checkpoint traffic substantially, but at a prohibitive CPU cost. It also incurs the longest checkpoint transfer times, which lowers the achievable checkpointing frequency and makes it unsuitable for interactive, latency-sensitive applications. Delta compression incurs a low CPU overhead. However, it requires a cache larger than the average checkpoint size of the protected VM to achieve reasonable traffic reductions. For workloads that touches large areas of memory rapidly, hundreds of MBs of RAM must be provisioned for the delta cache, creating a significant memory overhead.

Similarity compression eliminates redundant contents *within* checkpoints of the same VM (intra-VM similarity), and *between* checkpoints of different VMs on a host (inter-VM similarity). Our evaluation shows that non-trivial VM similarity exists in VM clusters running homogeneous workloads, such as HPC clusters, especially when the VMs in a cluster collaborate with one another. In such cases, similarity compression achieves effective traffic reduction using both CPU and memory efficiently. However, limited similarity is found between VMs running heterogeneous workloads, for which gzip and delta compression are better suited.

The contribution of this work is threefold: First, it proposes similarity compression, a resource-efficient alternative, especially for use in homogeneous workload scenarios. Second, to our best knowledge, it presents the first detailed evaluation and characterization of checkpoint compression methods, considering gzip, delta and similarity compressions. Third, based on the evaluation results, it suggests guidelines for selecting and using these compression methods for different workload types and resource constraints.

The remainder of the paper is organized as follows. Section 2 provides background on checkpoint replication. We describe our evaluation framework and metrics in Section 3, and the three compression methods we evaluate in Section 4. In Section 5 we present the experimental results and discuss the insights from the results. Section 6 discusses related work, and the paper concludes with Section 7.

2. CHECKPOINT REPLICATION

2.1 Background

Checkpoint replication protects a VM from the failure of its physical host by sending checkpoints of the VM to a backup host continuously [13, 28, 8]; the backup is chosen so that it is immune to the failure of the protected host. When protection begins, a full checkpoint containing every memory page and the CPU state of the protected VM is replicated to the backup. It is stored in the backup's RAM, and becomes the *fail-over image* of the protected VM. [1]

As the VM executes, *incremental* checkpoints are taken and replicated to the backup, usually at fixed time intervals (a pre-configured checkpointing frequency.) An (incremental) checkpoint mainly consists of the VM pages dirtied during the last checkpointing interval. After all dirty pages in a checkpoint are replicated to the backup, their contents are stored in proper locations in the fail-over image according to the page indexes. The fail-over image is not updated as each dirty page is received, since if the protected host fails in the middle of sending a checkpoint, the image becomes inconsistent and unusable for recovery. If a checkpoint takes longer than the configured checkpointing interval to replicate, the subsequent checkpoint is not taken when the next interval begins, but delayed until the on-going replication finishes. This way, checkpoints are not sent faster than they can be stored and made useful.

[1]VM disks are usually hosted in a shared storage accessible to all VM hosts. A disk state consistent with the memory and CPU state in the fail-over image may be maintained by the storage system using copy-on-write techniques [3, 20].

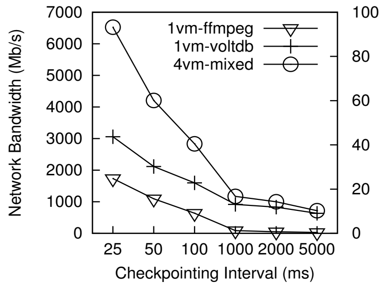


Figure 1: The bandwidth requirements and total traffic of checkpoint replication for two minutes.

Once a failure of the VM host is detected, the HA system initiates a fail-over. The failed VM is restored based on its fail-over image (in the backup's RAM) and a consistent disk state (in a shared storage), and resumes operation from the most recent checkpointed state in the backup host. In order to make this fail-over transparent to the VM's external clients, the HA system ensures that the clients never see "unprotected" VM state—the state yet to be backed up. Specifically, during normal operation of the VM, outgoing network packets are withheld until the checkpoint capturing the state from which the packets are generated is fully replicated to the backup. It is therefore common that HA systems use checkpointing intervals of tens of milliseconds or even shorter, to checkpoint and release network packets very frequently and achieve reasonable application performances.

2.2 The Need of Compression

We ran different workloads in VMs, and replicated VM checkpoints for two minutes of workload executions. Figure 1 shows the network bandwidth required and the total traffic generated by checkpoint replication (see Table 1 for workload details.) When checkpoints are replicated every 25 ms, protection of a transcoding (ffmpeg) server and a database (voltdb) server uses more than 1700 Mb/s and 3000 Mb/s of network bandwidth, respectively. Even a dedicated GbE link cannot meet such bandwidth requirements for protection of a single VM. When 4 VMs are protected concurrently, replicating checkpoints every 25 ms create almost 100 GB of traffic in the network in only two minutes. To replicate these checkpoints at the configured checkpointing frequency, over 6500 Mb/s of network bandwidth is required. Even if a 10GbE link is available, it soon becomes saturated with just a few more VMs to protect.

The prohibitive network requirements of checkpoint replication can use up all available resources and interfere with normal VM traffic, degrading application performance and users' experiences of the VMs. It is therefore crucial to compress and reduce replication traffic when providing VM protection. This paper proposes compressing checkpoints by VM similarities, and evaluates it with two other compression methods systematically, to provide guidelines for their selections. Next, we describe how each method is evaluated.

3. EVALUATION METHODOLOGY

3.1 Framework

To facilitate systematic comparison of multiple compression methods, we took an emulation approach in our evaluation. We built a framework consisting of a checkpoint sender (`emulhacp`) and a checkpoint receiver (`emulharcv`), which emulate the replication and storage of VM checkpoints in a real HA system, as described in Section 2.1. Different compression methods are implemented as modules inserted into the framework for evaluation.

To use this framework, we capture VM checkpoints *a priori* in a real HA system [17], and store them as individual files. `emulhacp` runs in a protected host. It reads a complete checkpoint from file into a memory buffer, and from then operates on the buffer. It processes each dirty page using the compression method to be evaluated, and then sends the page to `emulharcv`, which runs in the backup. Once all dirty pages are received, `emulharcv` sends an ACK to `emulhacp`, and begins to decompress each page and store the page content to the fail-over image kept in RAM.

After `emulhacp` receives the ACK, it waits until the current checkpointing interval ends, and replicates the subsequent checkpoint when the next interval begins. If an ACK is not received by the beginning of the next interval, `emulhacp` waits until the on-going replication finishes, and then replicates the subsequent checkpoint immediately. In that case, `emulharcv` is receiving a new checkpoint while storing the one just received at the same time.

3.2 Metrics

We evaluate the average **traffic reduction** achieved in each checkpointing interval, and the memory and CPU used to achieve such reduction. If compression uses excessive resources in the protected host, it can greatly interfere with the normal operation of the protected VMs. The resource usage in the backup is also considered, since other active VMs may be running in the host (and backed up elsewhere) and their performances can be affected.

A compression method's **memory cost** is evaluated by the average memory usage of its key data structures which enable page compression/decompression. **CPU cost** is evaluated by a per-page metric. We measure the total CPU time `emulhacp` takes to compress and send checkpoints for all concurrently protected VMs. We then divide this time by the number of dirty pages processed, and obtain the average CPU time spent for each page. Likewise, we obtain the average CPU time `emulharcv` takes to receive, decompress and store each page in the backup. These per-page metrics facilitate fair comparison of different compression methods.

Compression also affects the **transfer time** of each checkpoint, which starts when `emulhacp` begins to send the checkpoint, and ends when an ACK for the checkpoint is received. Checkpoint transfer time consists of two components: the time to process/compress the dirty pages (processing time), and the time to send them over the network (sending time). Replicating compressed checkpoints reduces sending time, but performing compression lengthens processing time. The overall effect of compression on checkpoint transfer time needs to be quantified experimentally.

Checkpoint transfer time is an important metric because it has a direct impact on feasible checkpointing frequencies, which in turn affects the performance and HA properties of a protected VM. Since a subsequent checkpoint may be replicated only after the on-going replication finishes, the actual (*elapsed*, not configured) checkpointing interval must be longer than the transfer time of a checkpoint. If a compression method incurs long transfer times, consecutive checkpoints must be separated by large intervals, thus lowering the checkpointing frequency achieved. This can degrade the performance of latency-sensitive, server applications severely during normal operations, since network packets are checkpointed and released infrequently. For computation jobs without external observers, lowering checkpointing frequency results in a greater loss of completed work upon a fail-over, since the VM has to resume execution from an earlier point in time.

4. CHECKPOINT COMPRESSION

We evaluated the following three compression methods by the framework and metrics discussed in Section 3.

4.1 Existing Techniques

gzip is a commonly-used, general-purpose compression algorithm. Its application on checkpoint traffic was briefly discussed in [13] without a thorough evaluation.

Delta compression identifies the parts in a dirty page that are changed when the page is written to, *i.e.*, the page delta, and replicates the delta to the backup instead of the entire page. It has been used in a few HA systems [21, 24].

Before sending a checkpoint, each dirty page is XOR'ed with its content in the last checkpointing interval. The outcome is compressed by RLE (Run-Length Encoding), and the compression result is sent to the backup. To restore the page content in the backup, the RLE result is decoded, and the outcome is XOR'ed with the content of the page in the fail-over image; no extra memory copying is needed. Since keeping the prior content of every page incurs a 100% memory overhead, like previous work, we maintain a fixed-size cache of transmitted dirty pages. If a dirty page finds its prior content in the cache (*i.e.*, a cache hit), delta compression is performed. Otherwise, the entire page is replicated without compression. We implemented a LRU cache using a double linked list for efficient replacement and a lookup array to speed up queries.

4.2 Exploiting VM Similarity

Since VMs in a virtualized data center are often created from template images consisting of the same or similar operating systems and applications, they can load nearly identical kernel images and software binaries into memory, and read duplicate data from common files. Many systems use these content redundancies to enable page sharing [30, 16, 22]; they detect redundancies in the entire memory of colocated VMs, and coalesce identical pages in the same physical frame to save host memory.

We argue that not only may VMs be created from similar sources, even as they execute, various activities can keep *changing* their memory state in similar ways. For example, during maintenance, a group of VMs is updated with the same set of security patches at the same time. Also, in

HPC-C [2]	A suite of 7 calculation-intensive benchmarks essential to long-running scientific jobs. We run it in a single VM, and also multiple concurrently checkpointed VMs collaborating via MPI. Each VM uses 512 MB RAM.
RUBiS [4]	An auction site benchmark modeled after ebay.com. We use a three-tier setup: a back-end database, a RUBiS server (Apache/PHP) and a client emulator. Each tier runs in a separate VM on a separate host. Our experiments use checkpoints of the RUBiS server (a 512 MB VM).
FFmpeg [1]	An open-source tool to "transcode" video/audio files, *i.e.*, to convert their codecs and formats; the transcoded media are usually fed to a real-time streaming service to meet external clients' various requests. We run FFmpeg in 512 MB VMs.
VoltDB [7]	An in-memory database. We run it in a 1.75 GB VM to support a TPC-C-like workload generated from a client VM running in a separate host. This OLTP workload simulates an order-entry environment for a business with multiple warehouses [5]. Our experiments use checkpoints of the VoltDB server.

Table 1: The workloads used in our evaluation and their setup.

computing clusters, multiple VMs (and multiple processes in each VM) collaborate to finish computation-intensive tasks, each running the same application and working on a common set of data. These VMs are often checkpointed at the same time intervals to gain a comparable level of protection, and the similar changes made to their memory generate similar dirty pages in their checkpoints. We therefore propose **similarity compression**, to find content redundancy particularly in the *changed* set of memory pages, *i.e.*, the VM checkpoints, and send only one copy of the duplicate contents to reduce replication traffic.

To detect content redundancy at a finer granularity, we divide each dirty page into multiple *chunks*, and process checkpoints by chunks. *Unique chunks* are separated from *duplicate chunks*. A unique chunk contains a content different from any other chunks that have been processed. This content must be replicated to the backup in full. A duplicate chunk contains a content that is identical to at least one other chunk. Since the duplicate content can be found in another chunk that is already replicated (called a *reference chunk*), instead of sending the content again, for each duplicate chunk, we send a pointer to locate its reference chunk in the backup.

For similarity compression to be practically useful, two important requirements must be met: (1) unique and duplicate chunks must be separated quickly, and (2) the pointers sent for duplicate chunks must be small, yet contain enough information to restore the duplicate contents in the backup. To meet these requirements, we build a hash table in the protected host. The hash table maps a chunk *content* to a chunk *location* that has the content. Chunk location is described by the VM to which the chunk belongs and the chunk's offset in the checkpoint containing the chunk. We use MD5 digests to compactly represent and efficiently compare chunk contents. We have also explored detecting content redundancy in checkpoints using Rabin fingerprints over a sliding window, and found that to be much slower. For efficiency, we choose to detect duplicate fixed-size chunks by hashes.

In each checkpointing interval, the hash table is initially empty, and the checkpoints taken for concurrently protected VMs are processed together. For each chunk in the checkpoints, we compute its MD5 digest, and query the hash table by the digest. If the digest is not found in the hash table, the chunk content is sent to the backup, since this is a unique content that is not seen before. A new entry is inserted into the hash table to record the chunk's content and location.

If the hash table lookup finds the chunk digest, we have a duplicate chunk, and the matching hash entry records the location of a replicated chunk with the same content, *i.e.*, the reference chunk. The location of the reference chunk (encoded in 4 bytes in our implementation) is sent to the backup as a pointer. When incorporating the checkpoint into the fail-over image, the duplicate content is retrieved following the pointer and stored to the image. The reference chunk may belong to the same, or a different VM than the duplicate chunk, upon detection of *intra-* and *inter-VM* similarity, respectively.

Checkpoint Sizes (MB)	Checkpointing Intervals (ms)					
	5000	2000	1000	100	50	25
HPC-C	19.3	18.2	13.1	3.1	2.1	1.7
RUBiS	13.0	11.4	8.2	4.4	4.1	3.7
FFmpeg	19.0	12.9	10.8	8.0	6.8	5.4
VoltDB	396.7	207.5	114.4	20.0	13.2	9.6

Table 2: The average checkpoint sizes of each workload.

5. EXPERIMENTAL RESULTS

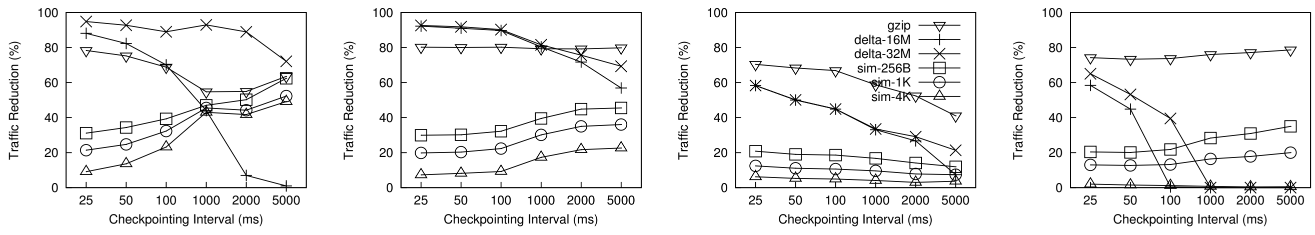
This section presents and analyzes our evaluation results of the three compression methods. We first describe the workloads and testbed used in our evaluation in Section 5.1. Section 5.2 evaluates the traffic reductions achieved by each compression method. Section 5.3 and Section 5.4 evaluate the associated resource and time overheads, respectively.

5.1 Workloads and Testbed

The computation tasks needing HA most are the ones that are not repeatable or prohibitively expensive to repeat after a failure occurs. These tasks include server workloads that constantly interact with external clients, and long-running computing jobs such as scientific computations. Our evaluation uses four different workloads of these types. Table 1 summarizes the workloads we use and their setup.

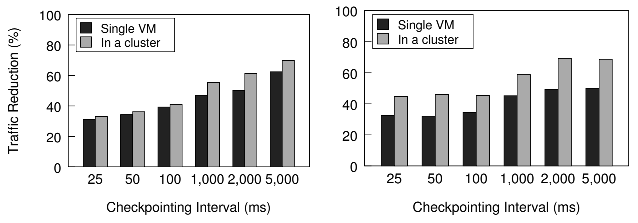
We run the workloads in VMs, take checkpoints for two minutes of workload execution, and store the checkpoints taken for repeated use in our various experiments. [2]For each workload we capture multiple series of checkpoints, and in each series we use a different checkpointing frequency. We use sub-second (25, 50 and 100 ms) and one-second checkpointing intervals to reflect those used in current HA systems [13], and also multi-second (2 and 5 secs) intervals to explore a wider parameter space. For two minutes of workload execution, checkpointing every 5 seconds to every 25 ms generates 24 to 4800 checkpoints in each series. Table 2 summarizes the average checkpoint sizes of the differ-

[2]Checkpoints are taken on the prototype of our prior work, a disk-based HA system for VMs [17].



(a) HPC-C (b) RUBiS (c) FFmpeg (d) VoltDB

Figure 2: The average traffic reductions achieved for a single checkpointed VM running different workloads.



(a) An Independent HPC-C (b) A Collaborating HPC-C

Figure 3: The traffic reductions achieved by processing a HPC-C VM's checkpoints alone and with other VMs' checkpoints in a cluster.

ent workloads. While in some cases individual checkpoints seem small, especially when taken at short intervals, sending these checkpoints frequently creates excessive network traffic in only two minutes, as discussed earlier in Section 2.2.

All checkpoints are taken on HP Proliant BL465c blades, each with two dual-core AMD Opteron 2.2GHz CPUs, 4–8GB RAM, one GbE interface and two SAS 10K rpm disks. All our experiments are run on the same testbed. The blades in our testbed are in the same LAN, resembling a typical setup in data centers where protected and backup hosts are connected by an internal LAN for management operations. We run `emulhacp` and `emulharcv` on top of Xen in the Domain-0 of two separate blades. In each experimental run, the programs process complete series of checkpoints, and report average results over the checkpoints processed.

5.2 Traffic Reduction

We first evaluate how each compression method reduces checkpoint traffic for a single protected VM. Figure 2 shows the traffic reductions achieved when different workloads are running in the VM. *gzip is generally effective for various workloads and checkpointing frequencies.* It reduces traffic by more than 70% in most cases. However, it compresses less effectively for FFmpeg and HPC-C at checkpointing intervals of one second and larger. These FFmpeg checkpoints consist of many media contents that are already encoded by video/audio codecs, and hence are not compressed much further by gzip. HPC-C checkpoints contain many numerical values from the workload's computation matrices. The randomness of these values is not particularly friendly to the compression algorithm of gzip.

The effectiveness of delta compression varies widely for different workloads and checkpointing frequencies, and is mainly impacted by how the delta cache is sized in relation to the size of the checkpoints. We start our experiments with a 32MB cache, which is large enough to store at least one complete checkpoint for most of the workloads and checkpointing frequencies we use. To test the sensitivity of traffic reduction to cache size, we also evaluate a smaller (16MB) cache.

The results show that a 16MB cache is already effective for RUBiS, especially when checkpointed at sub-second intervals. In these cases, delta compression outperforms gzip and achieves up to 92% traffic reduction, since the cache can hold at least 5 consecutive checkpoints. Keeping a good history of the workload's dirty page contents, the cache produces over 98% hit rates, letting almost all checkpointed pages be compressed before transmission. On the other hand, a 32MB cache is still far from enough for VoltDB. At one-second and longer intervals, the checkpoint traffic of VoltDB is hardly reduced, since each of these checkpoints is much larger than the cache; almost all cached dirty pages are replaced before enabling any compression.

In these experiments, similarity compression uses only intra-VM similarities—traffic is reduced by removing the redundant checkpoint contents *within* each VM, since only one VM's checkpoint traffic is processed at a time. We evaluate three chunk sizes for similarity compression: 256, 1K and 4K bytes. 256-byte chunks always reduce traffic more effectively than 1K chunks, which are, in turn, more effective than 4K chunks.

While similarity compression achieves lower traffic reductions than the other two methods, it performs particularly well for HPC-C. Using 256-byte chunks, traffic is reduced by 46–62% at one-second and longer intervals.We initially suspected that much of the reduction comes from the elimination of zero pages, generated upon memory allocations by the workload. An off-line analysis showed that less than 2.5% of these checkpointed pages contain all zeros. Thus, most of the redundant contents are non-zero workload data, which are likely duplicated in the multiple processes that HPC-C spawns in the VM to collaborate on the workload's computation problems. These results suggest that *similarity compression is particularly effective for workloads that have multiple components collaborating on a shared set of data.*

5.2.1 Multiple Concurrently Protected VMs

We then apply the compression methods to the checkpoint traffic of four VMs simultaneously, and evaluate the traffic reductions achieved in the following two scenarios: (S1) a HPC cluster, representative of a homogeneous workload environment; and (S2) A heterogeneous mixture of workloads. In S1, each of the four VMs runs an instance of HPC-C. VM1 and VM2 work independently on separate problem sets. VM3 and VM4 collaborate on a larger set of problems. For S2, we use the combination of the HPC-C, RUBiS,

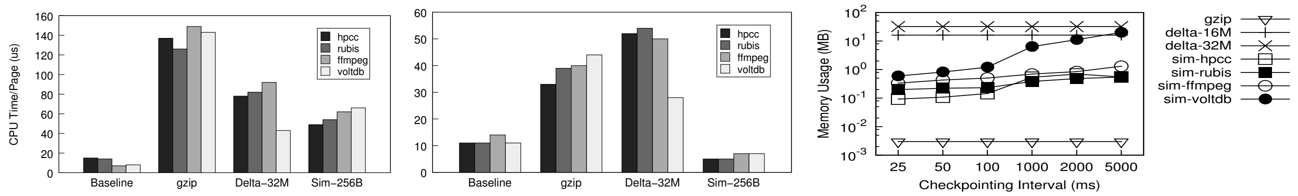

(a) CPU costs in the protected host (b) CPU costs in the backup host (c) Memory costs in the protected host

Figure 4: The resource costs of replicating checkpoints of the single checkpointed VMs (showing the CPU costs of 100ms checkpointing intervals; results of the other intervals show consistent trends.)

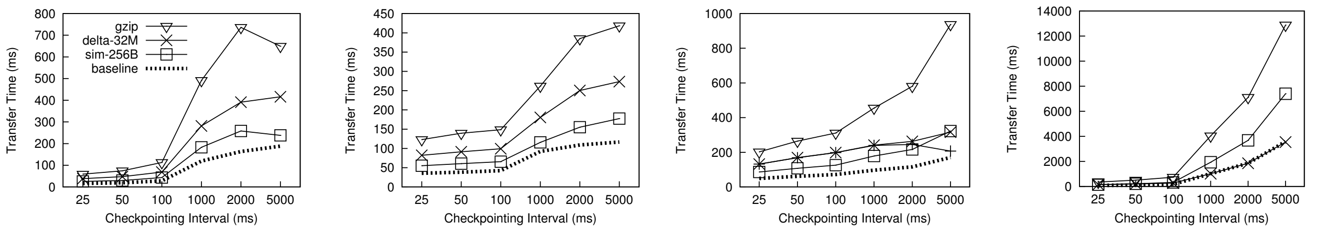

(a) HPC-C (b) RUBiS (c) FFmpeg (d) VoltDB

Figure 5: The average checkpoint transfer times of the single checkpointed VMs.

FFmpeg and VoltDB checkpoint series which have been individually analyzed in the previous discussion of single checkpointed VMs. In both scenarios, the VMs are co-located in a single protected host, and their checkpoints are captured at the same frequency.

In the three methods evaluated, similarity compression is the only one that may achieve additional traffic reductions when processing checkpoints of multiple VMs concurrently, since dirty page contents duplicated across VM boundaries are also eliminated. To understand the effect of exploiting inter-VM similarity, we choose an individual VM, and compare the reductions achieved for the VM when its checkpoints are processed alone versus with other VMs' checkpoints in a cluster. This is more reasonable than simply comparing the overall reductions achieved in a single- and a 4-VM scenario, since the overall reduction observed in a multi-VM cluster is biased by the individual VMs' checkpoint sizes. We ask the following questions: How much benefit does inter-VM similarity provide in a homogeneous workload environment for improving traffic reductions? In such an environment, is there any difference if the VMs collaborate on the same tasks? On the other hand, is there any inter-VM similarity in a heterogeneous environment?

Figure 3(a) shows the traffic reductions achieved by similarity compression for VM1 in scenario S1, which works on a HPC problem set independently. When the VM's checkpoints are processed with the other three VMs', an additional 2–11% reduction is achieved. Figure 3(b) shows the reductions achieved for VM3 in scenario S1, which collaborates with another VM on the same HPC problem set. Processing this VM's checkpoints in a cluster yields a greater additional traffic reduction of 11–20% more. Even at subsecond intervals, a meaningful degree of inter-VM similarity is observed between collaborating VMs.

Non-trivial inter-VM similarity exists and enables greater traffic reductions in a homogeneous workload environment, especially when VMs collaborate on a common task set. However, limited similarity is found between VMs running heterogeneous workloads. We found in the workload mixture of scenario S2 that each VM's checkpoint traffic is hardly further reduced when their checkpoints are processed together versus separately; the greatest improvement of traffic reduction in the cluster is observed with RUBiS at only 6% more.

5.3 CPU and Memory Costs

Figure 4 shows the resource requirements of each compression method. *While reducing traffic effectively, gzip incurs the largest CPU overhead for compression.* This prohibitive CPU cost can greatly interfere with the normal VM operations in the protected host, especially when checkpoints of multiple VMs are processed by gzip concurrently.

Delta compression consumes excessive RAM in the protected host to cache transmitted dirty pages. From our evaluation, in order to achieve reasonable traffic reductions, the delta cache must be large enough to store at least one complete checkpoint taken for the protected VM. Although our experiments evaluate fixed size caches of tens of MBs, in practical use, for workloads that touch large areas of memory rapidly (like VoltDB), a cache of hundreds of MBs of RAM or even larger must be provisioned. As more VMs become protected, the total memory cost quickly grows, creating memory pressure in the protected host.

Similarity compression uses both CPU and memory efficiently. Even using 256-byte chunks (computing 16 digests per page), its CPU cost is the lowest of the three methods evaluated. Low memory overheads are incurred, ranging from 95KB to 1.3MB with 256-byte chunks and less (8–350KB) when 1KB and 4KB chunks are used, except for VoltDB, which has exceptionally large checkpoints and many unique chunks in each checkpoint. Similarity compression uses less memory in the presence of greater VM similarity, since fewer unique chunks need to be stored in the hash table. In our 4-VM HPC cluster (scenario S1 in Section 5.2.1), significant intra- and inter-VM similarity exist, and the hash table uses less than 2MB of memory at all times to process checkpoints of all four VMs concurrently.

5.4 Checkpoint Transfer Time

Figures 5(a)–5(d) show the checkpoint transfer time of the single checkpointed VMs. For all workloads, transfer time is the shortest when compression is not used (baseline). *gzip incurs the longest transfer time.*Even though gzip sends only 30% of the original checkpoint data in most cases, transfer time becomes up to 14x longer than the configured interval, due to performing compression. As a result, checkpointing frequencies of every 100ms and higher are no longer feasible. For VoltDB, none of the evaluated checkpointing frequencies can be achieved when gzip is used.

Such long transfer time lowers the checkpointing frequencies achievable with gzip, and limits its use for server applications that are very sensitive to network latencies. Delta and similarity compressions are better suited for these applications. In most cases, similarity compression takes about 1.5x less time than delta compression to replicate a checkpoint using 256-byte chunks. We observed that similarity compression requires even shorter transfer time when processing checkpoints by larger (1K and 4K) chunks. For VoltDB, delta compression incurs the shortest transfer time, since few compressions are performed with a 32MB cache; especially at one-second and longer intervals, the behavior of delta compression is almost like that of the baseline.

5.5 Discussions

From our evaluation, there is hardly a single best compression solution. gzip and delta compression reduce checkpoint traffic substantially, but incur a prohibitive CPU and memory cost, respectively. Similarity compression usually achieves smaller traffic reductions. However, it reduces traffic very effectively for VMs collaborating on a shared task set, using both CPU and memory efficiently.

Compression methods should be selected based on the workload types and resource constraints in the target environment. Our results suggest that for VM clusters running homogeneous workloads, similarity compression is particularly suitable. Since non-trivial intra- and inter-VM similarity exist, significant traffic reductions can be achieved at low overheads. For other workload scenarios, especially a heterogeneous mixture of workloads, gzip and delta compression are better candidates. They are effective for a wider range of workload types, although heavier-weight comparing to similarity compression. The resource availability in the target environment is an important factor to consider when selecting these methods.

To use gzip, the protected host must have sufficient CPU to support checkpoint compression in addition to normal VM operations. Since gzip is bottlenecked on compression rather than transmission of checkpoints, its usefulness greatly degrades if processing checkpoints of multiple VMs creates severe CPU contention. This not only affects normal VM operations, but further lengthens the time taken to replicate each checkpoint. The protected VMs thus must be checkpointed at even lower frequencies, making gzip almost unusable for server applications.

For delta compression to be effective, sufficient memory must be available in the protected host. Our results suggest that an effective delta cache is usually larger than the average checkpoint size of the protected VM. A proper cache size may be determined by profiling the target workloads *a priori*. Alternatively, the cache may first be over-provisioned, and dynamically adjusted as the workloads execute.

6. RELATED WORK

Two types of approaches have been proposed to reduce checkpoint replication traffic in HA systems. One reduces the amount of VM state to protect/checkpoint. RemusDB [21], a highly available database system in VM, does not checkpoint clean disk buffers, and "de-protects" certain data structures in the database system that can be regenerated after a failure. This creates smaller checkpoints and thus less replication traffic in the network. However, it requires in-depth understanding of the applications running in the protected VMs to identify data structures that can be safely de-protected. The VM and applications must also be instrumented to recover un-checkpointed state after a fail-over.

The other type of approaches reduces the amount of data sent for each checkpoint taken. Checkpoint compression is generally applicable regardless of the applications in the VMs, and requires less system instrumentation; hence it is the focus of our study. The authors of [13] briefly discussed compressing checkpoints by gzip and delta compression, although a thorough evaluation was not included. RemusDB [21] and SecondSite [24], designed for database HA and datacenter disaster recovery, respectively, use delta compression in their systems. They both find page delta by XOR, and compress the delta by RLE, like evaluated in this paper. Lu *et al.* propose fine-grained dirty region tracking (FDRT) [19], which shares the same concept of delta compression. FDRT divides each dirty page into fixed-size regions, and replicates only the regions that are modified to the backup. Delta compression has also been used to reduce VM live migration traffic [27, 31].

Similarity compression exploits memory content redundancy to reduce checkpoint replication traffic. Different from memory sharing systems which find and coalesce redundant pages in the entire memory of co-located VMs [30, 16, 22], similarity compression finds redundant data in dirtied memory pages particularly. It also detects redundancy continuously and at much higher frequencies comparing to redundancy elimination during VM live migration [26, 31]. Note that similarity compression examines the similarity in the memory of live, executing VMs continuously, different from VMFlock [9], which utilizes the similarity in VM disks during migration of static VM images. Instead of finding "identical" data in dirty pages, Gerofi *et al.* find memory areas that are "similar" to the dirty pages, and send only the differences between the dirty pages and these memory areas to reduce checkpoint replication traffic [15]; their approach is currently applied to each VM independently. The same idea has also been used to reduce VM migration traffic [32, 14]. In a broader context, content redundancy is widely used for storage deduplication to improve I/O performance [23, 25, 18], and in network infrastructures to improve network capacity and end-to-end application performance [10, 11].

7. CONCLUSIONS AND FUTURE WORK

Reducing checkpoint traffic is crucial to using checkpoint replication for maintaining VM availability. In this paper, we propose similarity compression to reduce traffic by eliminating redundant checkpoint contents, and evaluate it with gzip and delta compression, characterizing the compression methods based on their effectiveness and overheads. We find no single best solution that suits all workloads and operating conditions, and our characterization suggests that compression methods can complement one another in a hybrid approach. We are exploring hybrid compression approaches that combine a lightweight technique, like similarity compression, with another heavier-weight one, such as gzip. For example, coarse-grained similarity compression can achieve a meaningful reduction of checkpoint sizes at a low computing overhead. The remaining checkpoint data can then be compressed greater and faster by gzip. We are also extending similarity compression to detect content redundancy across checkpointing intervals, by increasing hash table lifetime, and to exploit VM similarity beyond host boundaries, by building a distributed hash table.

8. REFERENCES

[1] FFmpeg. `http://www.ffmpeg.org`.

[2] HPC Challenge. `http://icl.cs.utk.edu/hpcc`.

[3] LVM2. `http://sourceware.org/lvm2`.

[4] The RUBiS benchmark. `http://rubis.ow2.org`.

[5] The TPC-C-like benchmark of VoltDB. `http://community.voltdb.com/node/134`.

[6] VMware fault tolerance (FT). `http://www.vmware.com/products/fault-tolerance`.

[7] VoltDB. `http://community.voltdb.com`.

[8] A. Agarwal, D. Shah, N. Kalmala, N. Panchaksharam, R. Bharadhwaj, S. Lokray, S. Sm, and T. Bean. Method and apparatus for transactional fault tolerance in a client-server system, Oct. 2009. Patent, US 7610510.

[9] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. VMFlock: virtual machine co-migration for the cloud. In *Proc. of the 20th Symp. on High Performance Distributed Computing*, 2011.

[10] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proc. of the SIGCOMM Conf.*, 2008.

[11] A. Anand, V. Sekar, and A. Akella. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *Proc. of the SIGCOMM Conf.*, 2009.

[12] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. on Computer System.*, 14(1), 1996.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchisonson, and A. Warfield. Remus: High-availability via asynchronous virtual machine replication. In *Proc. of the 5th Symp. on Networked Systems Design and Implementation*, 2008.

[14] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *Proc. of the Symp. on High Performance Distributed Computing*, 2011.

[15] B. Gerofi, Z. Vass, and Y. Ishikawa. Utilizing memory content similarity for improving the performance of replicated virtual machines. In *Proc. of the 4th Conf. on Utility and Cloud Computing*, 2011.

[16] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. of the 8th Symp. on Operating Systems Design and Implementation*, 2008.

[17] K.-Y. Hou, M. Uysal, A. Merchant, K. G. Shin, and S. Singhal. HydraVM: Low-cost, transparent high availability for virtual machines. Technical report, HP Labs, 2011.

[18] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing content similarity to improve I/O performance. In *Proc. of the 8th Conf. on File and Storage Technologies*, 2010.

[19] M. Lu and T.-C. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Proc. of the 39th Conf. on Dependable Systems and Networks*, 2009.

[20] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Proc. of the 3rd EuroSys Conf.*, 2008.

[21] U. F. Minhas, S. R. B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent high availability for database systems. *PVLDB*, 4(11), 2011.

[22] D. G. Murray, S. H, and M. A. Fetterman. Satori: Enlightened page sharing. In *Proc. of the USENIX Annual Technical Conference*, 2009.

[23] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. of the 1st Conf. on File and Storage Technologies*, 2002.

[24] S. Rajagopalan, B. Cully, R. O'Connor, and A. Warfield. Secondsite: disaster tolerance as a service. In *Proc. of the 8th Conf. on Virtual Execution Environments*, 2012.

[25] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proc. of the USENIX Annual Technical Conference*, 2008.

[26] P. Riteau, C. Morin, and T. Priol. Shrinker: improving live migration of virtual clusters over WANs with distributed data deduplication and content-based addressing. In *Proc. of the European Conference on Parallel Processing*, 2011.

[27] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proc. of the 7th Conf. on Virtual Execution Environments*, 2011.

[28] Y. Tamura, K. Sato, S. Kihara, and S. Moriai. Kemari: Virtual machine synchronization for fault tolerance. In *USENIX Annual Technical Conference (Poster)*, 2008.

[29] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proc. of the 1st Symposium on Cloud Computing*, 2010.

[30] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, 2002.

[31] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. Cloudnet: dynamic pooling of cloud resources by live WAN migration of virtual machines. In *Proc. of the 7th Conf. on Virtual Execution Environments*, 2011.

[32] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proc. of the International Conf. on Cluster Computing*, 2010.