Future Generation Computer Systems 29 (2013) 1085–1095

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Utilizing memory content similarity for improving the performance of highly available virtual machines

Balazs Gerofi *, Zoltan Vass, Yutaka Ishikawa

Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, Japan

ARTICLE INFO

Article history:
Received 2 February 2012
Received in revised form
14 May 2012
Accepted 11 June 2012
Available online 4 July 2012

Keywords:
Virtualization
Hypervisor
Checkpoint
Recovery
Fault tolerance

ABSTRACT

Checkpoint-recovery based Virtual Machine (VM) replication is an emerging approach towards accommodating VM installations with high availability. However, it comes with the price of significant performance degradation of the application executed in the VM due to the large amount of state that needs to be synchronized between the primary and the backup machines. It is therefore critical to find new ways for attaining good performance, and at the same time, maintaining fault tolerant execution. In this paper, we present a novel approach to improve the performance of services deployed over replicated virtual machines by exploiting data similarity within the VM's memory image to reduce the network traffic during synchronization. For identifying similar memory areas, we propose a *bit density based hash function*, upon which, we build a content addressable hash table. We present *a quantitative analysis on the degree of similarity* we found in various workloads, and introduce *a lightweight compression method,* which, compared to existing replication techniques, *reduces network traffic by up to* 80% and yields a *performance improvement over* 90% for certain latency sensitive applications.

1. Introduction

With the recent increase in cloud computing's prevalence, the number of online services deployed over virtualized infrastructures has experienced a tremendous growth. At the same time, however, the latest hardware trend of growing number of components in current computing systems renders hardware failures common place rather than the exception [1]. Replication at the Virtual Machine Monitor (VMM) layer is an attractive technique to ensure fault tolerance in such environments, primarily, because it provides seamless failover for the entire software stack executed inside the Virtual Machine (VM), regardless the application or the underlying operating system. One particular approach, checkpoint-recovery based VM replication, has gained a lot of attention recently [2–5].

Checkpoint-recovery based replication of virtual machines is attained by capturing the entire execution state of the running VM at relatively high frequency in order to propagate changes to the backup machine almost instantly. Essentially, it keeps the backup machine nearly up-to-date with the latest execution state of the primary machine so that the backup can take over the execution in case the primary fails [2].

Between checkpoints the VM executes in log-dirty mode, i.e., write accessed pages are recorded so that when the snapshot is taken only pages that were modified in the most recent execution phase need to be transferred. One phase of dirty logging and transferring the corresponding changes is often called a *replication epoch* [2,4,5]. In order to reduce the overhead of transferring dirty pages, replication data can be transferred asynchronously, overlapping the VM's execution in the subsequent epoch.

However, any fault tolerant system needs to ensure that the state from which an output message is sent will be recovered despite any future failure, which is commonly referred to as the *output commit* problem [6]. As a consequence of such a requirement, during the execution phase of each epoch, output of the running VM needs to be held back, i.e., disk I/O and network traffic have to be buffered and can be released only after the backup machine acknowledged the corresponding update.

With workloads that touch memory rapidly, the time required to propagate changes at the end of an epoch may exceed the replication period itself, leading to substantial overhead, and causing significant performance degradation (over 2 X slowdown) to the application, even if dirty content is transferred asynchronously [2]. This anomaly becomes rather severe in case the application is latency sensitive, such as several online services [7].

Various recent papers have explored the phenomena of content redundancy. *VMware ESX Server* [8] and *Satori* [9] eliminates identical pages shared among and within VMs' memory content for better physical memory utilization. Koller and Rangaswami [10]

* Corresponding author.
E-mail addresses: bgerofi@il.is.s.u-tokyo-ac.jp (B. Gerofi), zvass@il.is.s.u-tokyo-ac.jp (Z. Vass), ishikawa@is.s.u-tokyo-ac.jp (Y. Ishikawa).

0167-739X/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2012.06.008

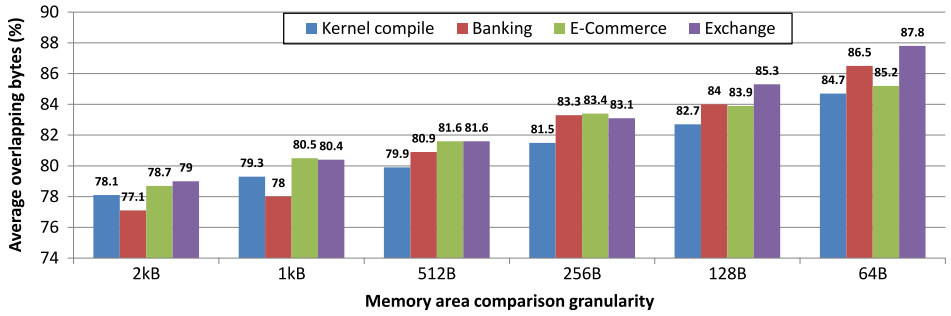


Fig. 1. Average overlapping bytes of the non-zero dirty memory areas and their most similar matches in the non-dirty memory region or in the *recently dirtied pages* cache according to the comparison unit size in various workloads.

proposed I/O deduplication, a mechanism that utilizes content redundancy for improving I/O performance. All these studies suggest that there is a significant degree of content self-similarity in today's complex workloads.

In this paper we investigate how to utilize such similarities to improve the efficiency of virtual machine replication, and thus, the performance of services being executed inside the replicated VM. We make the following contributions:

- A *quantitative analysis of several workloads regarding the degree of self-similarity* within their memory content is presented.
- Taking advantage of such redundancy we propose a *lightweight compression method* which, instead of transferring the actual dirty pages, finds similar areas in the memory content corresponding to the VM's previous replication epoch and transfers a compressed difference. Having the backup VM waiting a replication epoch behind the primary enables us to simply apply the difference and bring its memory content to the latest replication state.
- For identifying similar memory areas, we propose a *bit density based hash function*, upon which, we build a content addressable hash table.
- Finally, *we eliminate the VM downtime at the data transfer phase* of each replication epoch *by having the virtual machine executed in copy-on-write* mode until the compression is finished.

Our mechanism reduces the amount of data transferred during replication by up to 80% and improves the performance of certain latency sensitive applications over 90% as opposed to the regular asynchronous replication.

We begin with characterizing various workloads in terms of memory content self-similarity in Section 2. Section 3 describes the design of our proposed replication method and Section 4 provides details on the implementation. Experimental evaluation is given in Section 5. Section 6 surveys related work, and finally, Section 7 presents future plans and concludes the paper.

2. Background and content similarity analysis

In this section we present the motivation and rationale behind this study. We start with describing each workload we investigated, which is then followed by a quantitative analysis regarding the degree of content self-similarity they exhibit.

2.1. Workloads

Reliable execution may be required by a diverse set of applications, such as long lasting computations or mission critical online services. Inspired by previous studies in the domain of high-availability [2,4] we chose four different workloads. Three of them were deployed on Ubuntu Linux and one on Windows Server 2003.

- *Linux kernel compile* is an elaborate workload, stressing mainly CPU and memory, but doing a fair amount of disk I/O as well. We compile the *bzImage* target of the vanilla Linux kernel version 2.6.31 with default configuration.
- *SPECweb 2005 banking* emulates an Internet personal banking web-site, where clients are accessing their accounts, making transactions, etc. Requests are transmitted over SSL throughout the whole benchmark [7].
- *SPECweb 2005 e-commerce* resembles the workload characteristics of an online store. Customers are browsing, customizing and purchasing products. Both SSL and plain HTTP are utilized [7].
- *Exchange load generator* is a benchmark utility that stresses Microsoft's Exchange Server. It simulates a scenario where multiple users read and send messages, browse their calendars, request meetings, and so forth [11].

2.2. Analysis

As mentioned earlier, checkpoint-recovery based replication of virtual machines is delivered by capturing snapshots of the running VM at relatively high frequency so that changes can be reflected to the backup machine almost instantly.

Between subsequent snapshots, write accessed memory pages are logged to narrow the image necessary to transfer at the end of each epoch. Previous studies suggested that the memory content of nowadays' complex workloads may exhibit a rather high degree of self-similarity. We were curious to see to what extent the content of dirty pages could be expressed with help of the content from the previous epoch. In order to retain access to most of the previous epoch's memory content, we maintain a small cache of the *recently dirtied pages* (RDP). The cache consists of 5120 pages and it is updated at the end of each epoch replacing pages in a least recently used (LRU) fashion.

We analyzed the similarity attributes of each workload by performing an extensive search over the non-dirty memory region and the RDP cache and identified the best match for every non-zero dirty area. Such comparison were carried out in every 100 ms for a 10 min execution of each workload. To speed up the search, we utilized our content addressable hash table, searching through all entries in the corresponding hash buckets. For detailed information of the content addressable hash table refer to Section 3.

We collected statistics of the average percentage of overlapping bytes between each non-zero dirty area and its best match in the content hash. We were also wondering how the unit size of the search may affect such property and used 2, 1 kB, 512, 256, 128 and 64 B as comparison granularity.

Fig. 1 indicates the results obtained for each workload. Looking at the numbers of kernel compilation, the figure shows that the degree of similarity scales from 78% up to almost 85% with shrinking the comparison granularity from 2 kB gradually down to 64 bytes. *SPECweb*2005's Banking workload shows the

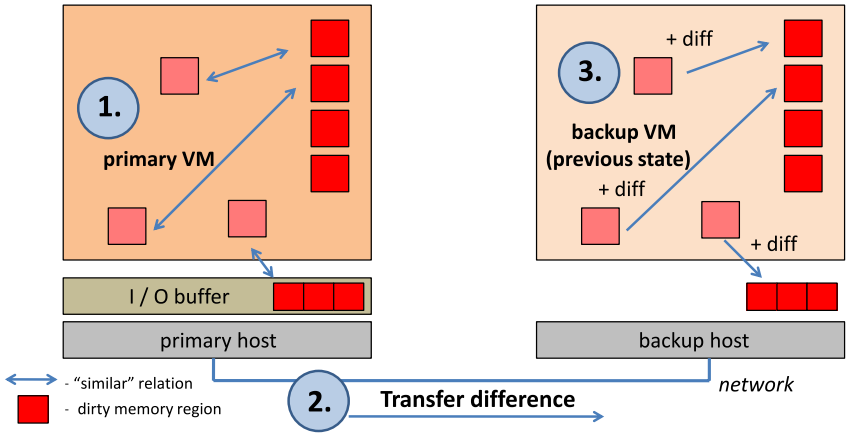


Fig. 2. Utilizing content similarity in VM replication, high-level design. *Three main steps of updating the backup VM during a replication epoch:* (1) *For each dirty memory area identify a similar region in the VM's memory content corresponding to the previous replication epoch and generate a compressed difference.* (2) *Transfer the update to the backup machine.* (3) *Apply the difference to the backup VM's memory.*

Table 1
Distribution of memory areas (unit size 512 bytes).

Workload	Dirty pages	Zero areas (%)	Identical match (%)	RDP cache (%)
Kernel	3842	9.2	15	41
Banking	2153	9.6	36	44
E-commerce	1797	10.2	34	48
Exchange	2925	9.8	38	48

steepest increase, from 77% up to nearly 87% gaining almost 10% improvement as the comparison granularity is set finer. The E-commerce workload shows fewer improvements with the comparison granularity change and grows approximately 6% from the initial 79%. Among all workloads, however, the Windows Server based Exchange Server proved to have the highest degree of self-similarity, scaling from 79% up to almost 88% when reaching 64 bytes comparison granularity.

As seen, for all workloads there is an apparent increase in the degree of similarity with the decreasing unit size of the memory comparisons. While a finer grained comparison granularity clearly leads to a higher compression ratio, it also introduces additional overhead to the compression mechanism itself. Smaller unit size implies an increase in the number of data structures representing the memory (see Section 3), as well as in the number of hash table lookups during the compression.

We opted to use 512 bytes as area unit size in our experiments, because it is fine grained enough to give reasonable compression and the number of data structures is also acceptable. Table 1 provides further insights regarding the distribution of different memory area types with this unit size. For each workload it indicates the average number of dirty pages in 100 ms, the ratio of areas that contain only zeros, the ratio of areas that have fully identical matches either in the non-dirty region or in the RDP cache, and the ratio of areas that have their most similar pair in the RDP cache.

One of the key observations is that the kernel compilation workload is significantly heavier than the rest in terms of memory usage. It also has a substantially lower degree of similarity as the ratio of memory areas that have identical matches. Another observation is the ratio of zero areas, which appears to be close across all workloads. Note that the regular asynchronous replication can also omit transferring zero pages, but since it operates on page size granularity it is unable to eliminate zero areas that are smaller in length than a page. It is also worth pointing out, that only less than half of the dirty areas have their most similar matches in the RDP cache, i.e., more than half of them are rather similar to the non-dirty memory region of the VM.

3. System design

In this section we give an overview of the system architecture, describe how similar memory areas are identified and detail some of the design choices we faced during the development of our replication strategy.

The main idea of the algorithm is depicted by Fig. 2. Three major steps are executed during every epoch of the replication. After the VM is suspended and the dirty page map is updated, instead of transferring dirty pages directly to the backup machine, we first attempt to find similar memory areas both in the VM's non-dirty memory region and in the cache of most recent dirty pages. For each area we make an XOR based diff against the best match and compress it with a lightweight method, explained below. Second, the compressed data along with the addresses of the reference areas are transferred to the backup machine asynchronously. Finally, the backup VM applies the uncompressed diffs to the referenced memory areas and updates the dirty regions.

3.1. Finding similar memory areas

There have been several solutions proposed in the literature for finding similar elements in high-dimensional spaces, which may be also considered for application in the context of finding similar memory regions. A short survey regarding some of the possible techniques is presented in Section 6.

Hashing is one of the prevalent approaches, although choosing the right hash function in this case is rather complicated, due to the desire for having similar elements mapped to the same hash value. Notice, that the ultimate purpose of finding a similar memory area is to generate an XOR based difference that holds zero values on most of its offsets. While many of the existing hashing solutions consider the actual bit sequences of the input vectors, an XOR based diff may result in many zeros already if it is just ensured that the compared vectors have high bit density on the same offsets.

Driven by this idea, we propose a simple hashing solution that is built upon a pop-count based projection. Fig. 3 depicts the hashing mechanism. Regardless the size of the memory area concerned, it is divided into 32 sections where each section corresponds to one bit in the hash (resulting in a 4 bytes long hash value). On each section the number of bits set is calculated and compared against a threshold, which determines whether the corresponding bit in the hash value is set or not. Our current implementation utilizes an empirical value of having 80% of the number of bits set as threshold. Each bit in such projection indicates the density of bits set in the corresponding section of the original memory

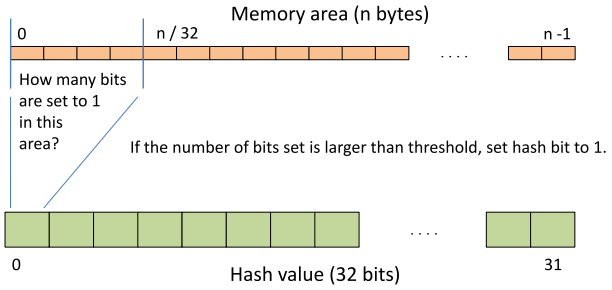


Fig. 3. Density based hash function.

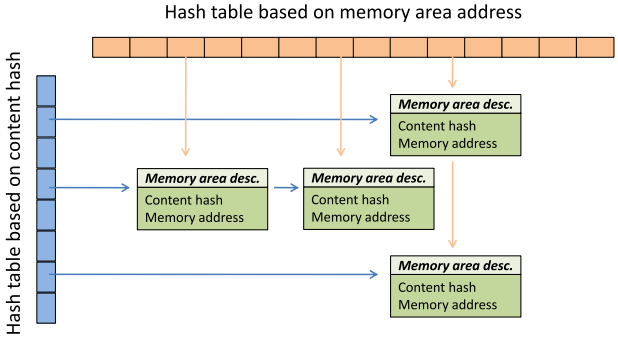


Fig. 4. Content and address based hash table.

area, mapping similar memory areas to the same hash value. Since the introduction of SSE4.2 instruction set extension, pop-count is natively supported by the x86 architecture, which makes it computationally very efficient. Pop-count also proved to be sufficient enough throughout our experiments, and while a more rigorous comparison would be desired among existing hashing techniques, such a study is outside the scope of this paper.

3.2. Content addressable hash table

Using the pop-count based hash function we built a hash table that can be addressed through two dimensions: address and content. Fig. 4 demonstrates the hash table's arrangement. Each non-zero memory area is represented by a memory descriptor, that holds the memory address of the area and the corresponding content hash value. All descriptors are inserted through both dimensions, where descriptors residing in the same hash bucket of the content hash table refer to memory areas that have likely similar content. Our current implementation uses 18 bits wide hash tables both in address and content dimensions.

Once the dirty memory areas are identified at the end of a replication epoch, the corresponding memory descriptors are removed from the hash table. Memory descriptors that belong to the RDP cache are distinguished and they hold a pointer to the cached data instead of the actual VM memory. Otherwise, they play the same role with other entries that describe non-dirty memory.

3.3. Compression

The purpose of finding similar areas in the memory content corresponding to the VM's previous replication epoch is to decrease the network traffic required to update the backup machine. Once we obtain the corresponding memory area, the most similar the content hash table could identify, a XOR based compression is then performed.

One possible light-weight approach to compressing XOR vectors, suggested by related work [12], is run-length encoding (RLE). RLE tends to be very efficient on samples where long

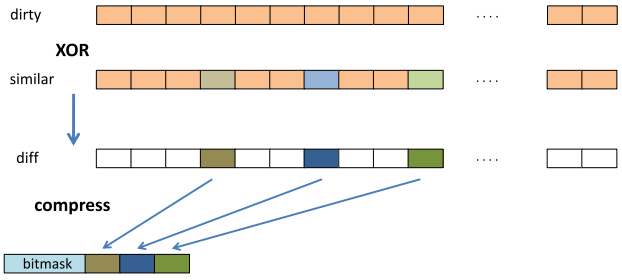


Fig. 5. XOR compression of dirty memory areas.

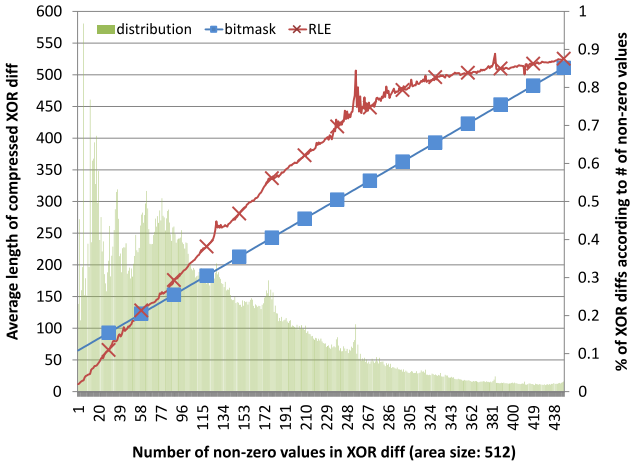


Fig. 6. Comparison of bitmask based and RLE compressions. *Average compressed length of each method (left axis), and the distribution of XOR vectors (right axis) according to the number of non-zero elements they contain. Samples are from the kernel compilation workload.*

sequences of the same value are frequent, however it gives poor results when this isn't the case. Fig. 5 demonstrates an alternative, bitmask based mechanism, where the main steps are as follows. First, the dirty memory area and its similar pair is compared and a XOR based diff is generated. Since we expect that many offsets of the resulting vector are filled with zeros, we generate a bitmask to describe which offsets hold non-zero values. The bitmask and the actual non-zero values are then simply concatenated.

We collected XOR vectors from the execution of the kernel compilation workload in order to make a comparison between the above mentioned two compression methods. For each vector we recorded the number of non-zero elements and the length of the compressed vectors for both techniques. At the end, the average length as the function of non-zero elements and the method used was computed.

Fig. 6 compares the efficacy of the two approaches. As seen, when the number of non-zero elements in the XOR vector is small, RLE outperforms the bitmask based approach. However, if the number of non-zero elements is larger than approximately 10% of the unit size, the bitmask compression yields better results. Fig. 6 also shows the distribution of XOR vectors (on the right axis) with respect to the number of non-zero elements they contain. According to the observed distribution, XOR vectors on which the bitmask based compression outperforms RLE comprise over 61% of the samples.

Our implementation takes advantage of both approaches. It starts performing a bitmask based compression, but if the compressed size falls below the observed threshold it dynamically switches to run length encoding.

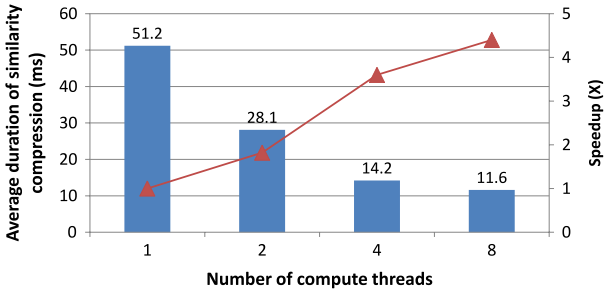


Fig. 7. Average time spent on similarity compression according to the number of compute threads utilized. *Kernel compilation workload, replication frequency is* 100 ms, *average number of dirty pages is approximately* 3800. *(Host machine is a* 4 *cores Intel Xeon with* 2 *hyperthreads each core.).*

3.4. Copy-on-write

The merit of asynchronous data transfer during replication is the reduction of VM downtime, i.e., the time while the virtual machine is suspended when replication data is transferred to the backup machine. Instead of waiting until the transfer completes, the regular asynchronous replication first copies all dirty pages into a local buffer, resumes the VM immediately and then transfers the data to the backup host [2]. This way, data transfer overlaps the next epoch's execution phase.

Unfortunately, in our case this solution is not directly applicable, because we need the entire memory content of the VM from the given epoch so that comparison against the non-dirty memory region can be also performed consistently.

In order to prevent extending VM downtime, we modified the virtual machine monitor so that it does copy-on-write (COW) apart dirty page tracking when it is desired. COW is enabled only during the compression to ensure that the similarity scan accesses memory content which corresponds to the previous epoch. Clearly, COW demands a certain amount of extra memory so that the previous value of write accessed pages can be retained. However, because COW is only enabled for a short period of time (see Section 3.5) during each replication epoch, we observed a modest demand for additional memory, up to 20 MB in the worst case.

3.5. Multi-core optimization

As the latest trend in computer hardware is the ever increasing number of CPU cores, an obvious approach to achieving higher performance is exploiting concurrency. Computing new hash values for dirty memory areas, looking up similar regions in the content addressable hash table, as well as performing the actual XOR compression can be parallelized efficiently.

In order to avoid synchronization as much as possible our similarity compression mechanism exploits two techniques, the content hash table is implemented in a lock-free fashion so that compute threads can access it simultaneously; moreover, during compression thread-local data structures are utilized to collect changes that need to be applied to the global hash table at the end of each replication epoch.

Some basic synchronization is still inevitable, such as notifying compute threads when data is available and making sure all compute threads have finished before applying changes to the global hash table.

Fig. 7 compares the efficiency of running the compression on different number of CPU cores. As seen, for up to 4 threads we achieve near linear speedup, but the improvement between 4 and 8 cores doesn't follow this trend. We believe this is the effect of the low-level arrangement of the Xeon CPU we used for our experiments, which provides the illusion of 8 CPU cores via hyper-threading, sharing the same amount of cache with the 4 CPU cores configuration.

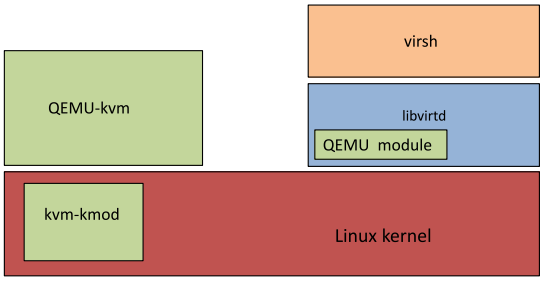


Fig. 8. The Linux KVM architecture.

4. Implementation

This section discusses technical issues regarding our implementation.

4.1. KVM

We chose the Linux Kernel Virtual Machine (KVM) [13] as the platform of this study. KVM takes advantage of the hardware virtualization extensions so that it achieves nearly the same performance with the underlying physical machine.

Fig. 8 depicts the high-level KVM architecture. The most important components are the *kvm* kernel module and *qemu-kvm*, a KVM tailored version of QEMU. On top of these, *libvirtd* is an often used facility for managing virtual machines, for which *virsh* provides a command line interface. A major advantage of the KVM architecture is the full availability of user-space tools in the QEMU process, such as threading, libraries and so on. We make changes to all components in order to integrate replication support into KVM.

4.2. Copy-on-write

On the lowest level, we extended the KVM kernel module to perform copy-on-write when it's requested by *qemu-kvm*. Copy-on-write is a well applied technique in operating systems, particularly for enforcing private access to an otherwise shared memory area among separate address spaces. However, in our case, COW is not as straightforward as it is with regular processes, because the compression threads and the running VM actually share the same address space. When a page is written and COWed, the VM still needs to access the most recent content, while the compression threads should see the previous epoch's value. In order to meet both requirements we remap the old content of the page to another address and maintain a translation table, which is queried by the compression threads to find out whether or not a page has been COWed. Note, that COW pages are recycled in each epoch once the compression is finished.

4.3. Compression and I/O buffering

Most of the replication logic, including the similarity based compression is implemented in *qemu-kvm*, leveraging a great part of the live migration code.

For disk I/O and network buffering we modified the virtio drivers of *qemu-kvm*. The disk I/O buffer behaves also as a hash table that operates on sector granularity so that read requests referring to sectors which are already buffered can be accessed consistently. As for network buffering we maintain an extra packet queue that captures outgoing packets during the execution phase of a replication epoch. Once the backup machine acknowledges the update both disk and network buffers are committed.

4.4. Transactional updates

Another particular issue worth mentioning is the transactional nature of updating the backup machine. When replication data are

sent to the backup host, *qemu-kvm* cannot just read and apply the changes directly, because a failure during the update would leave the backup machine in an inconsistent state.

This implies that at the end of each replication epoch the backup machine needs to collect the updates first and then apply all changes together in a transactional fashion, only if all data were received successfully. Unfortunately, the network protocol of *qemu-kvm*'s live migration code doesn't support this by default.

For this reason we extended the *QEMUFile* object with a buffer and a flag that indicates that the file is in accumulating mode. The primary machine toggles this flag on the file corresponding to the backup connection and all subsequent writes are first buffered. We record the number of bytes to be transmitted and inform the backup machine in advance regarding the length of the update. It can then read the whole stream, store it in a buffer and toggle the backup file's flag to indicate that subsequent read operations issued by *qemu-kvm* should access the buffer instead of receiving data from the network.

5. Evaluation

5.1. Experimental framework

Our experiments were conducted on three server nodes, each machine equipped with a 4 cores Intel Xeon 2.2 GHz CPU (2 hyperthreads per core), 3 GB of RAM, a 250 GB SATA harddrive and two Broadcom NetXtreme II BCM5716 Gigabit Ethernet network interfaces. One of the physical network cards were bridged to the virtual machine and used for application traffic and the other was dedicated to the replication protocol. The host machines run Ubuntu server 9.10 on Linux kernel 2.6.31 and we used qemu-kvm 0.12.3 with kvm-kmod 2.6.31.6b as the basis of our implementation. For both the Linux and Windows Server 2003 virtual machines we used the KVM virtio disk and network drivers. We do not present performance results on the native host machine, because in virtualized environments direct access to the underlying machines is normally not available. However, we had Intel's hardware MMU virtualization support, i.e. Extended Page Tables (EPT) enabled in all experiments. Each VM had one virtual CPU and 1 GB of RAM allocated with memory ballooning support disabled.

We used a separate desktop computer running Windows XP for executing the Exchange Load Generator and another Ubuntu desktop for the SPECweb client scripts. Furthermore, for the SPECweb workload, one of our server nodes was utilized to host a VM for SPECweb's backend server deployment.

5.2. Kernel compilation

Our first target is the kernel compilation workload. In this experiment we compile the *bzImage* target of Linux kernel version 2.6.31 using the default configuration. We repeated each experiment three times for each VM setup and report the average wall-clock time measured.

Fig. 9 illustrates the results with respect to four different VM setups. Native VM measures the execution time on the original, unmodified KVM virtual machine. Regular asynchronous replication indicates the method proposed in the *Remus* paper [2], where the primary machine accumulates all replication data first, resumes the VM immediately, and lets the actual transfer overlap with the next epoch's speculative execution. The similarity compression mode enables our proposed compression method, but the VM is suspended until the end of compression in each epoch. Finally, similarity compression with copy-on-write indicates the case where data stream is similarity compressed and

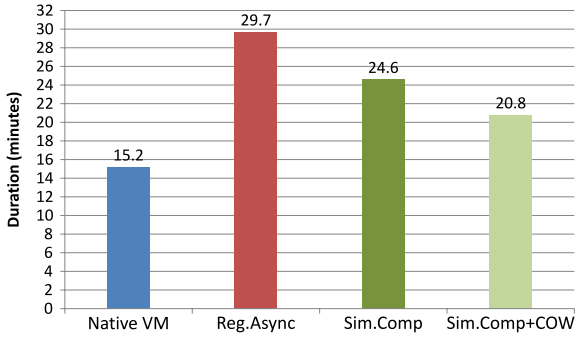


Fig. 9. Duration of kernel compilation on native VM and with various replication strategies.

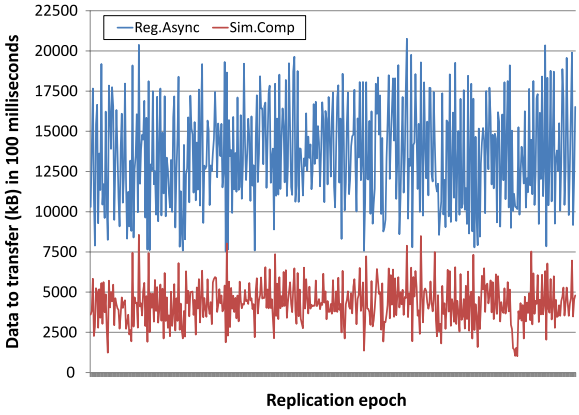


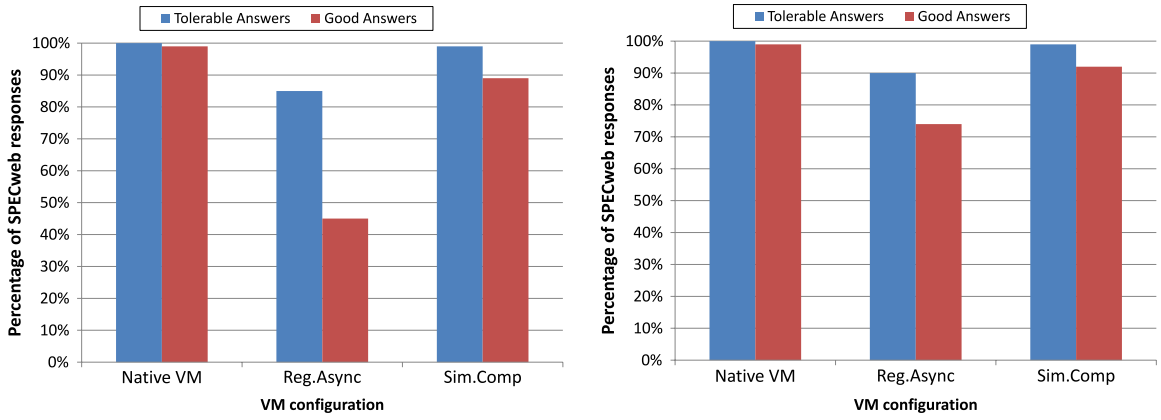
Fig. 10. Compression ratio for kernel compilation.

the VM downtime is eliminated by having the virtual machine executed in COW mode for the duration of the compression.

As seen, in terms of execution time, the overhead introduced by the regular asynchronous replication leads to significant performance degradation, showing nearly two times worse efficiency compared to the native VM's configuration. The 15 min required by the native VM increased up to approximately 29 min in this case. Similarity compression alleviates this overhead by reducing the execution time near to 24 min, yielding a 20% performance improvement.

We have previously discussed (in Section 3.5) how multi-core execution is utilized for better performance. Fig. 7 revealed the time spent on data compression in each 100 ms epoch. We emphasize again, that without copy-on-write, throughout the similarity compression the VM needs to be suspended so that compute threads see a consistent view of the VM's memory image. In case of using four CPU cores the introduced downtime is around 16 ms in each 100 ms replication epoch. Fig. 9 indicates the improvement when such service interrupt is eliminated by copy-on-write. As seen, with COW enabled the kernel compilation completes in less than 21 min, reaching 70% of the native VM performance, which corresponds to over 40% performance improvement compared to the regular asynchronous replication.

Fig. 10 illustrates the compression ratio achieved by the similarity compression. It shows the number of bytes transferred in a couple of hundred subsequent epochs. We also logged the compression ratio and computed the number bytes the regular asynchronous replication would have transferred. One of the key observations here is that the Gigabit Ethernet network doesn't offer wide enough bandwidth so that the regular replication could catch up with our mechanism, this is in fact the reason of the attained speedup. As seen, the average compression ratio settles around 30%.



(a) SPECweb Banking workload.

(b) SPECweb E-Commerce workload.

Fig. 11. Average percentage of good and tolerable answers in SPECweb workloads.

5.3. SPECweb

The next two applications we investigate is SPECweb's Banking and E-Commerce workloads. The SPECweb configuration requires at least three machines for running the experiments [7]. One of the server hosts is the actual SPECweb application server, which is accompanied by a backend machine. These were deployed in two VMs residing on two separate physical machines. Besides these, a desktop machine was utilized for running the SPECweb client side scripts.

We replicate only the main SPECweb application server, for which another physical machine was utilized to serve as backup host. We ran three different setups, first we tuned the SPECweb configuration so that 99% of the responses are categorized as "good" when executed on the native VM. Both the regular asynchronous replication and the similarity compressed method were then measured with the same configuration and we compare the average percentage of "good" and "tolerable" responses reported by the SPECweb client script. The replication period is set to 50 ms in these experiments, because SPECweb is more sensitive to network latency. Note, that changing the replication period doesn't affect the fairness of the comparison itself, because the same epoch length is used in all setups.

Fig. 11(a) compares the results obtained for the Banking workload. SPECweb reports two separate values for each experiment, the ratio of "good" and "tolerable" answers. As mentioned above, the experiment is configured so that SPECweb evaluates 99% of the responses from the native VM as "good".

A closer look at the results reveals that again, the regular asynchronous replication introduces severe performance degradation to SPECweb. The ratio of results marked as "good" dropped below 45% in this case, although 85% were still evaluated as "tolerable".

When similarity compression is performed, note that similarity compression means COW enabled here, the ratio of "good" results increased to 88%, yielding a 95% improvement compared to the regular asynchronous replication. As for the ratio of "tolerable" answers, in case of similarity compression, 98% of the results are "tolerable", which is nearly as good as the performance of the native VM.

Fig. 11(b) illustrates the same comparison for the E-Commerce workload. As previously, the configuration was tuned to achieve 99% of the replies marked as "good" on the native VM. The performance degradation imposed by regular asynchronous replication is not as heavy as in case of the Banking workload and it only drops to 90% and 74%, for "tolerable" and for "good", respectively.

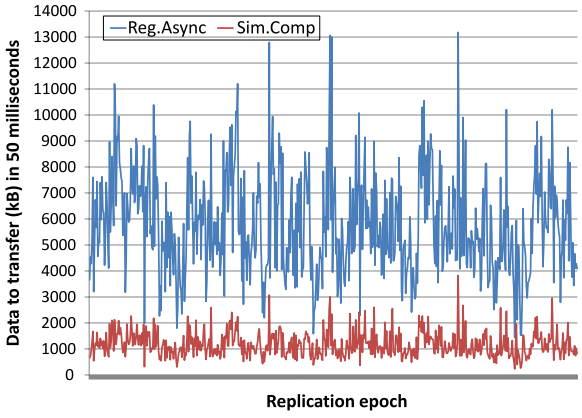


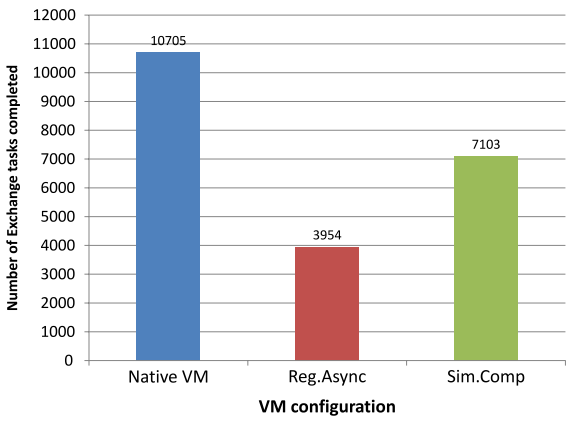
Fig. 12. Compression ratio for SPECweb banking.

Previously, Table 1 in Section 2.2 showed a comparison of the average number of dirtied pages for all workloads. As seen, the E-Commerce workload is substantially lighter in terms of memory usage compared to Banking. We believe this implies the lower degradation in performance due to the fact that the overhead's main factor is the amount of data to be transferred. Our proposed method mitigates this overhead achieving 99% "tolerable" and 92% "good" responses, which, respectively, corresponds to a 10% and 24% improvement over the regular asynchronous replication.

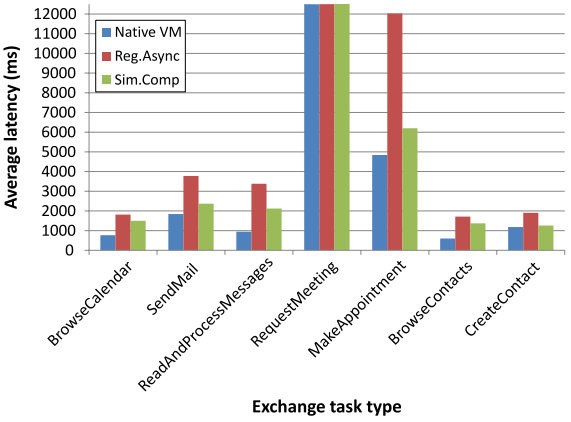
Fig. 12 depicts the obtained compression ratio for the Banking workload. We followed the same procedure as in case of kernel compilation, recording the number of bytes transferred, the ratio achieved and computing the amount of bytes the regular replication would have had to transfer. Again, one of the key observations is the fact that the Ethernet bandwidth would be insufficient to keep up with the pace of the produced dirty data in case of the regular replication. As shown, our proposed mechanism attains an average of 20% compression ratio for the Banking workload.

5.4. Exchange server

The results presented so far were all obtained on Ubuntu Linux. In this section we evaluate the performance of our mechanism when applied to Microsoft Exchange Server 2007 deployed over Windows Server 2003. The Microsoft Exchange Server is a messaging system that provides e-mailing, calendars,



(a) Average tasks completed under stress-mode in 10 min.



(b) Task latency details. The values for RequestMeeting are truncated for clarity. They are 24 583, 66 763, and 32 583, respectively.

Fig. 13. Microsoft exchange load generator results.

attachments, contacts, etc. We used the Microsoft Exchange Load Generator utility [11], which simulates the server workload that is generated by interaction of multiple users. This benchmark tool is mainly used for the purpose of server sizing and deployment plan validation, but it also provides a facility for stress-testing server installations.

We ran the Exchange Load Generator on a separate Windows XP client machine. Only the server host was replicated in our experiments and we used the same three setups as we did with SPECweb, native VM, regular asynchronous replication, and similarity compression with COW enabled. The replication period was calibrated to 50 ms. The Exchange Load Generator was executed three times for 10 min under stress-test mode and we report the average number of tasks finished for each setup.

Fig. 13(a) demonstrates the average number of Exchange tasks completed in 10 min with respect to the different VM setups. Compared to the native VM's over 10 000 tasks, the achieved performance in case of regular asynchronous replication degrades to as low as 3954. Our proposed mechanism alleviates this degradation finishing approximately 7100 tasks in 10 min, which in turn is a 79% performance improvement.

Exchange Load Generator provides detailed information on certain attributes of the executed tasks. We computed the average latency of the most frequent tasks during the experiments. Fig. 13(b) depicts the actual numbers obtained. As seen, when compared to the native VM's performance, the general tendency is that responses generated during the regular replication have significantly higher latency. On the other hand, latencies for the similarity compression method reside in the interval of the native and the regular replication's, yielding significant improvements in some cases, such as the *SendMail*, *MakeAppointment*, or the *CreateContact* tasks. A closer look at the numbers reveals that similarity compression, for these particular tasks, attains substantially closer efficiency to the native VM than to the regular asynchronous replication.

Another key observation is that all tasks have higher latency than 50 ms even in case of the native VM, which implies that the main factor of the regular replication overhead is the inability of propagating changes to the backup VM fast enough. We have verified this by running a couple of experiments with 100 ms replication epoch and observed very similar results.

Fig. 14 illustrates the compression ratio similarity compression achieves against the regular data transfer. We used the same method to generate this figure as for the previous workloads and we draw similar conclusion. The main source of the attained performance improvement is the fact that the compressed stream

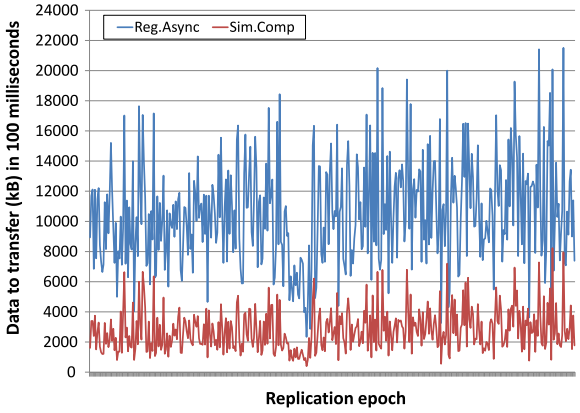


Fig. 14. Compression ratio for Windows Exchange Server.

can propagate more changes than the regular replication. Our proposed solution maintains an average 28% compression ratio for the Exchange Server workload.

5.5. CPU and memory consumption

Previously, we showed how replication of various workloads benefits from our proposed compression technique. Clearly, performing such activity in every 50/100 ms requires additional resources on the primary VM's host machine. In this section we turn our attention to evaluate the price of the compression in terms of CPU and memory consumption.

There are several sources for additional memory demand when performing VM replication. Disk I/O buffering and network packet capturing both allocate extra chunks of memory, which are inevitable for ensuring consistency. We logged two attributes of the block cache, the frequency how often disk I/O was involved in replication data, and the number of sectors dirtied when disk I/O occurred. The replication epoch was set to 100 ms and the block sector size was 512 bytes.

Table 2 illustrates the obtained results for all workloads. The first column shows the percentage of replication epochs when block I/O was involved, the second and third columns show the maximum and the average size of the I/O buffer per epoch during the experiment. As seen, block I/O varies significantly according to the workload considered. Across all workloads the average amount of memory consumed as block cache scales from 160 to 800 kB, reaching 4.7 MB in the worst case.

Table 2
Replication resource consumption (area unit size 512 bytes).

Workload	I/O freq. (%)	Max. size of disk I/O buff	Avg. size of disk I/O buff (kB)	Max. size of content hash (MB)	Avg. size of content hash (MB)	Avg. CPU overhead (%)
Kernel compilation	11	4.7 MB	800	19	13	125
SPECweb Banking	6	850 kB	280	24	23	134
SPECweb E-commerce	6	352 kB	160	22	21	132
Microsoft Exchange	70	4.6 MB	265	52	50	148

In Section 3, Fig. 4 showed the arrangement of the content hash, which holds a descriptor structure for each non-zero area in the VM's memory content. Table 2 also shows the average and the maximum size of the content hash table. The table reveals that the memory allocated for the content descriptors scales from 18 to 50 MB, and from 13 to 50 MB, as average, and maximum, respectively. Moreover, we maintain an LRU cache of 5120 pages that allocates another 20 MBs. Overall the memory consumption of the content hash scaled from 38 up to 70 MB in our experiments, which we think is acceptable for a 1 GB virtual machine.

We used the *atop* [14] utility for logging CPU consumption of *qemu-kvm* during the experiments both with and without replication enabled. To assess the replication overhead in terms of CPU consumption, we computed the average CPU usage in both cases and report the difference between the replication enabled and the native VM cases. The last column of Table 2 indicates the obtained results, note that 100% corresponds to one CPU core here. As seen, the additional CPU power required is about 130%, which equals to a little above utilizing an extra CPU core. We believe such overhead is reasonable, especially with the ever growing core number of recent CPU architectures.

6. Related work

Similar elements in high dimensional spaces. Finding similar regions to dirty memory areas is essentially a similarity search in a high dimensional space. Previous works have yielded several approaches for finding similar elements in high-dimensional spaces. Solutions, such as K-clustering [15] or R-trees [16] could provide very accurate results, however, due to their computational complexity they cannot be applied in the scenario of VM replication.

Another prevalent approach is hashing, although, one-way hash functions such as MD5 or SHA-1 are not feasible, because by definition, they map elements that are close in the input space to different hash values. To overcome this problem, local-sensitive hash functions [17] have been proposed in the literature, but none of them turned out to be efficient enough in our case. In the context of network filesystems, LBFS [18] suggested combining SHA-1 with Rabin fingerprints in order to locate identical areas on different offsets within files. However, we are aiming at finding similar areas, not only identical ones. A recent work, *Difference Engine* [19] suggested using one-way hash functions, but instead of hashing the whole memory area, only portions of the page are hashed at random offsets and the obtained values are then combined. In this paper we proposed a pop-count based hash function, which is computationally less expensive and extracts information based on bit density of the whole memory areas rather than the bit sequence of random offsets.

Memory content similarity and deduplication. Content similarity in memory has been also investigated in the literature. *VMware ESX Server* [8] and *Satori* [9] introduced techniques for better utilizing the physical memory in virtualized systems by eliminating duplicate memory content across and within virtual machine instances. Identical pages are detected and deduplicated into one single read-only page. Copy-on-write is then utilized to ensure consistency in case the page is modified. *Difference Engine* [19] aims at the same goal, but it leverages sub-page level page sharing and memory compression to further improve memory efficiency. Koller and Rangaswami [10] proposed I/O deduplication, a mechanism that utilizes content similarity for improving I/O performance by eliminating I/O operations and reducing the mechanical delays during I/O operations. Of these, *Difference Engine* and I/O deduplication have apparent similarities to our work because they both utilize a content based hash table to find similar content in the memory. However, our hashing mechanism and sharing granularity is different than those proposed in the above papers.

Virtual machine migration. Checkpoint-recovery based fault tolerance captures snapshots of the running VM at high frequency, often leveraging the live migration support of the underlying Virtual Machine Monitor (VMM). Thus, VM live migration is closely related to checkpoint-recovery based replication. Solutions, such as *Xen* [20], *KVM* [13], and *VMware's VMotion* [21] all provide the capability of live migrating VM instances. Pre-copy is the dominant approach to live VM migration [20,21]. It initially transfers all memory pages then tracks and transfers dirty pages in subsequent iterations. When the amount of data transferred becomes small or the maximum number of iteration reached, the VM is suspended and finally, the remaining dirty pages and the VCPU context is moved to the destination machine. VM replication, on the other hand, leaves the VM running in pre-copy mode at all times so that dirty pages are logged and the entire execution state can be reflected to the backup node at the end of each replication epoch [2,3]. In parallel with our work a recent paper proposed a technique similar to ours, where content similarity is exploited in the context of VM live migration [12]. However, their proposed technique for identifying similar memory pages is different than ours, furthermore, VM replication involves various different technical issues, which distinguishes our work from this study.

Virtual machine replication. Bressoud and Schneider [22] introduced first the idea of hypervisor-based fault tolerance by executing the primary and the backup VMs in lockstep mode, i.e., logging all input and non-deterministic events of the primary machine and having them deterministically replayed on the backup node in case of failure. While Bressoud and Schneider demonstrated this technique only for the HP PA-RISC processors, VMware's recent work implements the same approach for x86 architecture [23]. Deterministic-replay, however, imposes strict restrictions on the underlying architecture and its adaption to multi-core CPU environment is cumbersome, because it requires determining and reproducing the exact order in which CPU cores access the shared memory. *Flight Data Recorder* [24] and the work of Dunlap et al. [25] enable deterministic replay for SMP environments, but it is unclear what degree of concurrency they can handle without significant performance degradation.

Checkpoint-recovery based solutions such as *Remus* [2] and *Paratus* [3] overcome the problem of multi-core execution by capturing the entire executions state of the VM and transferring it to the backup machine. Although most of the data transfer can be overlapped with speculative execution, transferring updates to the backup machine at very high frequency still comes with great performance overhead. *Kemari* [26] follows a similar approach to *Remus*, but instead of buffering output during speculative

execution, it updates the backup machine each time before the VM omits an outside visible event.

Improving the performance of checkpoint-recovery based VM replication has become an active research area recently. Lu and cker Chiueh [4] proposed fine-grained dirty region identification to reduce the amount of data transferred during each replication epoch, while Zhu et al. [5] improved the performance of log-dirty execution mode by reducing read- and predicting write-page faults. In this paper we also focus on reducing the amount of data transferred during each replication epoch, although we utilize content similarity instead of fine-grained dirty region identification.

7. Conclusions and future work

Checkpoint-recovery based virtual machine replication is attractive, it provides high availability for the entire software stack executed in the VM, and it runs on commodity hardware. However, it comes with great overhead due to the large amount of state that needs to be synchronized frequently between the primary and the backup hosts.

In this paper we have first presented a quantitative analysis of various workloads in terms of content similarity in their memory image. For all workloads we investigated, we have found that the degree of overlapping bytes between dirty data and the previous epoch's memory content is high, about 80% when the comparison granularity is 512 bytes.

We have proposed a novel compression method to alleviate the replication overhead by exploiting such similarities. Our mechanism uses a content addressable hash table to identify similar memory areas to the dirty content in memory region corresponding to the previous replication epoch and expresses the changes with a compressed XOR difference.

The proposed compression method can reduce network traffic by up to 80%, thus, propagating changes faster and yielding a performance improvement of over 90% for certain latency sensitive applications when compared to the regular asynchronous replication. We also showed that the compression comes with modest resource consumption, it requires up to 70 MB extra memory when it is applied to a 1 GB VM and utilizes a little more than an extra CPU core for computation.

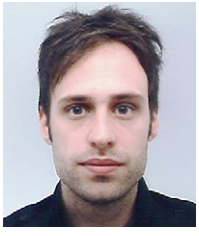
One of the merits of checkpoint-recovery based replication is its inherent capability of handling symmetric multiprocessing (i.e. multiprocessor) virtual machines. Checkpoints cover the entire execution state of the VM, including any non-determinism that arises due to concurrent access of shared memory in case of SMP configurations. Since wide-spread usage of SMP virtual machines is highly anticipated [27], in the future we intend to evaluate the scalability of our approach over VMs with multiple virtual CPUs and larger amount of memory.

Acknowledgments

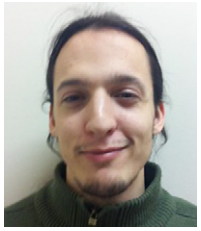
This work has been supported by the CREST project of the Japan Science and Technology Agency (JST).

References

[1] K.V. Vishwanath, N. Nagappan, Characterizing cloud computing hardware reliability, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC'10, ACM, New York, NY, USA, 2010, pp. 193–204.
[2] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, A. Warfield, Remus: high availability via asynchronous virtual machine replication, in: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI'08, 2008, pp. 161–174.
[3] Y. Du, H. Yu, Paratus: instantaneous failover via virtual machine replication, in: Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing, GCC'09, IEEE Computer Society, 2009, pp. 307–312.
[4] M. Lu, Tzi-cker Chiueh, Fast memory state synchronization for virtualization-based fault tolerance, in: Dependable Systems Networks, 2009. DSN'09. IEEE/IFIP International Conference on, 2009, pp. 534 –543.
[5] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, X. Li, Improving the performance of hypervisor-based fault tolerance, in: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 2010, pp. 1 –10.
[6] R.E. Strom, D.F. Bacon, S.A. Yemini, Volatile logging in *n*-fault-tolerant distributed systems, in: Fault-Tolerant Computing, Eighteenth International Symposium on, Jun 1988, pp. 44 –49.
[7] R. Hariharan, N. Sun, Workload characterization of SPECweb2005, 2006. http://www.spec.org/workshops/2006/papers/02_Workload_char_SPECweb2005_Final.pdf.
[8] C.A. Waldspurger, Memory resource management in VMware ESX server, SIGOPS Oper. Syst. Rev. 36 (2002) 181–194.
[9] G. Miłós, D.G. Murray, S. Hand, M.A. Fetterman, Satori: enlightened page sharing, in: Proceedings of the 2009 Conference on USENIX Annual technical conference, USENIX'09, USENIX Association, 2009.
[10] R. Koller, R. Rangaswami, I/O deduplication: utilizing content similarity to improve I/O performance, Trans. Storage 6 (2010) 13:1–13:26.
[11] Microsoft Corporation. Microsoft Exchange Load Generator. 2007. http://www.msexchange.org/articles/Microsoft-Exchange-Load-Generator.html.
[12] X. Zhang, Z. Huo, J. Ma, D. Meng, Exploiting data deduplication to accelerate live virtual machine migration, in: Cluster Computing (CLUSTER), 2010 IEEE International Conference on, 2010, pp. 88 –96.
[13] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: the Linux virtual machine monitor, in: Ottawa Linux Symposium, July 2007, pp. 225–230.
[14] ATOP System & Process Monitor. 2010. http://www.atcomputing.nl/Tools/atop.
[15] J.A. Hartigan, Clustering Algorithms, ninety ninth ed., John Wiley & Sons, Inc., New York, NY, USA, 1975.
[16] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD'84, ACM, 1984, pp. 47–57.
[17] Q. Lv, W. Josephson, Z. Wang, M. Charikar, K. Li, Multi-probe LSH: efficient indexing for high-dimensional similarity search, in: Proceedings of the 33rd international conference on Very large data bases, VLDB'07, 2007, pp. 950–961.
[18] A. Muthitacharoen, B. Chen, D. Mazières, A low-bandwidth network file system, in: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP'01, ACM, New York, NY, USA, 2001, pp. 174–187.
[19] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, A. Vahdat, Difference engine: harnessing memory redundancy in virtual machines, Commun. ACM 53 (2010) 85–93.
[20] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, Live migration of virtual machines, in: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, NSDI'05, USENIX Association, Berkeley, CA, USA, 2005, pp. 273–286.
[21] M. Nelson, B.H. Lim, G. Hutchins, Fast transparent migration for virtual machines, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC'05, USENIX Association, Berkeley, CA, USA, 2005, p. 25.
[22] T. Bressoud, F.B. Schneider, Hypervisor-based fault tolerance, in: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, SOSP'95, ACM, New York, NY, USA, 1995, pp. 1–11.
[23] D.J. Scales, M. Nelson, G. Venkitachalam, The design of a practical system for fault-tolerant virtual machines, SIGOPS Oper. Syst. Rev. 44 (2010) 30–39.
[24] M. Xu, R. Bodik, M.D. Hill, A "flight data recorder" for enabling full-system multiprocessor deterministic replay, in: Proceedings of the 30th Annual International Symposium on Computer Architecture, ISCA'03, ACM, 2003, pp. 122–135.
[25] G.W. Dunlap, D.G. Lucchetti, M.A. Fetterman, P.M. Chen, Execution replay of multiprocessor virtual machines, in: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE'08, 2008, pp. 121–130.
[26] Y. Tamura, Kemari: virtual machine synchronization for fault tolerance using DomT, 2008. Technical report, NTT Cyber Space Labs.
[27] R. McDougall, J. Anderson, Virtualization performance: perspectives and challenges ahead, SIGOPS Oper. Syst. Rev. 44 (2010) 40–56.



Balazs Gerofi is a Ph.D. candidate of the Computer Science Department at the University of Tokyo as of April 1, 2009. He received his M.Sc. degree in Computer Science (cum laude) from the Vrije Universiteit Amsterdam in October, 2006. His research focuses on operating systems, distributed computing, virtualization and dependable systems.



Zoltan Vass is a M.Sc. candidate of the Computer Science Department at the University of Tokyo as of April 1, 2011. He received his B.Sc. degree of Computer Engineering from the University of Pannonia at Veszprem, Hungary in January, 2009. His research focuses on high performance computing, communication, and distributed computing.

Yutaka Ishikawa is a professor of the University of Tokyo, Japan. Ishikawa received the B.S., M.S., and Ph.D. degrees in Electrical Engineering from Keio University. From 1987 to 2001, he was a member of AIST (former Electrotechnical Laboratory), METI. From 1993 to 2001, he was the chief of Parallel and Distributed System Software Laboratory at Real World Computing Partnership. His interests include next generation supercomputers, cluster technologies, dependable parallel and distributed systems.