Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines

Petter Svärd
Umeå University
petters@cs.umu.se

Benoit Hudzia
SAP Research CEC Belfast
benoit.hudzia@sap.com

Johan Tordsson
Umeå University
tordsson@cs.umu.se

Erik Elmroth
Umeå University
elmroth@cs.umu.se

Abstract

Despite the widespread support for live migration of Virtual Machines (VMs) in current hypervisors, these have significant shortcomings when it comes to migration of certain types of VMs. More specifically, with existing algorithms, there is a high risk of service interruption when migrating VMs with high workloads and/or over low-bandwidth networks. In these cases, VM memory pages are dirtied faster than they can be transferred over the network, which leads to extended migration downtime. In this contribution, we study the application of delta compression during the transfer of memory pages in order to increase migration throughput and thus reduce downtime. The delta compression live migration algorithm is implemented as a modification to the KVM hypervisor. Its performance is evaluated by migrating VMs running different type of workloads and the evaluation demonstrates a significant decrease in migration downtime in all test cases. In a benchmark scenario the downtime is reduced by a factor of 100. In another scenario a streaming video server is live migrated with no perceivable downtime to the clients while the picture is frozen for eight seconds using standard approaches. Finally, in an enterprise application scenario, the delta compression algorithm successfully live migrates a very large system that fails after migration using the standard algorithm. Finally, we discuss some general effects of delta compression on live migration and analyze when it is beneficial to use this technique.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Measurement, Performance

Keywords Virtualization, Live migration, Compression, Performance evaluation

VEE '11 Mar 09-11 2011, Newport Beach, CA, USA

1. Introduction

Virtualization is an abstraction of computer hardware as a software implementation of a machine, a virtual machine (VM) that can execute programs just as a physical machine. By using a hypervisor, several VMs can share the same underlying hardware with neglectable overhead [2]. Virtualization thus allows multiple operating system instances to run concurrently on a single physical machine. This enables more efficient usage of hardware resources by use of techniques such as server consolidation. Virtualization also makes it easier to achieve isolation of services since each service can run on a dedicated virtual server. Other benefits of virtualization are dynamic allocation of resources (i.e., resizing of VMs) to meet deviations in workload as well as simplified provisioning, monitoring, and administration of servers.

In addition, it is possible to move (migrate) a VM from one host to another without shutting it down, increasing flexibility in VM provisioning. Live migration techniques for VMs [6] focus on capturing and transferring the run time, in memory, state of a VM over a network. Live migration enables administrators to reconfigure virtual machine hardware and move running virtual machines from one host to another while maintaining near continuous service availability. This ability to migrate applications encapsulated within VMs without perceivable downtime is becoming increasingly important in order to provide continuous operation and availability of services in the face of outages, whether planned or unplanned. Live migration also increases efficiency as it is possible to manage varying demands in an organization with fewer physical servers and less human effort by enabling on-the-fly server consolidation.

So far, live migration research has focused on transferring the run-time in-memory state of VMs with relative limited memory size in Local Area Networks (LANs). The usability of current live migration techniques is limited when large VMs or VMs with high memory loads such as VMs running large enterprise applications have to be migrated, or when migration is performed over slow networks. The reason for this shortcoming is that the hypervisor is unable to transfer the VM memory in the same rate as it is dirtied by the VM. This limitation can easily neutralize the advantages of live migration.

This work extends on an idea by Hacking et al. [7], who propose a live migration algorithm where the VM in-memory state is compressed during transfer. In this contribution, we modify the KVM hypervisor by implementing a delta compression live migration algorithm where changes to memory pages are transferred in-

stead of the full page contents. We evaluate the performance of this approach by analyzing the effects of compression on migration downtime and total migration time. The evaluation is based on experiments with migration of VMs running three different workloads, a synthetic benchmark and a transcoding video server over 100 Mbit/s network (Fast Ethernet), and an enterprise application over a 1000 Mbit/s network (Gigabit Ethernet). The results demonstrate significantly reduced migration downtimes in all cases. We also discuss some general principles regarding the effects of delta compression on live migration performance and analyze when the use of such techniques is beneficial.

2. Background and related work

In order to migrate a VM, its state has to be transferred. A VM's state consists of its memory contents and local file system. However, the transfer of the local file system can be avoided by keeping it on a network file system storage volume accessible to both the source and the destination. To migrate a VM, it is first suspended, its state is then transferred, and the VM is finally resumed on its new host. While the VM is suspended it is unreachable and service is thus interrupted.

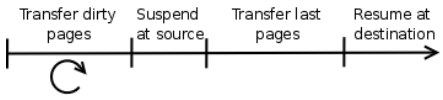

Figure 1. Overview of a typical live migration process.

Live migration aims to solve the problem of migrating a VM without service interruption by keeping the VM running on the source while transferring the state in the background to the destination, an approach first demonstrated by Clark et al. [6]. Although various versions of this algorithm exist, including usage of push and pull techniques for page transfers, on-demand paging, etc., the typical live migration algorithm is as follows. First, all pages are marked as dirty. Then, a number of iterations are performed. In each iteration, the memory pages dirtied in the host during the previous iteration are retransferred. Notably, all memory pages are transferred in the first iteration. Once the number of dirty pages in the source is below a certain threshold, typically set based on the network bandwidth and the desired maximum downtime, the VM is suspended at the source and the last few pages are transferred. Finally, the VM is resumed on the destination. This typical live migration process is outlined in Figure 1. The rationale behind this scheme is that as most of the state already is transferred when the VM is suspended, the downtime before the VM is resumed at the destination can be significantly shortened. If the suspension time is short enough, network connections can be kept alive and service is thus not interrupted. Live migration does not significantly degrade the performance experienced by users (the perceived downtime) of services running in the VM, Furthermore, the migration process is completely transparent as services running in the migrated VM do not need to be migration-aware in any way.

Most virtual machine hypervisors, such as Xen [5], KVM [11], and VMWare [20] support live migration [6, 21] that involve extremely short downtimes ranging from tens of milliseconds to a second. Usually, these techniques assume that migration is performed over a Local Area Network (LAN). However live Wide Area Network (WAN) migration has been demonstrated by several researchers [4, 8, 16, 19], achieved by use of novel techniques based on, e.g., IP tunneling, Mobile IP, and light path reservation.

Bradford et al. [4] propose a solution to live migration of VMs over WANs that uses pre-copying and write throttling. The latter is a dynamic VM rate limiting technique where the amount of hardware resources allocated to the migration is increased at the expense of the performance of the VM. Bradford et al. use a combination of dynamic DNS and IP tunneling to ensure that network connections in the VMs are kept alive during migration. In a contribution by Harney et al. [8], Mobile IPv6 is used to ensure continuous network connectivity without the use of virtual networks. This approach is evaluated in scenarios with and without shared storage. Sapuntzakis et al. [18] present several techniques to improve VM migration, including the use of memory ballooning where a VM can dynamically change its memory usage by evicting unused memory during runtime to make its memory state more compressible. Other techniques used by Sapuntzakis et al. include demand paging of VM disks and content-based block sharing across disk state. Zhao et al. analyze how performance degradation varies with various migration strategies [23]. So far, most of the experiments concerning live migration has focused on small to medium sized VMs typically running web server workloads. Notably, Anedda et al. propose a method for migrating virtual clusters for data-driven HPC applications using delta compression techniques [1]. Their work is however restricted to non-live migration.

3. The problem of Live Migration

While a VM is running, its memory is dirtied at a specific rate, the *dirtying rate*. If this rate is higher than the network throughput, the number of dirty pages to re-transfer increases for each iteration of page transfers (see Figure 1). The hypervisor's migration algorithm can thus not complete the dirty page transfer phase of the migration and the VM has to be prematurely suspended in order to be migrated. The *migration downtime* is the time period from when the VM is suspended in the source host until it is resumed and responding to requests at the destination host. If the memory block that is left untransferred when the VM is suspended is large, this downtime can be long.

This drawback of the current live migration approaches can lead to extensive migration downtime when migrating VMs with high workloads, and/or large RAM sizes such as VMs running large enterprise applications, or when migrating over slow networks. Liu et al. show that the total migration time and the migration downtime do not only depend on the amount of VM memory to transfer but also on the workload type [12]. They illustrate that even for a small VM with a mere 156 MB of RAM, a migration downtime of three seconds and a total migration time of ten seconds may be necessary even over a Gigabit Ethernet network.

3.1 Effects of extended migration downtime

Extended migration downtime can lead to several problems, including issues related to service interruption, consistency problems and unpredictable performance.

3.1.1 Service interruption

Most server applications rely on TCP connections and the connectivity of these must be guaranteed during live migration of the VMs in which they are running. To achieve reliable transmission, TCP packages are resent unless acknowledgements are achieved. The amount of time between retries is determined by the Smoothed Round Trip Time (SRTT). The SRTT varies with the network speed and latency as TCP tunes itself to each network connection for retransmissions. Over Fast Ethernet, TCP connections are aborted after around eight seconds while over Gigabit Ethernet, which is recommended for nearly every implementation of live migration, the timeout occurs even faster, after around five seconds. In effect, this means that all TCP connections are dropped if a VM is suspended with more than 500 MB of memory left to transfer over a Gigabit Ethernet network. Considering that many of today's applications use more than 16 GB of RAM and have large workloads, TCP timeouts are not uncommon.

3.1.2 Consistency and transparency problems

A direct consequence of migration downtime is the difficulty to maintain consistency and transparency. Online transaction processing systems such as Enterprise Resource Planning (ERP) software are very sensitive to variations in delay. Such a system relies heavily on precise scheduling capabilities, triggers, timers, and on the underlying database systems to perform various operations efficiently and with desired properties, i.e., that these ensure Atomicity, Consistency, Isolation and Durability (ACID). If the VM is suspended for too long during a critical execution phase of the system, the downtime often results in data corruption and/or a crash due to missed timers, unscheduled or delayed events, clock drift, etc. For the enterprise application used in the evaluation of our work, downtimes as low as half a second are known to result in unrecoverable problems of this kind.

3.1.3 Unpredictability and rigidity

In their current state, live migration operations tend to be unpredictable as to how much time they need in order to complete. This is because it is difficult to assess the number of iterations required in the dirty page retransfer phase of the typical live migration algorithm. This in turn depends on the page dirtying rate that is unique to each application [6] and cannot be determined by the amount of resources allocated to the VM (CPUs, amount of RAM), etc. As a result, the use of current live migration algorithms is infeasible when timely migrations are needed.

3.2 Existing solutions

To shorten migration downtime, current hypervisors use various methods such as dynamic rate-limiting [4, 6] where the migration traffic bandwidth limit is dynamically increased during migration. There is however a risk of networking resource outage for the VM unless a dedicated network card is used for live migration operations since migration traffic tend to have a higher priority. Because of this, dynamic rate-limiting gradually degrades performance of the VM and to a certain extent the performance of any co-located VMs. Another approach is rogue processes stunning, where processes that write too much in memory are stunned or freezed. Rouge process stunning can be highly disruptive and dangerous with applications that use transactions. Both dynamic rate-limiting and rogue process stunning might significantly degrade the performance of the running VM which is in conflict with the goals of live migration.

These drawbacks are worsened when live migration is done over unreliable or slow connections such as WANs and the Internet. In such environments, it is quite often as efficient and recommended to simply shut-down the VM before migrating it rather than performing the migration live.

4. A Delta Compression approach

Current live migration algorithms normally starts by marking all RAM memory pages as dirty on the source host. Then, a typical algorithm iteratively transfers pages that have been dirtied since the last round of transfers. The longer time an iteration takes, the more memory is dirtied during the process and has to be re-transferred. When the estimated remaining transfer time is below a certain threshold, the VM is suspended to stop further memory writes and allow the remaining pages to be transferred. The remaining transfer time is calculated as $t_r = m_d/b$, where m_d is the amount of dirty RAM remaining and b is the available bandwidth. The main bottleneck in this algorithm is the transfer of memory pages over the network, as even Gigabit Ethernet is several orders of magnitude slower than RAM or disk access. If the VM dirtying rate is higher than the migration throughput, the migration downtime can be long and service interruption may occur. In order to improve the performance and shorten the migration downtime, either the dirtying rate has to be reduced or the network throughput increased.

As discussed earlier, reducing the dirtying rate by stunning or freezing processes leads to performance degradation and possibly also service interruption. Increasing network throughput thus seems like a more promising approach.

4.1 Run Length Encoding Delta Compression

By compressing the memory pages before transfer, the amount of data to transfer is reduced and the migration throughput is thus increased. This allows the live migration algorithm to better keep up with the VM's dirtying rate. It is important to note that as the type of data in the VM's memory varies depending on which workload the VM runs, the compression algorithm must be able to efficiently compress any type of data. In many applications, the same set of memory pages are written to over and over again, i.e. a specific part of the VM's RAM is constantly dirtied which means that delta compression is a promising candidate to compress the memory pages.

Delta compression is a method for storing data in the form of changes between versions instead of the full data sets. The technique is relatively old and its origins comes from the problem of finding the minimum number of edit operations necessary to transform one string into another [22]. One early application of delta compression was to manage revision control systems in order to save large amounts of storage by storing changes to files rather than the files themselves. Another well known use of binary delta compression is by Microsoft to reduce the download size of software update packages [14]. Delta compression is already used to improve VM operation but its usage is mainly focused on compressing I/O buffers for VMs or storage [10].

The VM memory pages that needs to be compressed are in binary form and it is thus easy to compute the delta page by applying XOR on the current and previous version of a page. To compose the new page at the destination, this process is simply reversed to produce a copy of the source's current version from the previous version and the delta page. The delta page is the same size as the original memory page and as our goal is to reduce the amount of transferred data during migration, the delta page must be compressed. The data must be compressed efficiently enough to significantly reduce transfer time and the compression algorithm should consume a minimum of CPU resources in order not to slow down the migration process or hurt the VM's performance. Fortunately, in many cases when a page is dirtied, only a couple of words are changed. This means that the delta pages can be efficiently compressed by binary Run Length Encoding (RLE) [15] which is a well-known, fast, and efficient compression algorithm.

Another approach is to use domain-specific compression algorithms and select the most suitable one according to the nature of the workload on the VM to be migrated. However, even if the memory contents of a VM running for example a streaming video server would probably compress better using video compression techniques this would come at the expense of VM performance. The live migration algorithm runs in parallel with the VM and each CPU cycle added affects the performance of the VM as well as the live migration and most domain-specific compression techniques tend to be CPU intensive. In addition, many domain-specific compression techniques are destructive and, as a bit-for-bit replica of the memory contents is needed, such techniques cannot be used. Finally, a choice would have to be made as to which compression technique to use for a specific application, making the live migration process less transparent. We therefore believe that it is best to use a lean and efficient general compression technique and, in our

evaluation, we opted to use a a straight-forward word-wise RLE algorithm.

4.1.1 Caching

In order to compute a delta page, the previous version of the page is needed. On the destination side this is not a problem since the source's previous version of a page is the destination's current version. However, on the source side the memory contents are continuously overwritten and a copy of the previously sent version of the page must be kept. It is not feasible to store a copy of every page in memory since this would double the VMs memory usage and render the caching approach unusable for CPU intensive VMs with large memory, which are the ones most likely to benefit from this technique. Hence, only a subset of the VM's memory pages should be stored which means that a caching scheme must be used.

As discussed eariler, it is often the case that a set of memory pages is being constantly dirtied, this set is called the VM's *hot pages set*. For large VMs or VMs running heavy workloads, the hot pages set can be large and the page cache needs to be large enough to hold a significant part of the hot pages or cache churn might occur, i.e., the contents of the cache is constantly replaced and migration performance is thus decreased. In addition, the cache lookup time should remain constant when the cache size increases as not to decrease migration performance when using larger cache sizes. Finally, the algorithm should consume a minimum of CPU resources, both for cache hits and cache misses to not slow down the migration process or disrupt the performance of the VM.

Various caching schemes like LRU and ARC [13] were evaluated, but failed to fulfill these requirements. Finally, a 2-way set associative caching scheme, commonly used in high performance CPU caches [9], was chosen. In this scheme, the cache maps memory addresses to cache buckets, each bucket holding maximum two memory pages, see Figure 2. If a bucket is full, the oldest entry is evicted and replaced with the new entry. This sort of cache is suitable in our case as it is very lean and the time to locate an element is constant even as the cache size grows.

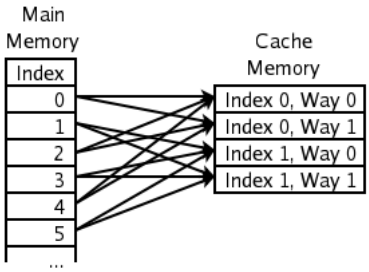

Figure 2. 2-way set associative cache.

4.2 Delta Compression Algorithm Overview

In our evaluation, we use a XOR binary RLE (XBRLE) live migration algorithm in order to increase migration throughput and thus reduce downtime. When transferring a page, the source checks if a previous version of the page exists in the cache. If this is the case, a delta page between the new version and the cached version is created using XOR. The delta page is compressed using XBRLE and a delta compression flag is set in the page header. Finally, the cache is updated and the compressed page is transferred. On the destination side, if the delta compression flag is set for a page, the delta page is decompressed and the new version of the page is reconstructed from the delta page and the destination's previous version of the page using XOR. A schematic overview of the delta compression scheme on the source and destination side respectively can be found in Figures 3 and 4. The algorithm is implemented as modification to the user space code of the qemu-kvm [11] hypervisor.

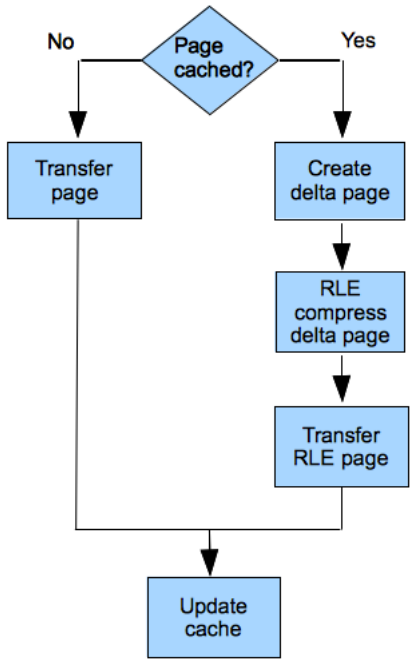

Figure 3. Source side delta compression scheme.

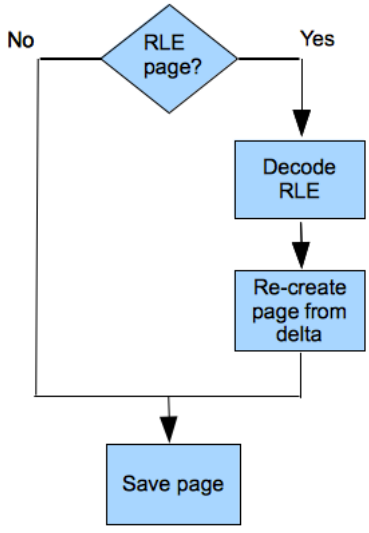

Figure 4. Destination side delta compression scheme.

5. Evaluation

To evaluate the performance of the XBRLE live migration algorithm, a number of VMs were live migrated. In order to test the algorithm under various conditions we investigated three live migration scenarios using VMs running different workloads. The workloads where selected to range from ideal conditions (high dirtying rate, data suitable for delta compression, limited bandwidth) to less ideal cases (data not so suitable for delta compression, high bandwidth).

The algorithm's performance is highly dependent on the cache size. A too small cache will most likely result in cache churn and no performance gain. A large cache size does not hurt the performance but is wasteful with the host's system resources. In the three scenarios, the cache size was chosen to match the size of the hot pages set of the VM. The size of the hot pages set was estimated by studying the memory usage of the running processes.

5.1 Experimental Scenarios

As described in the previous section, three scenarios were considered in the performance evaluation. For the "ideal" scenario, a synthetic benchmark was used. In the second scenario, a real-world application which produces a high dirtying rate was used, namely a VLC server. Finally, a SAP ERP system was chosen as it is a well-known, widely used business application that puts a high load on the system and is sensitive to network timeouts.

The characteristics of the three scenarios are as follows:

Fast Ethernet (benchmark): 1 GB RAM, 1 vcpu VM running two instances of the LMbench memory write benchmark of 256 MB each.

Fast Ethernet (HD video transcoding): 1 GB RAM, 1 vcpu VM running a VLC transcoding video server streaming 720p video.

Gigabit Ethernet (ERP system): 8 GB RAM, 4 vcpus VM running a SAP CI ERP system with 100 concurrent users.

In the first two scenarios, the tests were performed on two Intel 2.66 GHz core2quad servers with 16 GB of RAM running Ubuntu 9. The servers were connected to fast Ethernet using separate network adapters for the host and the VM networks. The VM images were stored on a NFS share on the source machine. The size of the 2-way set associative cache was 512MB.

In the last scenario, the test was performed on two Intel 3 GHz 4 x Dual Core Xeon servers with 32 GB of RAM running Ubuntu 10.4. The servers were connected to a Gigabit Ethernet network using a shared network adapter for the host and the VM networks. The VM images were stored on a 16 TB Raid 5, 4 x quad core Xeon NFS server with 16 GB of RAM and a 6 Gbit/s trunked Ethernet card. In this scenario, the size of the set associative cache was 1GB.

Each test was run twice, the first time using the standard qemu-kvm migration algorithm, henceforth referred to as *vanilla*, and a second time using the XBRLE algorithm. The version of qemu-kvm used in the evaluation is 0.11.5. The modified version of qemu-kvm used for the evaluation is built from the qemu-kvm 0.11.5 source with the delta compression modifications described above.

5.2 Experimental Results

In this section, the results from the three test scenarios are presented. The performance evaluation demonstrate that migration downtime, both VM suspension time and perceived downtime, is reduced in all of the studied scenarios when using XBRLE compression. The difference in the first two cases is significant and promises improved live migration of memory-intensive VMs with large working sets and migration over slower networks without service interruption. In the last scenario, the difference in migration downtime is less significant, but large large enough for the migration to succeed in the XBRLE case whereas it failed using the vanilla algorithm.

5.2.1 Fast Ethernet (benchmark) scenario

In the first scenario a synthetic benchmark which gives a high dirtying rate was run. The total migration time, the suspension time, and the migration downtime for the benchmark test is shown in minutes, seconds, and milliseconds in Figure 5. The total migration time was decreased from 65 s to 48 s and the suspension time dropped from 32 s to 0.297 s using the XBRLE algorithm. The migration throughput in MB/s, shown in Figure 7, was increased by around 80%. The ping downtime, i.e., the time when the VM is unresponsive to ICMP ping requests, dropped from 32 s to 1 s. However, as the ping resolution used was 1 s, it is likely that the downtime in the XBRLE case was even shorter.

The synthetic benchmark used in the evaluation, the LMbench [3] memory write benchmark, is part of the LMbench suite of performance benchmarks for UNIX. The *bw_mem_wr* benchmark allocates a specified amount of memory, zeros it, and then coordinates the writing of that memory as a series of four byte integer stores and increments. This benchmark causes a very high dirtying rate within the allocated memory block since the memory contents are constantly overwritten. This type of workload is also known as a *diabolic workload*. As the dirtying rate is higher than the network throughput, the standard migration algorithm cannot keep up and has to suspend the VM with around half of the working set untransferred (256 MB), which causes the extended downtime that can be seen in Figure 5. As the data consists mostly of zeros, it is very suitable for RLE encoding. The hypervisor thus manages to transfer almost all of the working set before suspending the VM in the XBRLE case. To conclude, the results from the benchmark test demonstrate the benefits of the XBRLE live migration algorithm under near ideal conditions.

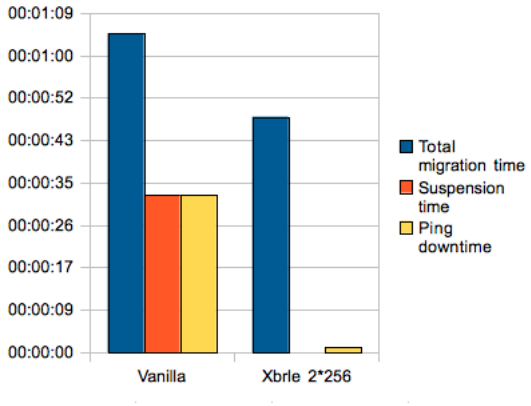

Figure 5. Migration time (LMbench).

5.2.2 Fast Ethernet (HD video transcoding) scenario

In the first real-world application scenario, a VLC transcoding video server was migrated over a LAN. The total migration time, which can be seen in Figure 6, was two seconds longer in the XBRLE case (28 s vs 26 s) but the suspension time was much shorter (0.028 s vs 2 s). The *UDP downtime*, which is the time when no UDP packets are received at the client, was decreased from 8 s to 2 s and the migration throughput was increased from 11.39MB/s to 12.38 MB/s.

Streaming video is an example of a service that is widely available over the Internet and, as in the benchmarking case, this type of application causes a high dirtying rate, especially when, as in our case, the video stream is transcoded. However, the data is not very compressible as the content of the streaming buffer changes constantly and the difference between subsequent video frames can be large. Because of this, many pages could not be compressed as the data stream had changed too much since the last iteration of the dirty pages transfer. This explains that, in the test, the increase in throughput was lower than in the benchmark case. However, the shortened suspension leads to a shorter UDP downtime which means that the gap in the video stream is reduced. This difference is clearly visible in Figure 8 and Figure 9, which show the

amount of UDP packages sent from the video transcoding server to the client during migration of the VM that runs the streaming server. Notably, the spikes in these graphs that appear after migration is caused by the post-migration decrease in I/O bandwidth that results from keeping the VM file system on a NFS share on the source host. Because of the difference in downtime, when migrating using the vanilla algorithm, the video on the client was frozen for eight seconds and choppy for some additional time. When using the XBRLE algorithm, the migration was unnoticeable since the gap in the data stream was short enough to fit in the client side streaming buffer. These results demonstrate that also applications that have a high page dirtying rate and working set data not suitable for RLE compression can be migrated with good results using the XBRLE algorithm.

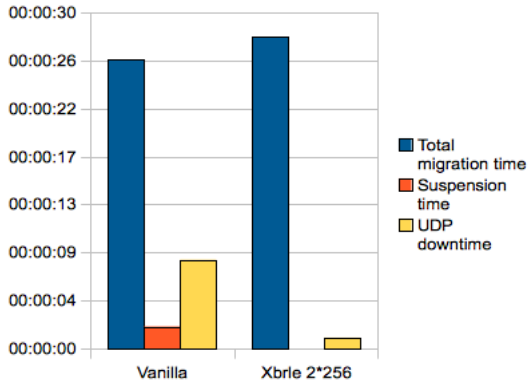

Figure 6. Migration time (HD video).

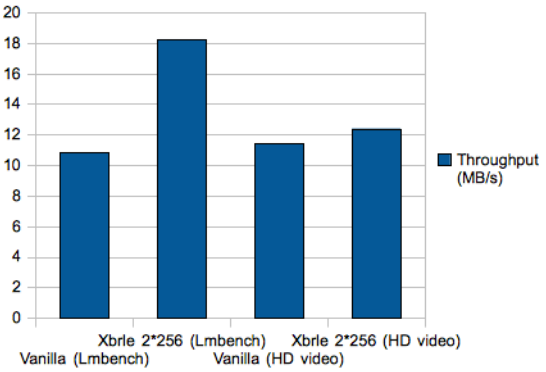

Figure 7. Migration throughput for LMBench and HD video (100 Mbit/s Ethernet).

5.2.3 Gigabit Ethernet (ERP system) scenario

In the final test, a VM running a SAP Central Instance ERP system was migrated over Gigabit Ethernet. With XBRLE, the total migration time was reduced from 235 s to 139 s, the suspension time was reduced from 3 s to 0.2 s, and the ping downtime from 5 s to 1 s, all illustrated in Figure 10.

SAP is one of the most commonly used business intelligence applications. It puts a high load on the CPU and it is considered notoriously hard to migrate these kind of systems live because of their dependency on transactions and time-out sensitive network connections. According to our experience with the SAP ERP system, fatal timing problems can occur already for migration downtimes as low as half a second. In our test, the total migration time was reduced with around 40% but as the system is very complex, the migration time can vary greatly depending on actual load during migration, etc. It is thus hard to draw any definitive conclusions regarding total migration time. More importantly, the migration downtime usually does not vary as much and even if the difference in downtime was not as dramatic as in the two other scenarios, it was still reduced from 3 s to 0.2 s with the XBRLE algorithm. The smaller decrease in downtime compared to the earlier cases can partly be explained by the fact that the network is not as much of a bottleneck as in previous two cases.

Notably, in our test, the vanilla migration did not succeed. In this case, the system was resumed after migration but, as shown in Figure 11, application CPU usage (the dark line) rose to 100% and the system was in an unstable state, probably due to timing errors. Also shown in Figure 11 is the system CPU usage (bright line), of which the KVM hypervisor constitutes the greater part. This value was around 10% during the page transfer iterations, fluctuated substantially as the VM was shut down and resumed, and dropped almost to zero post migration. The third line in Figure 11 shows I/O wait, i.e., processes waiting for network or disk I/O. The I/O wait showed a similar pattern as the system CPU usage.

In contrast, the XBRLE algorithm successfully migrated the ERP system, even at a higher load (70% application CPU) than where the vanilla algorithm failed (20% application CPU). The application CPU usage during migration is shown as the upper dark line in Figure 12. This value fluctuated a bit as the VM was suspended and resumed, but returned to the 70% level after migration. As for system CPU usage (the bright line in Figure 12), this value varied between 5% and 10% during the page transfer iterations. The spikes in this value can be attributed to the rather CPU-intensive compression of delta pages that are part of each iteration. These spikes got higher when more pages were compressed in later iterations, as the reduced amount of VM RAM remaining to transfer resulted in a higher cache hit ratio. The dark line in the lower part of Figure 12 denotes I/O wait, which was low for the XBRLE algorithm, except for a spike during the final shutdown and resume of the VM.

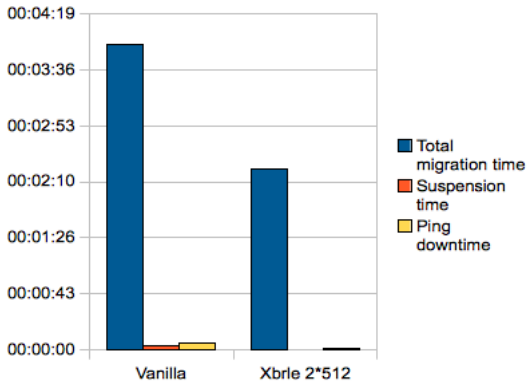

Figure 10. Migration time (ERP).

As the system CPU utilization on the VM was level at around 10% during migration in both cases, we conclude that the XBRLE migration algorithm had about the same impact on VM performance as the vanilla algorithm.

6. Analysis of the General effects of delta compression on Live Migration

In this section, we analyze in which cases delta compression is beneficial by use of a simple model. We also discuss the impact

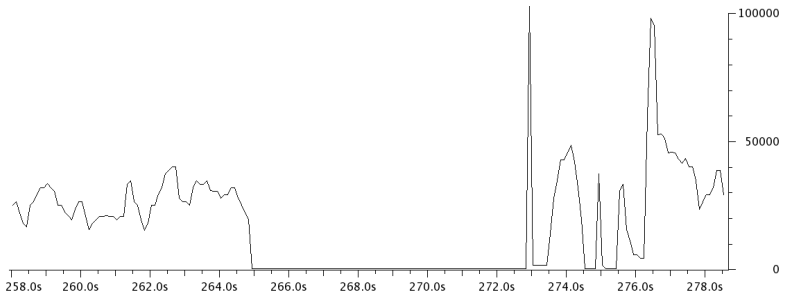

Figure 8. UDP traffic, HD video (vanilla).

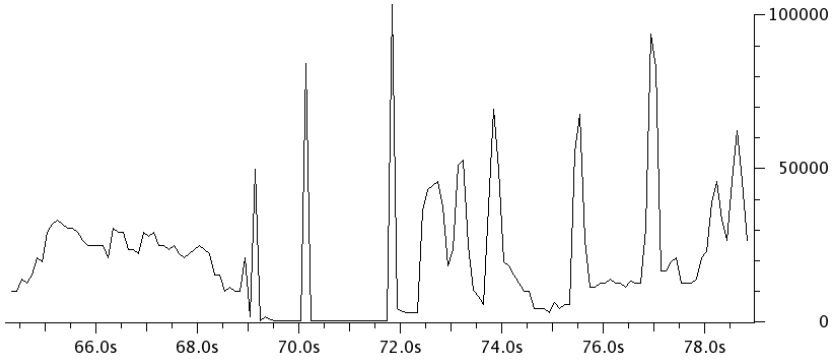

Figure 9. UDP traffic, HD video (XBRLE).

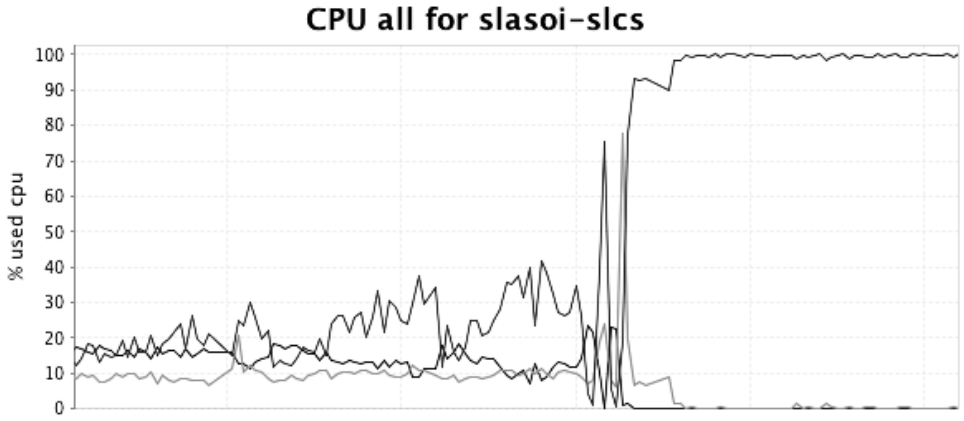

Figure 11. CPU utilization (ERP vanilla).

of cache size on migration performance. When studying the use of delta compression in live migration algorithms there are several parameters that must be considered. As our goal is to reduce migration downtime, the per page migration time is central. In the standard non-compression live migration algorithm, this time can be described as:

$$t_{nc} = t_{tr}, \tag{1}$$

where t_{tr} is the transfer time. When using the delta compression approach evaluated in this paper there are two cases, the first being cache hit:

$$t_h = t_c + t_{cp} + t_{trh} + t_d, \tag{2}$$

where t_c is the cache time, t_{cp} is the compression time, t_{trh} is the cache hit transmission time, and t_d is the decompression time. The second case is cache miss, where:

$$t_m = t_c + t_{tr}. \tag{3}$$

From this follows that the total migration time using the no compression approach can be expressed as:

$$t = \sum_{1}^{n} P t_{nc}, \tag{4}$$

where P is the total number of pages and n is the number of iterations. The total migration time using the delta compression method is:

$$t = \sum_{1}^{n} P_h t_h + \sum_{1}^{n} P_m t_m, \tag{5}$$

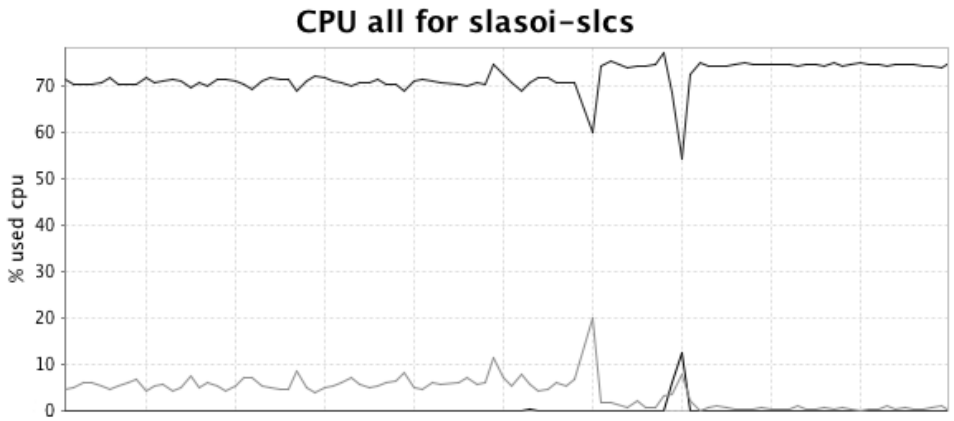

Figure 12. CPU utilization (ERP XBRLE).

where P_h is the number of cache hit pages and P_m is the number of cache miss pages. The total migration time, can also be expressed as:

$$t = n(ht_h + (1-h)t_m), \tag{6}$$

where n is the number of page transfer iterations and h is the cache hit ratio.

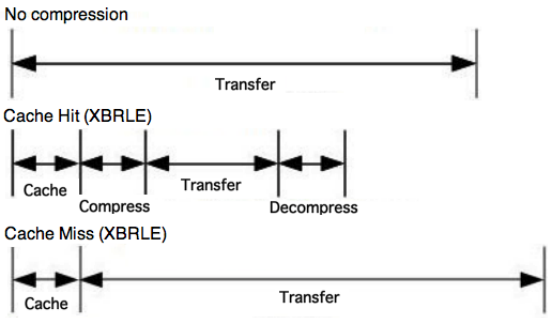

Figure 13. Page transfer times compared.

As illustrated in Figure 13, if the time to cache, compress and decompress a page exceeds the difference in transfer time between the uncompressed and compressed case, the compression case performs worse than the uncompressed. However, using an efficient caching and compression scheme, the shorter transfer time for the cache hit pages makes the total time to process and transfer a page significantly shorter as the time to cache, compress, and decompress the page is short compared to the time gained by transferring less data.

From equations 1 and 3 as well as figure 13 it can be seen that the time it takes to transfer a cache miss page is longer than the time it takes to transfer the same page without compression. From this follows that the cache hit ratio needs to be large enough for the algorithm to be efficient. The potential time gain can be expressed as:

$$t_g = (1-h)(t_m - t_{nc}) - h(t_{nc} - t_h). \tag{7}$$

If the cache hit time, cache miss time, and no compression time can be measured, it seems that the needed cache hit ratio in order for the delta compression algorithm to be faster than the no compression case can be calculated. However, this is an oversimplification. Because the delta compression algorithm is faster at catching up with the VM dirtying memory, the downtime can be reduced even if the total migration time might be longer than using the standard algorithm. This behavior was observed in the streaming video scenario. The reason for this is that during the initial iterations, there is a high number of cache misses but after a couple of iterations, the cache hit ratio increases and the algorithm begins to catch up with the VM. In subsequent iterations, the cache hit ratio increases even more until it reaches a stable value. At this point, the VM dirties memory pages in the source at the same rate as the (compressed) pages can be transferred to the destination. The dirty pages set does thus not decrease in size and the VM has to be suspended for the migration to proceed. In comparison, the vanilla algorithm reaches this point with more memory remaining to transfer since it has no possibility to catch up when the VM dirtying rate is as fast as the algorithm can transfer pages over the network.

In order to achieve this increase in cache hit ratio and the associated reduction in number of pages to transfer in the next iteration, a large enough cache must be used. With a too small cache, the cache hit ratio typically never increases over the iterations and the behavior of the XBRLE migration algorithm becomes very similar to that of the vanilla one. Exactly how large cache is required to achieve the desired performance is difficult to determine analytically as this depends on the page dirtying rate, size of the hot pages set, page compression ratio, network bandwidth, etc. In our evaluation, the cache size was set to match the memory usage of the most active process in the VM which is an estimation of the VM's hot pages set. In the three scenarios investigated, a 512 MB cache was used for migration of the benchmark and HD video VMs, both with 1 GB RAM. For the ERP scenario, a 1 GB cache was used to ensure successful migration of the 8 GB VM running the SAP application.

Figure 14 gives a conceptual illustration of the relationship between the cache hit ratio, amount of dirty pages, number of iterations, and iteration length. In this figure, the height of each block denotes the amount of dirty VM memory remaining to transfer in each iteration. The width of the block is the transfer time. Notable, both the amount of memory remaining and the transfer time are reduced for each iteration, until the steady state is reached. The solid line in Figure 14 shows the cache hit ratio. For the initial iterations, this value is low (zero for the very first iteration), but it then increases rapidly and converges towards a value associated with the steady state where the number of dirty pages is constant between two subsequent iterations.

Another performance metric can be given by the the migration throughput, which can be expressed as:

$$t = P_t * B_p / t_{tr}, \tag{8}$$

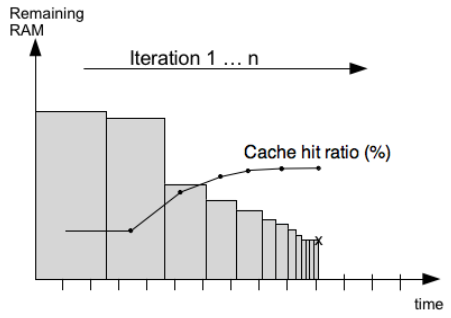

Figure 14. Illustration of how cache hit ratio, number of dirty pages, and transfer time evolve with each iteration of page transfers.

where P_t is the number of transferred pages, B_p is the page size in bytes and t_{tr} is the transfer time. A large increase in throughput compared to the vanilla algorithm is an indication of an efficient compression algorithm and a large number of cache hits. However, it might also be the case that the hot pages set is only a little bit too large for the hypervisor to be able to suspend the VM with an acceptable downtime using the vanilla algorithm. In this case, not that many pages need to be compressed to reach the point where the VM can be suspended and this results in only a small increase in migration throughput. This means that while the throughput sometimes is a good measure of the algorithms performance, this is not always the case.

7. Conclusion and future work

Improved live migration performance is important since it allows for a broader usage of live migration technologies, for example by cloud infrastructure providers in provisioning and relocating of VMs in or between data centers [17]. In this paper, we demonstrate that using delta compression when live migrating large VMs or VMs with heavy load can shorten migration time, migration downtime, and enable live migration with a reduced risk of service interruption. We investigate three test cases migrating VMs that run different workloads, spanning from a synthetic benchmark to an ERP application. For these, we observe a reduced migration downtime of a factor 100 for the benchmark (the ideal scenario), a drop in user-perceived downtime from 8 s to zero for transcoded streaming video, and a reduction of migration downtime from 3 to 0.2 s for an ERP application. Although the improvement was less significant in the ERP scenario, it made, due to the rigid timing requirements of this application, the difference between successful and failed migration.

In summary, our evaluation indicates that using XBRLE compression is particularly beneficial for live migration of:

- VMs running workloads with a highly compressible working set,
- VMs running heavy workloads with large working sets,
- and/or over slow networks (i.e., WANs).

To further improve the performance of live migration, we plan to investigate the use of a better strategy for selection of which memory pages to transfer and in what order to transfer them. The overall idea is to transfer the busiest pages last, as these pages are the ones most likely to be dirtied again. We believe that this method can reduce the number of page re-transfers and thus shorten migration time as well as migration downtime. We also plan to investigate dynamic adjustment of the cache size during migration. As it is very difficult to determine a suitable cache size analytically, this method can be used both to improve migration performance, should the allocated page cache be too small, and reduce the amount of memory allocated to the migration when a too large cache is chosen. Another benefit of dynamic cache size adjustment is that, due to the fact that as the VM working set size changes with each iteration of page copying, the optimal cache size varies accordingly.

Acknowledgments

We acknowledge Stuart Hacking for his contributions to the foundation of this work and Tomas Ögren for installing and configuring the experimental environment. We also thank Eliezer Levy who has been instrumental in setting up this research collaboration. Financial support has been provided by the European Commission's Seventh Framework Programme ([FP7/2001-2013]) under grant agreements no. 215605 (RESERVOIR) and 257115 (OPTIMIS) as well as by UMIT research lab and the Swedish Governments strategic research project eSSENCE.

References

[1] P. Anedda. Suspending, migrating and resuming HPC virtual clusters. *Future Generation Computer Systems*, 26(8):1063 – 1072, 2010.

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

[3] Bitmover. Lmbench, 2010. `http://www.bitmover.com/lmbench/`, visisted June 2010.

[4] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, June 2007.

[5] Citrix Systems. Xen. `http://www.xen.org`, visited October 2010.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286. ACM, May 2005.

[7] S. Hacking and B. Hudzia. Improving the live migration process of large enterprise applications. In *VTDC '09: Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, pages 51–58. ACM, June 2009.

[8] E. Harney, S. Goasguen, J. Martin, M. Murphy, and M. Westall. The efficacy of live virtual machine migrations over the internet. In *VTDC '07: Proceedings of the 3rd international workshop on Virtualization technology in distributed computing*, pages 1–7. ACM, November 2007.

[9] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, 1987.

[10] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High performance virtual machine migration with RDMA over modern interconnects. Technical report, IBM T. J. Watson Research Center, 2007.

[11] Kernel Based Virtual Machine. KVM - kernel-based virtualization machine white paper, 2006. `http://kvm.qumranet.com/kvmwiki`, visited October 2010.

[12] P. Liu, Z. Yang, X. Song, Y. Zhou, H. Chen, and B. Zang. Heterogeneous live migration of virtual machines. Technical report, Parallel Processing Institute, Fudan University, 2009.

[13] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130. USENIX Association, March 2003.

[14] S. Potter. Using binary delta compression (BDC) technology to update Windows XP and Windows Server 2003, October 2005. `http://www.microsoft.com/downloads/details.aspx?FamilyID=4789196c-d60a-497c-ae89-101a3754bad6`, visited October 2010.

[15] D. Pountain. Run-length encoding. *Byte*, 12(6):317–319, 1987.

[16] K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. Live data center migration across WANs: a robust cooperative context aware approach. In *INM '07: Proceedings of the 2007 SIGCOMM workshop on Internet network management*, pages 262–267. ACM, August 2007.

[17] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan. The RESERVOIR model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):1–11, 2009.

[18] C. P. Sapuntzaki, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

[19] F. Travostino. Seamless live migration of virtual machines over the MAN/WAN. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 290. ACM, November 2006.

[20] VMWare. VMWare. `http://www.vmware.com`, visited October 2010.

[21] VMWARE. VMware VMotion: Live migration of virtual machines without service interruption datasheet, 2007. `http://www.vmware.com/files/pdf/VMware-VMotion-DS-EN.pdf`, visited October 2010.

[22] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[23] M. Zhao and R. J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *VTDC '07: Proceedings of the 3rd international workshop on Virtualization technology in distributed computing*, pages 1–8. ACM, June 2007.