High performance live migration through dynamic page transfer reordering and compression

Petter Svärd and Johan Tordsson
Dept. of Computing Science
Umeå University
SE-90187 Umeå, Sweden
Email: {petters, tordsson}@cs.umu.se

Benoit Hudzia
SAP Research CEC Belfast
SAP (UK) Limited
BT37 0QB Newtownabbey
Email: benoit.hudzia@sap.com

Erik Elmroth
Dept. of Computing Science
Umeå University
SE-90187 Umeå, Sweden
Email: elmroth@cs.umu.se

***Abstract*—Although supported by many contemporary Virtual Machine (VM) hypervisors, live migration is impossible for certain applications. When migrating CPU and/or memory intensive VMs two problems occur, extended migration downtime that may cause service interruption or even failure, and prolonged total migration time that is harmful for the overall system performance as significant network resources must be allocated to migration. These problems become more severe for migration over slower networks, such as long distance migration between clouds. We approach this two-fold problem through a combination of techniques. A novel algorithm that dynamically adapts the transfer order of VM memory pages during live migration reduces the risk of re-transfers for frequently dirtied pages. As the amount of transferred data is thereby reduced, the total migration time is shortened. By combining this technique with a compression scheme that increases the migration throughput the migration downtime is also reduced. An evaluation by means of synthetic migration benchmarks shows that our combined approach reduces migration downtime by a factor 10 to 20, shortens total migration time by around 35%, as well as consumes between 26% and 39% less network bandwidth. The feasibility of our approach for real-life applications is demonstrated by migrating a streaming video server 31% faster while transferring 51% less data.**

***Index Terms*—Platform virtualization, Virtual machine monitors, Performance evaluation**

I. Introduction

Current live migration techniques have two main problems when it comes to migrating memory or CPU intensive Virtual Machines (VMs) or migrating VMs over slow networks. Firstly, the sheer amount of data that needs to be transferred is very large. Memory pages which are frequently updated are likely to be sent several times during the live migration process, which leads to long migration times and thus wastes valuable network bandwidth. Secondly, as a VM easily can dirty memory pages faster than these can be transferred over the network, extended *migration downtime* commonly occurs, potentially causing service interruption.

In this contribution we analyze and address these two main problems by using a combination of techniques. To reduce the *total migration time* we propose a method to dynamically adapt the transfer order of memory pages, *dynamic page transfer reordering*. In this scheme, the page update frequency is sampled and this information is used to calculate a page weight, which is then used to prioritize the transfer of less frequently updated pages before the busy ones. By leaving busy pages until last, the number of page re-transfers is reduced, thereby reducing the amount of data to be transferred and thus the total migration time. However, this technique does not directly address the second problem. Migration downtime can still be long and to reduce it, we combine dynamic page transfer reordering with delta compression techniques, investigated in a previous contribution [13], to increase the migration throughput and thus reduce migration downtime.

We implement the combined page transfer reordering and compression algorithm as a modification to the KVM hypervisor. The performance in terms of re-sent pages, total migration time for migration, and downtime is evaluated using a series of tests with synthetic benchmarks and a real-world streaming video application. The evaluation demonstrates good results compared to the standard KVM algorithm. For live migration over a 1000 Mbit/s network, the total migration time was reduced with around 30% in the typical case, with savings in terms of the amount of data transferred with up to 39%. For synthetic benchmarks, the migration downtime was reduced by a factor of 10 to 20 depending on working set size.

II. Background

Live migration enables increased flexibility in provisioning of resources and current VM hypervisors have support for live migration with downtimes as low as tens of a second when migrating over Local Area Networks (LANs). Wide Area Network (WAN) migration, as demonstrated by Ramakrishnan et al. [11] and Travostion et al. [14] usually involves both longer migration times and downtimes.

In order to live migrate a VM its runtime state must be transferred from the source to the destination with the VM still running. If the VM's file system is kept on a network share accessible to both source and destination it need not be migrated. This is difficult in cross-site and WAN migration so in these cases the file system needs to be migrated. In this contribution we only consider memory migration, but our algorithms could be adapted for storage migration.

A. Typical live migration algorithm

There are several variations of the live migration algorithm but most implementations share the same basic idea, first

proposed by Clark et al. [3]. The migration starts with the hypervisor marking all memory pages as dirty. The algorithm then iteratively transfers dirty pages over the network until the number of pages remaining to be transferred is below a certain threshold or a maximum number of iterations is reached. Transferred pages are marked as clean by the hypervisor. Notably, as the VM operates as usual during live migration, already transferred memory pages may be dirtied during an iteration and must thus need to be re-transferred. To stop further memory writes and enable transfer of the remaining pages, the VM is at some point suspended on the source. When the complete memory contents has been transferred, the VM is resumed at the destination and the live migration is complete. An illustration of this process can be seen in Figure 1. This figure also illustrates two important performance criteria for (live) migration: migration downtime and total migration time.

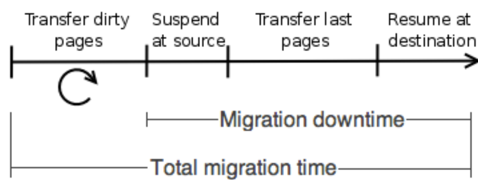

Fig. 1. Overview of a typical live migration process.

In addition to transparency to users in terms of non-interrupted service operation, other requirements for live migration include low impact on the performance of the running VM and any co-hosted VMs. If the live migration process uses too much system resources, VM performance suffers and in the worst case, service is interrupted. Another requirement is that live migration should be transparent to VMs and applications running inside the VMs so that they need not be migration aware in any way.

III. Potential Problems with Live Migration

There are two main challenges for live migration that can limit its usefulness, especially in WAN migration scenarios, extended total migration time and extended migration downtime. While extended migration downtime hurts service operations, extended total migration time is harmful to the infrastructure as more resources are consumed. In this section we describe them in more detail and discuss their effects on VMs and hosted applications during live migration.

A. Extended total migration time

When migrating large memory intensive VMs and/or migrating over slow network links, the amount of data that needs to be transferred over the network can be large which leads to long migration times. Most hypervisors keeps track of which pages that are modified by using a dirty page bitmap, a memory structure where a bit is set for each dirty page. During live migration, the bitmap is scanned from top to bottom, starting with the lowest RAM offset. If a page is marked as dirty in the bitmap it is transferred to the destination. For VMs running memory intensive workloads, there is a high probability that frequently updated pages are transferred multiple times during the live migration process since they are being dirtied again between iterations. Since only the final version of a page is needed, re-transfers are an unnecessary waste of bandwidth and also increase the total migration time. This page resend problem was first identified by Clark et al. [3] and they also propose a solution where a page that has been dirtied since the last iteration is skipped. Although this approach can reduce the number of page resends it is a rather blunt tool, for example, pages that are dirtied every other iteration are still re-transferred.

For the majority of migration scenarios, it is desirable to reduce the total migration time. For example, in large multi-VM data centers such as federated clouds [12] [5], VM migration must be performed as fast as possible as to free up the VM resources for other use and adapt quickly to meet business level objectives such as cost or energy consumption reduction. There is also the issue of resource consumption. As the live migration process uses significant network resources, the longer the migration runs, the more resources it consumes. This harms the overall system performance, both for the services provided by the VM and for any co-located VMs in the same data center.

B. Extended migration downtime

Since the VM continues to write to memory during live migration and memory bandwidth is orders of magnitude faster than network bandwidth, there is a risk that the memory pages cannot be transferred over the network as fast as they are being dirtied. In these cases, the VM must be suspended prematurely, potentially with a considerable amount of memory left to transfer, which can lead to extended migration downtime. This downtime depends not only on the VM size and network speed but also on the type of workload the VM is running. Memory and CPU intensive workloads are harder to migrate because of their high memory dirtying rate. It has been demonstrated by Liu et al. that for VMs as small as 156 MB of RAM, migration downtime can be up to 3 seconds over a Gigabit network [9]. In our previous work, downtimes of up to 28 seconds were measured for a 1 GB VM migrated over fast Ethernet [13].

Extended migration downtime can lead to several issues including service interruption, consistency issues, and unpredictable performance. The main problem is that if the downtime is too long, network connections time out and are dropped which leads to service interruption. Another problem is that during critical execution phases, extended downtime often leads to missed timers, delayed events, or clock drift. Server applications that make use of transactions and triggers rely heavily on precise scheduling, timers, and the capability of the underlying database system to perform operations efficiently and according to the ACID [6] properties. They are thus sensitive to variations in delay and these kind of issues can lead to data corruption and/or crashes.

IV. Algorithms for High Performance Live Migration

To address the page resend problem, we propose an approach, *dynamic page transfer reordering*, where the transfer of less frequently written pages is prioritized over frequently updated ones. By saving the frequently updated pages for last, the risk of having to re-transfer these pages is reduced. To achieve this, we calculate a page weight for each page based on the number of times a page has been updated during the migration process and transfer pages in order of page weight.

In a previous contribution [13] we propose a live migration algorithm that uses delta compression to increase migration throughput and thus reducing the migration downtime. By combining dynamic page transfer reordering with delta compression, we can increase reduce both the total migration time and the migration downtime.

Notably, dynamic page transfer reordering and delta compression are not only complementing techniques, but there are also synergy effects of using them in combination. As dynamic page transfer reordering prioritizes less frequently updated pages, these pages are typically sent at the start of the live migration process and the most frequently update pages are more likely to be transferred towards the end of the iterative dirty page transfer phase. As the number of pages remaining to transfer is smaller in later iterations and these pages are frequently updated, there is a high probability that a page that needs to be re-sent exists in the delta compression cache. To summarize, ideally, less frequently updated pages are sent only once while the frequently updated pages are re-sent in compressed form, thereby increasing migration throughput towards the end of live migration

As discussed earlier, the live migration process must not hurt the performance of the VM to be migrated or any co-hosted VMs. This stipulates the requirement that the algorithm has to be lean in terms of CPU and resource utilization. As the number of memory pages can be in the order of several millions for large VMs, code added to the live migration algorithm can severely affect the performance of both live migration and VM operation when not performing migration. Care must be taken as to design efficient algorithms that do not waste resources. In the remainder of this section, we outline the standard KVM live migration algorithms and our improvements to it.

A. Standard KVM live migration algorithm

The standard KVM migration algorithm for dirty page transfer scans the memory contents from the lowest address to the highest. For each page, the algorithm checks the dirty page bitmap to see if the page is dirty and thus needs to be transferred. If a page is empty, i.e. all zeroes, it is treated as a special case where only a header flag is sent instead of the full page. To avoid saturating the network and slowing down the VM, the scan exits after a certain amount of data has been sent and the dirty page bitmap is updated. At this stage, the hypervisor also calculates the estimated remaining time of migration. If this is below a certain threshold, live migration moves on to the stop-and-copy phase.

Because this algorithm transfers pages in a in a top to bottom order, it suffers from the page resend problem. Also, since it does not use any other form of compression than the empty page exception, migration throughput for non-zero pages is limited to the network bandwidth.

B. Improved algorithm

To realize our proposed dynamic page transfer reordering technique we calculate a page weight for each page according to its update frequency. Our algorithm then transfers the pages by order of this page weight. As these page weights must be re-calculated fast and efficiently between page transfer iterations and for the overhead reasons discussed earlier, a lean mechanism is preferred. Other requirements include the ability to group several pages together into one priority class. To meet these requirements, we add a page priority map on top of the dirty page bitmap. The page priority map organizes pages by different page weights as illustrated in Figure 2.

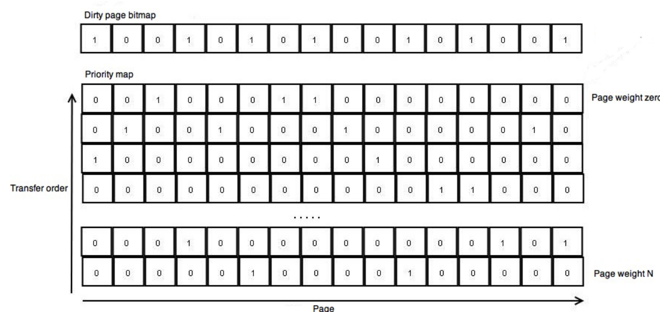

Fig. 2. Page priority map.

To populate this map the page update frequency information must be obtained and one way of counting page updates is to modify the hypervisor's VM memory write functionality. Pages that are being most frequently updated can then be given higher page weight, i.e., be pushed down the page priority map. However, as we do not want to hurt the performance of the VM outside of live migration this approach is not ideal. Instead, we manage the page update information between live migration iterations only, in conjunction with the dirty page bitmap update. The dirty page update algorithm is modified to keep track of the number of times each page is being updated and a page weight is calculated. The page weight update function can be implemented in a number of ways by taking different factors into account such as latest update iteration, number of resends, current page weight, etc. As a proof-of-concept we have settled on a additive increase/decrease scheme with different factors for increase and decrease based on the number of updates. The rationale behind this scheme is that if a page has been updated multiple times during live migration it should be pushed faster down the priority map than a page that has been updated once or twice only. An in-depth study of page weight calculation schemes is beyond the

scope of this paper. However, we have implemented a variation where page weights are randomized, making the algorithm transfer the pages in a pseudo random block order. In the evaluation we compare this version with our selected approach to see if transferring pages by update frequency actually makes a difference or if it is the non-sequential scan transfer order of pages that accounts for the improvement.

Our proposed algorithm can be modified to handle page weight by blocks of pages instead of single pages. Due to the relatively small page size compared to the memory structures handled by modern server applications, if a page is updated, it is likely that its neighboring pages are also updated in the near future. Handling blocks instead of pages saves space for the priority map and speeds up priority calculations and memory scans during live migration. However, a too large block size can lead to a coarse page weight assignment and the algorithm might fail to transfer pages by order of update frequency.

1) Delta compression extension: The idea of delta compression is to increase migration throughput by reducing the amount of data transferred. This is achieved by sending XOR deltas between page versions instead of the full page contents. As the hypervisor only keeps the most recent version of a page, previous versions must be stored in a cache to enable creation of the delta pages. Since the XOR delta page is the same size as the full page it has to be compressed prior to transfer in order to increase the migration throughput. The delta compression extension used in this evaluation is derived from the implementation used in our previous work [13] and adapted to work in parallel with the dynamic page transfer reordering technique.

2) Combined algorithm: To implement dynamic page transfer reordering, the standard live migration algorithm is modified to transfer pages by order of page weight. A current page weight value is maintained and incremented every time the page scan starts over from the lowest address. Only pages whose page weight matches the current page weight are considered during each scan. The current page weight counter is reset to zero between iterations in conjunction with the updates of the dirty bitmap and page weight. Note that only the page weight counter is reset, not the actual page weights. If the hypervisor runs out of pages at the current page weight, it moves on to the next weight level. This approach means that less frequently updated pages are prioritized, leading to a decrease in page re-sends.

The algorithm is also modified to support compression and caching. Pages are stored in a 2-way set associative cache [4]. If a previous version of a page exists in the cache, a delta page is created by applying a binary XOR operation to the old and the new version of the page and this delta page is compressed using binary RLE compression [10]. The RLE algorithm is particularly suitable in this case since the delta pages consists of sequences of zeros and ones. The compressed delta page is then sent instead of the full page.

Finally, the standard KVM approach to calculate the migration bandwidth in terms of transferred bytes/second is not suitable for our combined algorithm. As delta compressed memory pages are much smaller than uncompressed ones and there is a certain overhead for caching and compression, the bandwidth calculation would render a too small value, compared to how many pages that have actually been transferred. To rectify this problem, we utilize the number of sent pages instead of sent bytes as a measure of progress.

C. Implementation

Our dynamic page transfer reordering live migration algorithm is implemented as a modification to the KVM hypervisor. In KVM, the dirty page bitmap resides in a kernel module and is updated when a page write occurs. The obvious place to manage the page priority calculation functionality is when this bitmap is updated. However, apart from modifications to the kernel modules being cumbersome, such a solution also risks slowing the overall performance of VMs as this additional code always is executed, not only during migration. Instead, we can leverage the fact that KVM maintains a copy of the dirty page bitmap in userspace code. This bitmap is updated against the kernel module copy after each live migration iteration. Since the userspace copy is updated only during live migration, code inserted here does not affect the performance of the VM when the VM is not migrated. A drawback with this approach is that when live migration is initiated, all pages have the same update count. This can however be rectified by triggering the dirty page bitmap update function a while before live migration starts.

The delta compression implementation poses similar challenges. In order not to degrade performance the caching and compression algorithms must be lean and efficient. Each CPU cycle spent on caching is a waste if there is a cache miss. Also, the compression scheme must be efficient so that the time spent on compression is outweighed by the shorter transfer time of the compressed pages. To fulfill these requirements we have chosen a 2-way set associative caching scheme in combination with word-wise Run Length Encoding compression. An in-depth discussion of these techniques can be found in our previous contribution [13].

V. Evaluation

To evaluate the performance of our algorithms, we perform a series of live migration tests with VMs running two kinds of workloads with varying working set sizes. In these tests, the standard KVM Algorithm, denoted *Vanilla*, is compared to the algorithm proposed in this paper, denoted *PRIO*, where dynamic page transfer reordering is combined with delta compression, and a version with only the delta compression modification, henceforth called *XBRLE* (XOR Binary Run-Length Encoding).

A. Experimental scenarios

To put load on the VMs, the LMBench [2] benchmarking software is used. The LMBench benchmark generates a very high page dirtying rate by allocating a big block of memory and then continuously overwriting the memory contents through a series of 4 byte store and increments. Using several

instances of LMBench as workload, we vary the size of the working set and perform a series of migrations where we measure migration downtime, total migration time, amount of data transferred, and number of page resends.

Finally, to evaluate the PRIO algorithm's performance in a real world scenario, we live migrate a streaming video server over a limited bandwidth link to simulate a cross-site migration. The streaming video scenario is an example of a real world application where the memory data already is compressed as the video buffer is in h.264 format in our case. The details of the setup of the experimental scenarios are presented in Table I.

TABLE I
SUMMARY OF EXPERIMENTAL SCENARIOS.

Scenario	VM Size	Workload	Network	Algorithms
Migration downtime	2GB, 1 vcpu	LMBench	1000 Mbits/s	Vanilla, XBRLE, PRIO
Total migration time	2GB, 1 vcpu	LMBench	1000 Mbits/s	XBRLE, PRIO
Streaming video	512MB, 1 vcpu	VLC video server	100 Mbits/s	Vanilla, PRIO

B. Experimental setup

The evaluation was performed on two 3.06 GHz HP G6950 servers with 8 GB of RAM. The version of KVM used for the evaluation is 0.13.0. This version is used both as is (the Vanilla algorithm) and as a basis for the two modified versions. The cache size for the delta compression part of the algorithm was set to match the size of the working set in the VMs.

C. Experimental results

In this section, the results from the evaluation scenarios are presented. It can be seen that, in our tests, the PRIO algorithm outperforms the Vanilla and XBRLE algorithms in terms of transferred data and migration time while obtaining the same low migration downtime as the XBRLE algorithm.

1) Migration downtime comparison: In the first scenario, we compared the migration downtime for the Vanilla, XBRLE, and PRIO algorithms live migrating a 2 GB VM running LMBench with various amounts of RAM allocated, spanning from 64 to 1024 MB. The results can be seen in Table II. It is obvious that, even over Gigabit Ethernet, the Vanilla algorithm simply cannot keep up with memory intensive VMs with a large working set. Already with a working set size of a mere 64 MB the downtime exceeds 1 second and for 256 MB it is 3 seconds, which can be fatal to applications that rely on transactions and database connections.

Using delta compression, the migration throughput is increased and it can be seen that the XBRLE and PRIO algorithms both significantly outperform the Vanilla algorithm. The very short downtimes (less than 0.5 s) observed for both algorithms also for the largest working set sizes are well below the SRTT (TCP connection timeout threshold) of 5 seconds for Gigabit Ethernet. This means that both algorithms can be used to obtain sustained service operation for common client-server applications with 1024 MB working sets. Compared to each other, the algorithms have similar performance, although, with smaller working sets, the XBRLE algorithm achieves a slightly lower downtime, potentially due to additional hypervisor overhead induced by the slightly more complex PRIO algorithm.

TABLE II
LIVE MIGRATION DOWNTIME.

WS Size	64 MB	128 MB	256 MB	512 MB	1024 MB
Vanilla	1 s	1.6 s	3 s	5.8 s	9.1 s
XBRLE	0.05 s	0.8 s	0.1 s	0.2 s	0.35 s
PRIO	0.1 s	0.2 s	0.26 s	0.3 s	0.36 s

2) Total migration time comparison: In the second scenario, we compare total migration time, the amount of transmitted data, and the number of page resends for two different working set sizes, 512 and 1024 MB. The maximum allowed migration downtime was in this case set to 300 ms. As demonstrated in the previous scenario, the Vanilla algorithm would require a migration downtime of 6 and 9 seconds respectively for the two selected workload sizes and it is therefore left out of this scenario. The results from the downtime comparison can be seen in tables III and IV. From Table III it can be seen that migration time for PRIO is reduced from 32 s to 20 s in the 512 MB case and from 52 s to 36 s in the 1024 MB case, both when compared with XBRLE. The reason for this decrease in downtime is illustrated in Table III where it can be seen that the amount of transferred data for PRIO is reduced by about 39% for 512 MB working set and by 26% for 1024 MB.

To further study the second of these criteria, the number of page re-transfers, the hypervisor was modified to log the number of page sends during migration. As seen in Table IV the PRIO algorithm only sends four pages more than three times and just a small amount of pages more than two times, while in the XBRLE case, a significant amount of pages are being sent three and four times, accounting for the difference in transferred data between the two algorithms.

TABLE III
MIGRATION TIME/TRANSMITTED DATA.

VM Size	512 MB	1024 MB
XBRLE	32 s / 657 MB	52 s / 1316 MB
PRIO	20 s / 404 MB	36 s / 972 MB

TABLE IV
PAGE RESENDS FOR 1024 MB WORKING SET SIZE.

# Resends	1	2	3	4	5	6
XBRLE	528k	265k	262k	94k	23k	4
PRIO	528k	225k	48k	3	1	0

3) Streaming video: In the final scenario, a VLC streaming video server streaming a 720p video, was migrated using the PRIO algorithm using the Vanilla algorithm as comparison. The network bandwidth was limited to 100 Mbit/s to simulate a cross-site cloud migration scenario. The maximum migration

downtime was set to 500 ms which is as low as the Vanilla algorithm can manage in this scenario. This downtime is short enough for the gap in the data stream to fit inside the video buffer on the client, which means that the migration is transparent to the end user. In this scenario, the PRIO algorithm was also tested against a modified version where the pages where transferred in a pseudo-random order instead of by update frequency, denoted *Pseudo-random*.

The results from the test are shown in Table V. As seen in this table, total the migration time is reduced by 30% and the amount of transferred data is reduced by 51%. Notably, the pseudo-random algorithm also outperforms the Vanilla algorithm although not by much. For this use case, the properties of the application, more particularly, the fact that the video stream already is compressed, makes streaming video less suitable for the delta compression scheme. As even the Vanilla algorithm can achieve a short enough downtime in this case (with the help of client side buffering), the benefit of delta compression is neglectable from a downtime perspective. However, as compression also helps to reduce the amount of data transferred, this technique is useful when combined with page transfer reordering.

TABLE V
HD VIDEO MIGRATION.

	Total migration time	Transferred data
Vanilla	22.1 s	459 MB
Pseudo-random	20.1 s	389 MB
PRIO	15.4 s	225 MB

VI. DISCUSSION

In this section we discuss differences between the three live migration algorithms evaluated in the paper and their effects on live migration. We also discuss how the page weight of the PRIO algorithm can be used to identify which VMs are most suitable for migration.

A. Extended downtime and total migration time

Extended downtime is caused by the live migration algorithm's inability to keep up with the dirtying rate of the VM to be migrated, causing the algorithm to reach a steady state where the VM has to be suspended in order to transfer the remaining memory pages. As the XBRLE and PRIO algorithms compress the pages, they show a decrease in migration downtime because the hypervisor has the possibility to catch up as the cache hit ratio rises, improving the migration throughput. The XBRLE algorithm's slightly shorter downtime in the 64 to 256 MB cases is caused by the increased overhead in the PRIO algorithm. This makes more of a difference with smaller working set sizes, but the difference is marginal and at 512 MB the PRIO algorithm has caught up.

The total migration time varies with the amount of data transferred during live migration and with the migration throughput. Using the Vanilla and XBRLE algorithms, there is a high probability of the same pages being retransmitted during subsequent iterations due to the page resend problem, thereby increasing the amount of data being transferred. In the PRIO case however, if the page weight update function is successful in organizing the pages in order of update frequency, the algorithm tends to transmit different pages each iteration which leads to fewer iterations and less data being transferred in total. This is clearly visible in our second evaluation scenario, the total migration time comparison, and also in the streaming video scenario.

B. Suitable scenarios for dynamic page transfer reordering

If all pages have the same update frequency, there is a risk that the algorithm only chases the pages down the priority map, as pages receive similar page weights, leading to no gain in migration downtime. Dynamic page transfer reordering performs best with workloads where the VM working set has unevenly distributed page updates. Ideally, these pages are transmitted in order of update frequency. In our evaluation, we use the LMBench software which has an uniform distribution of page updates within the allocated memory block. However, as several LMBench instances of different sizes are used to add up to a larger working set size, this leads to a more heterogeneous distribution of updates and the PRIO algorithm is still successful in reducing the total migration time. Finally, in the streaming video case, using VLC server, the page updates are unevenly distributed and good results are obtained for the PRIO algorithm.

C. Impacts on live migration predictability

In cloud environments it is often desirable to migrate VMs between sites, e.g., to offload capacity, reduce costs, improve server consolidation, or bring services closer to users. It is not uncommon that several candidates exists as to which VMs to migrate. Traditionally, most placement algorithms for VMs do not take into account the time it takes to migrate a VM, the *migration cost*. Recent work [8] has outlined methods where the migration cost is taken into account. However the migration cost is considered the same for all VMs of a certain type. Since the migration time is directly dependent on the working set size, total amount of memory, and the migration throughput, it is beneficial to have a good estimate of the current working set size. We propose that the page weights can be used to estimate this. By collecting data for a period of time before migration, a reasonable guess as to the size of the working set can be made by examining the page weight distribution. This information can then be used in conjunction with CPU usage and total memory size to determine which VMs are best suited for live migration and approximate the cost in terms of downtime and resources to migrate them.

VII. RELATED WORK

The use of delta compression to increase migration throughput and reduce downtime is studied by us in a previous contribution [13]. Another use of delta compression is by Wood et al. [15] who propose a strategy to increase live migration performance between cloud sites. In addition to using delta compression, their solution involves data deduplication. Zheng

et al. [16] propose an improved VM disk storage block live migration algorithm that includes an analysis of the write history to storage blocks. They use this information to schedule the transfer order of the blocks, thereby reducing the amount of storage data being transferred during live migration. The idea of leveraging update frequency when migrating is similar to our page transfer reordering scheme but their approach is however concerned with storage migration only. Storage migration poses different challenges than memory migration, for example due to the slower updates of storage blocks compared to memory pages.

The page resend problem was identified by Clark et al. [3]. They propose a combination of techniques to reduce its impacts, by scanning the memory structure in a pseudo-random order and skipping pages that have been dirtied in the previous iteration. However, they do not analyze the page update frequencies or transfer pages by them so their approach is dependent of the success of the pseudo-random scan to select the proper pages to minimize page resends.

An alternative approach to live migration is the use of pull techniques, also known as post-copy migration. In this case, the processor state is transferred at the start of migration followed by the memory contents. Since the processor state is transferred the VM can be started before the full memory contents have been transferred. If a page is missing at the destination when the VM requests it, this page has to be transferred over the network before execution can continue. This approach is implemented and investigated by Hines et al. [7]. Their evaluation indicates that the page resend problem is virtually eliminated, resulting in very low migration downtime. However, in their evaluation an average of 21% of the pages in a VM generate network page faults, suggesting degraded service performance for some time after migration. If these page faults occur close in time to each other, the correct operation of transaction sensitive and real-time applications might be at risk. Furthermore, pull-based migration may be unsuitable in maintenance scenarios, where the source host and/or network link may need to be shut down rapidly.

VIII. Conclusion and Future Work

The goal of our algorithms is to tackle both the extended migration time and the extended downtime problems that exist in current VM live migration approaches. Our evaluation demonstrates that the use of page update priorities to reduce page resends during migration is a promising approach and reduces migration time in several cases. It also shows that the page priority approach can be combined with delta compression techniques to achieve the reduction in migration downtime associated with such techniques.

Our proposed algorithm is highly dependent on fast and efficient schemes for page privatization and compression. To further improve the performance of live migration, different page weight calculation schemes can be investigated. The overall goal is a lean page weight calculation that can dynamically adjust to achieve a suitable distribution of page weights for different kinds of workloads. To do this, an in depth study of how the hypervisor writes to pages before and during migration would be useful. This problem is very similar to the well studied one of how to design optimal page replacement algorithms for page tables [1]. Finally, the page weight calculation algorithm could be made more accurate if larger data samples are collected by tracking page writes also during normal hypervisor operation and not only during live migration. Such an always-on sampling algorithm must also be implemented in a manner that does not hurt the performance of the VM operation.

Acknowledgments

We acknowledge Aidan Shribman, Tomas Ögren, and Eliezer Levy for their contributions to this work. Financial support has been provided by the EC FP7 under grant agreements no. 215605 (RESERVOIR) and 257115 (OPTIMIS) as well as by UMIT research lab and the Swedish Government's strategic research project eSSENCE.

References

[1] A. Aho, P. Denning, and J. Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[2] Bitmover. Lmbench, 2010. http://www.bitmover.com/lmbench/.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI '05*, pages 273–286. ACM, May 2005.

[4] U. Drepper. Memory part 2: CPU caches. http://lwn.net/Articles/252125.

[5] A. Ferrer, F. Hernandez, J. Tordsson, E. Elmroth, et al. OPTIMIS: a holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28:66–77, 2011.

[6] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–317, 1983.

[7] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09*, pages 51–60. ACM, 2009.

[8] W. Li, J. Tordsson, and E. Elmroth. Modelling for dynamic cloud scheduling via migration of virtual machines. 2011. Accepted.

[9] 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.

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

[11] 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*, pages 262–267. ACM, August 2007.

[12] 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.

[13] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *VEE '11*, pages 111–120. ACM, 2011.

[14] F. Travostino. Seamless live migration of virtual machines over the MAN/WAN. In *SC '06*, page 290. ACM, November 2006.

[15] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe. Cloudnet: dynamic pooling of cloud resources by live WAN migration of virtual machines. In *VEE '11*, pages 121–132. ACM, 2011.

[16] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai. Workload-aware live storage migration for clouds. In *VEE '11*, pages 133–144. ACM, 2011.