Efficient Live Migration of Virtual Machines with a Novel Data Filter

Yonghui Ruan, Zhongsheng Cao, and Yuanzhen Wang

School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
caozhongsheng@126.com

Abstract. Live migration of virtual machines (VM) is useful for resource management of data centers and cloud platforms. The pre-copy algorithm is widely used for memory migration. It is very efficient to deal with the memory migration of read-intensive workloads. But for write-intensive workloads, the pre-copy's straightforward iteration strategy will become inefficient. In this paper, we propose a novel data filter to improve the pre-copy algorithm in this inefficient situation. In each round of iteration, the data filter forecasts the pages which will be subsequently dirtied, and then filters them from the send list. This prevents the pages from being repeatedly transmitted, thus reducing migration time and bandwidth resource consumption. Meanwhile, the data filter also checks if the previously filtered pages should be re-added to the send list. This ensures that the downtime will not be increased. Experimental results show that the improved algorithm effectively reduces the amount of migrated data, while keeping the downtime at the same level.

1 Introduction

Live migration of virtual machines (VM) is a powerful tool which allows for the relocation of VM between different physical hosts [1, 3–6, 9, 15, 20]. The whole software stack can be consistently transferred, while the continuous execution of the workload is guaranteed. Live VM migration provides considerable flexibility for many tasks of data centers and cloud platforms, including load balancing, online maintenance and fault management of the system [12, 14], and physical server integration [13], etc.

Live VM migration involves migrating the VM's memory data, network connection and virtual devices. In practice, the VM image file is usually stored in a network-attached storage (NAS) device. Therefore the disk storage does not need to be migrated. In this paper, we focus on the memory migration issues.

The pre-copy algorithm is a widely used memory migration approach which consists of two phases: an iteration phase and a stop-and-copy phase. In the iteration phase, the memory pages are copied to the destination in an iterative way, while the VM is still running at the source. This phase continues until either a maximum iteration count is reached, or a sufficient number of pages are synchronized, whichever comes first. Then, in the stop-and-copy phase, the VM is suspended and the remaining pages are copied to the destination.

In each round of the iteration phase, the pre-copy's iteration strategy tries to transmit all the pages which have not yet been synchronized. If a transmitted page is subsequently dirtied, it is re-sent in the next round. When encountered with write-intensive workloads, this strategy will cause a lot of repeated transmissions, which waste bandwidth resources and can not improve the downtime.

In this paper, we propose a novel data filter to improve the pre-copy algorithm in this inefficient situation. In each round, the data filter forecasts the pages which will be subsequently dirtied, and then filters them from the send list. This prevents the pages from being repeatedly transmitted, thus reducing the migration time and bandwidth resource consumption. Meanwhile, previously filtered pages will be reconsidered, to see if they can be added to the send list. Therefore, our migration algorithm can still transmit as many pages as possible in the iteration phase. This ensures that the downtime will not be increased.

The core of the data filter is a forecasting algorithm, which is used to forecast dirty pages in the iteration phase. To reduce the cost of analyzing the high rate input data stream of memory write, we propose a *state transition model of memory write* based on the analysis of the principle of locality. Furthermore, we propose the concept of the *local writable working set* (LWWS) to facilitate the analysis of the memory write behavior in a local time period. Based on these conceptions, we designed an applicable forecasting algorithm with sufficient forecasting accuracy.

We implement our algorithm based on Xen virtualization software and run experiments on a variety of workloads. Experimental results show that our algorithm effectively reduces bandwidth resource consumption, while achieving the same level of downtime.

The main contributions of this paper are as follows:

– We propose a state transition model to analyze memory write pattern, and propose the concept of the local writable working set to facilitate the analysis of memory write behavior in a short time period.
– We propose a novel data filter based on the state transition model and the local writable working set to improve the pre-copy's iteration strategy.
– We present the improved pre-copy algorithm, show detailed analysis of the algorithm, and provide a thorough evaluation using a variety of workloads.

The rest of the paper is organized as follows. Section 2 provides a survey of related literature, section 3 describes the data filter, section 4 describes the live migration process with our algorithm, section 5 evaluates our algorithm with a variety of workloads, and section 6 concludes this paper.

2 Related Work

Nicolae et al. [10] propose an approach of live virtual disk migration. For a disk I/O intensive workload, the local storage is required. Therefore, the migration of massive data in the virtual disk is involved in the live VM migration. While different from their study, we consider the general case of a shared storage structure, in which the VM accesses the virtual disk data through a NAS.

The pre-copy algorithm is widely used for memory migration. In the study of Clark et al. [1], the concept of the *writable working set* (WWS) is proposed. They track dirty pages under various workloads during normal operational phase of VM, and use this information to adjust the network rate for optimization. The WWS in their research is extended to the LWWS to facilitate the analysis of the memory write behavior in a short time period.

Liu et al. [6] propose a live migration algorithm called CR/TR-Motion that is based on checkpointing/recovery and trace/replay technology. Their algorithm sends the logs of execution trace instead of memory pages to achieve good migration efficiency for both LAN and WAN environments. The idea of forecasting and filtering in our algorithm can not be applied in combination with their method, since the trace/replay technology processes VM operations but not the memory pages.

Jo et al. [20] propose a technique to reduce the migration time while keeping the downtime to a minimum. They track the I/O operations between the VM and the NAS to maintain a map of the pages that reside on the storage device. For these pages, the memory-to-disk map is transmitted instead of the data itself. So these pages can be directly obtained from the NAS after the map is transmitted. As less pages are transmitted, the migration time is saved. On the other side, our algorithm largely improves bandwidth resource consumption and maintains the downtime at the original level.

There are many other approaches which optimize memory migration process, such as memory compression [4], migration throttling [1, 2, 9], and DSB(Dynamic Self-Ballooning) [3]. These studies try to optimize migration without changing the pre-copy algorithm itself. As a result, the data filter can be used in combination with these approaches to get better performance.

3 Data Filter

In this section we present the design of the data filter. As shown in figure 1, The data filter filters the original send list of each round to make a new send list. The source then sends pages to the destination according to the new list. Meanwhile, the pages that are filtered and updated in current round form the send list of the next round.

The core of the data filter is a forecasting algorithm, which forecasts dirty pages in the iteration phase. Below we introduce the state transition model of memory write, the framework of our forecasting algorithm.

3.1 State Transition Model of Memory Write

Given an observation period, let the frame number of dirty pages be the input. To forecast which pages are going to be dirtied in a given time interval, a forecasting model based on the analysis of the input data is needed.

Assume a workload with 1GB/sec rate of memory access, where the proportion of memory write is 10%. With a 4KB page size, 25.6K pages are dirtied

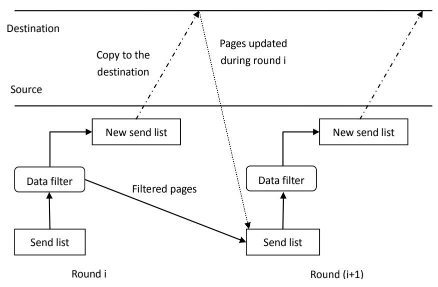


Fig. 1. Data filter

per second. That is, the input data stream has a rate of over 26,000 data points per second. In fact, some workloads may have even faster rates of memory write. The live migration task cannot afford the cost of analyzing such a high rate data stream.

In order to cope with this difficulty, we note that most of the repeated transmissions are actually caused by the frequently modified pages. This suggests that a forecasting model can be built by analyzing these frequent pages, which usually account for a small part of the VM memory.

A straightforward way to do this is to count how many times each memory page is modified during a normal operational phase of the VM, and identify the frequent pages among all the memory pages. However, the frequent pages which are analyzed in this way are the global frequent pages and almost useless. Because once they are filtered, there is no way to re-add them to the send list. If all of them are filtered, the downtime may be increased. But if only part of them are filtered, there may be many repeated transmissions which cause waste of bandwidth resources.

To avoid these disadvantages, a dynamic forecasting model based on the analysis of the frequent pages in a local time period is needed. To build such a model, the best knowledge we have is the principle of locality. The temporary locality principle states that if a page is currently being accessed, it is likely to be accessed again in the near future. The spatial locality principle states that the pages that will soon be accessed are close to the page that is currently being accessed. Based on these two principles, we infer that *within a short time duration, the pages which are frequently accessed are more likely to concentrate in some local regions of memory space.*

If we only consider memory write, the frequently modified pages will still gather in the same regions. Furthermore, if the observation period lasts for a long time, we'll find different groups of "hot spot" regions appearing in different short time durations. These groups of hot spot regions can be seen as states of memory write.

Assume that the VM's memory space is divided into a certain number of continuous linear regions of equal length. Within a given short time duration, we use a group of hot spot regions to describe a memory write state. When the time duration of a state has passed, it may stay or convert to a different state. If the state transition rules are also defined, the state transition model of memory writes is determined.

The state transition model is useful in that it helps to simplify the problem of the dirty page forecasting to the memory write state forecasting. Considering the example in the beginning of this section. For a 50ms time duration of each state, the input data stream will have a rate of only 20 states per second. Meanwhile, this rate is independent of the load characteristics.

3.2 Local Writable Working Set

The state transition model is used to predict hot spot regions of memory write. However, each hot spot region may also contain rarely modified pages, which should not be filtered. Therefore, the model needs to be further refined.

In the pre-copy algorithm implemented by Xen [1], Clark et al. proposed the concept of the writable working set. It is the set of the global frequently modified pages of the iteration phase. In order to avoid repeated transmission and achieve a better migration performance, these pages should be transferred via the stop-and-copy phase. To refine the state transition model, we extend the writable working set to the local writable working set.

Definition 1. Local Writable Working Set (LWWS). In the state transition model, each state corresponds to a group of hot spot regions. The frequently modified pages within these regions constitute a local writable working set.

Compared to the WWS, a LWWS is a set of frequently modified pages in a local time period. The LWWS has two useful properties.

Property 1. If a LWWS is associated with a frequent state, then it is a subset of the WWS.

If a state frequently occurs in the iteration phase, the pages of the corresponding LWWS will be the global frequent pages. In other words, these pages also belong to the WWS. Therefore, the LWWS is actually a subset of the WWS.

Property 2. If a LWWS is associated with a infrequent state, then the pages of it are infrequent pages.

Although a LWWS contains frequent pages under a corresponding state, the "frequent" pages related to the infrequent state will still be rarely modified. According to properties 1 and 2, the set of the LWWSs covers both frequent and infrequent pages of memory write.

3.3 Markov Model

In the above discussion, we present the framework of the forecasting model. In this section we discuss the implement of it.

We start from determining how to describe the transition rules between the memory write states. There are many factors that may affect the transition rules, including but not limited to system architecture, load characteristics, resource usage, etc. It involves extensive analytical work to learn how these factors affect the transition rules, and this is beyond the scope of this paper.

We use an alternative probabilistic method that is easier to implement and understand. The state transition process is considered as a stochastic process, where the state depends on previous states in a non-deterministic way.

In order to reduce the algorithm cost, we further simplify the model into two aspects. Firstly, we assume that the state transition process has the Markov property. So the state transition model becomes a Markov model. Secondly, when a group of hot spot regions is observed, only the hottest region is used to describe the corresponding state. Meanwhile, other hot spot regions are still recorded and associated to the state. In this way, we do not need to worry about the state explosion problem in practice.

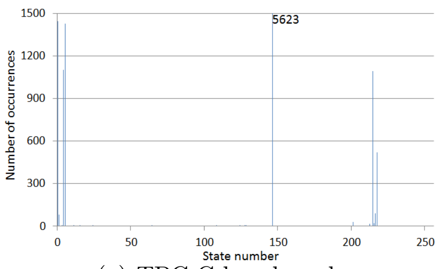
Build the Markov Model: At the normal operational stage of the VM, we use the shadow page table to track dirty pages. Within a given duration, we identify a group of hot spot regions of memory writes, record all the dirty pages, and use the hottest region to describe a memory write state. We also record all the state transitions in a state transition matrix. When the modeling time is over, we calculate the LWWS of each state by identify the frequent pages among all the recorded dirty pages. Finally, the set of the calculated LWWSs alone with the state transition matrix are outputted and used for forecasting dirty pages in the migration process.

Figure 2 shows the 4 different state distribution of the experimental workloads. Xen was running on a machine with a two quad-core 2.13GHz Inter Xeon CPU. We start each workload in one virtual machine with 1GB RAM and 2 VCPU. The dirty bitmap is read every 50ms to identify a state. The size of the state space is 256, and the size of each region is 4MB. During the whole observation period, a total of 12,000 states are identified. More detailed information about the workloads can be found in section 5.1.

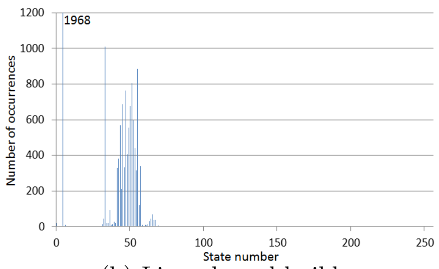
From this data we observe that the state distribution varies between the different workloads. For each workload, there is a group of states which have high frequency of occurrence. These frequent states represent the memory write pattern of each workload. When we treat each state as a local feature of memory write and learn transition rules between different states, we are able to make a fine grained prediction.

A key challenge of applying the Markov model is keeping the time-effectiveness of it, since the workload behavior may change with time. If the workload behavior changes during the migration process, there is no time to rebuild the Markov model. In this case, the data filter is disabled, and the original pre-copy algorithm is performed. In the following discussion, we assume that the memory write pattern does not change during the migration process.

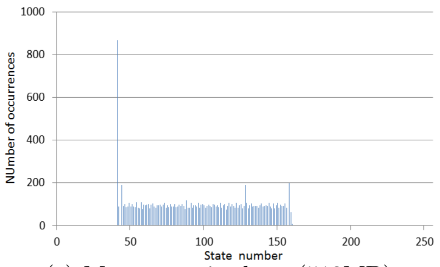
Maintain the Markov Model: After we build the Markov model, we calculate the probability distribution of states and state transitions. Then, we perform a periodically sampling to test whether the current memory write pattern complies with the established Markov model. More specifically, we check the sampling data



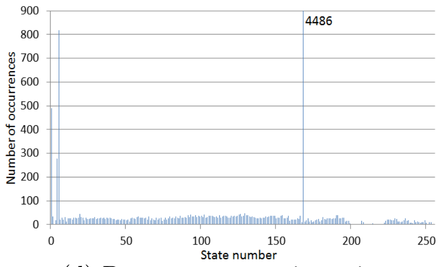
(a) TPC-C benchmark



(b) Linux kernel build



(c) Memory write loop (512MB)



(d) Data compress using gzip

Fig. 2. Distribution of memory write states

to see if it obeys the probability distribution of states and state transitions that are calculated before. If these examinations fail, the Markov model is considered to be out of date and is rebuilt.

4 Live Migration with Improved Pre-copy Algorithm

4.1 Performance Metrics

We use the downtime, the migration time and the amount of migrated data to evaluate the performance characteristics of the VM migration algorithm.

Downtime: Overall suspend time of the VM during the migration process.

The migration downtime measures the availability of the VM during the migration process. An ideal optimization method should not cause the downtime to be increased.

Migration Time: Overall time of the VM migration.

The migration time is the interval between the time migration is initiated and the time the VM is consistently migrated to the destination.

Amount of Migrated Data: Total number of memory pages transferred during the migration process.

We use this metric to measure bandwidth resource consumption of the VM migration. It should be as low as possible on the premise of the unchanged downtime.

4.2 Migration Algorithm

This section presents the improved pre-copy algorithm. Figure 3 shows the pseudo-code of the algorithm (line 2-12) and the data filter (line 13-23).

```
1. let N := memory size of VM

2. MigrationAlgorithm(iterclue, itercount, duration)
3.    let iter := 1
4.    let unsyn := N
5.    let dirtylist := set of all VM pages; filterlist := empty set
6.    while iter < itercount and unsyn > iterclue do
7.          let sendlist := dirtylist + filterlist
8.          set (sendlist, filterlist) := DataFilter(sendlist, duration)
9.          set dirtylist := SendData(sendlist)
10.         set unsyn := length(filterlist) + length(dirtylist)
11.         iter++
12.   Stop-and-copy()

13. DataFilter(sendlist, duration)
14.   let LWWS := empty set
15.   let filterlist := empty set
16.   let sendtime := length(sendlist) / netrate()
17.   let curr := current identified state
18.   while sendtime >= duration
19.         set (curr, LWWS) := Forecast(curr)
20.         set filterlist := filterlist + LWWS
21.         set sendtime:= sendtime – duration
22.   set sendlist := sendlist – filterlist
23.   return(sendlist, filterlist)
```

Fig. 3. Improved pre-copy algorithm

In line 2, the parameter ‘iterclue’ is the threshold of the number of the remaining pages which have not yet been synchronized. When the stop-and-copy phase starts, this number is directly proportional to the downtime. The parameter ‘itercount’ is the maximum iteration count. The parameter ‘duration’ is the time duration for each memory write state.

In the migration algorithm, the pages that are dirtied and filtered in each round are recorded in the ‘dirtylist’ and ‘filterlist’. In line 7, we use these two sets to calculate the ’sendlist’. It contains the pages which have not yet been synchronized in the beginning of each round. In line 9, the function SendData() transmits all the pages in the sendlist and returns dirty page set during the transmission process.

The data filter forecasts and filters dirty pages during the transmission time of the given ‘sendlist’. In line 16, the function netrate() calculates network rate (number of pages transmitted per second). The variable ‘sendtime’ is the transmission time of the pages in the ‘sendlist’. In line 17, the variable ‘curr’ is set to an identified state. In line 19, based on the identified state, the future states are forecasted using the state transition matrix. Each forecasted state is assigned to the ‘curr’ and used for forecasting in the next loop. Then, the pages of the corresponding LWWS is added to the ‘filterlist’ in line 20. As previously discussed, the state transition matrix and all the LWWSs are outputted after the Markov model is built.

5 Experiment

5.1 Test Setup

Our experimental platform consists of 3 identical physical servers, each with a two quad-core 2.13GHz Inter Xeon E5606 CPU, 4GB DDR RAM and Intel 82576 Gigabit Network Connection. The migrated VM is configured to use 2 VCPU and 1024MB of RAM. Xen-3.3.0 is used as the virtual machine monitor. The guest kernel is Linux 2.6.18, and the host kernel is a modified version of Linux 2.6.18 for both the source and the destination. Storage is accessed via iSCSI protocol from the third physical server configured as a NAS.

We implement the improved algorithm based on Xen virtualization software. The experiments use the following workloads:

TCP-C Benchmark: Mysql database and the open source test tool DBT2 [16] are used for the experiment. DBT2 is an implementation of the TPC's TPC-C Benchmark specification. We use a 60 warehouses dataset and 40 client connections. The Mysql database version is 5.1.7, with default settings.

Linux Kernel Build: The second experiment runs a system call intensive load — kernel compilation of Linux 2.6.18.

Memory Write Loop: We write a simple C program that writes constantly to a 256MB region and a 512MB region of memory.

Data Compression with gzip and bzip2: The test data is approximately 7GB of the XML text dump of the English version of Wikipedia [21] on June 4th, 2013, history 4 (approximately 7GB after decompressing with bzip2). We use the command 'gzip' and 'bzip2' to compress the data with the best compression ratio.

Unixbench: It's a fundamental high-level Linux benchmark suite that integrates CPU, file I/O, process spawning and other workloads. The following tails are performed: Load system with concurrent shell scripts, compiler throughput, recursion, dhrystone2 using register variables, arithmetic, pipe throughput, pipe-based context switching, process creation, and execl throughput.

To reduce the effect on other ongoing network services hosted on the source host, we limit the network bandwidth to 500Mbit/sec for the migration daemon. In all cases, the existing parameters of the pre-copy algorithm are set as default. For the improved algorithm, the observation interval is set to 50ms. For each workload, the VM migration of the two algorithms is started at the same time point.

5.2 Overhead of Data Filter

Table 1 shows the time cost for building the Markov model and applying the data filter. For the Markov model, the sampling interval of each state is 50ms, and the computation time is about 10ms. It takes about 60ms in all to identify and record a state. In each round of iteration, the data filter has to spend 50ms in waiting for the sampling results of the dirty page bitmap. After that, it takes a short time to forecast and filter dirty pages. From the third column of table 1 we can see that the time cost of the data filter is between 1.6 to 2 seconds. Noted that the maximum iteration count is 30 by default in Xen, it takes a total of 1.5 seconds to wait for the sampling results.

Let N be the memory size of the VM, let S be the size of the state space. After some simple optimizations, the space cost to record the state transition matrix, the LWWSs and some intermediate results is $O(N+S^2)$ (2.79MB memory space is used in our implementation). In the migration process, the space cost of the data filter is also $O(N + S^2)$ (1.63MB memory space is used in our implementation).

Table 1. Time cost of the Markov model and the data filter

Workloads	Time cost for modeling (millisecond per state)	Time cost of the data filter (millisecond)
TPC-C benchmark	59.735	1673
kernel-build	59.642	1636
gzip	58.232	1640
bzip2	59.982	1667
MW(512MB)	59.973	2000
MW(256MB)	59.987	1972
Unixbench	59.721	1662

5.3 Migration Performance for Different Size of State Space

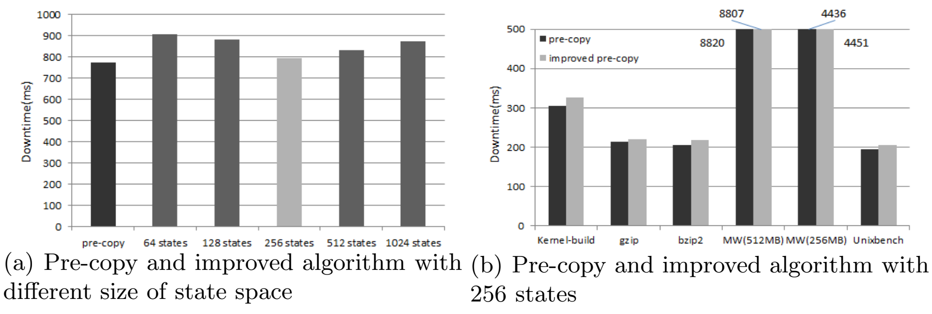
We first evaluate the migration performance of the improved algorithm with different size of the state space. The experiment uses the TPC-C benchmark.

Figure 4 (a), 5 (a), 6 (a) show the downtime, the migration time and the amount of transferred pages of the original pre-copy algorithm and the improved algorithm with 64, 128, 256, 512 and 1024 states. From this figure we observe that when the size of the state space is set to 256 (the region size is 4MB), we obtain the best migration performance.

When a smaller size is used, the performance is worse because the corresponding regions are getting larger. The more a region covers memory pages, the more difficult the corresponding state transfers to other states in memory write process. This makes it hard to reconsider transmitting the previously filtered pages to maintain a low downtime. At one extreme, the whole memory space contains only one region and our forecasting algorithm becomes a static method. In this case, there is no way to re-add the filtered pages to the send list of each round.

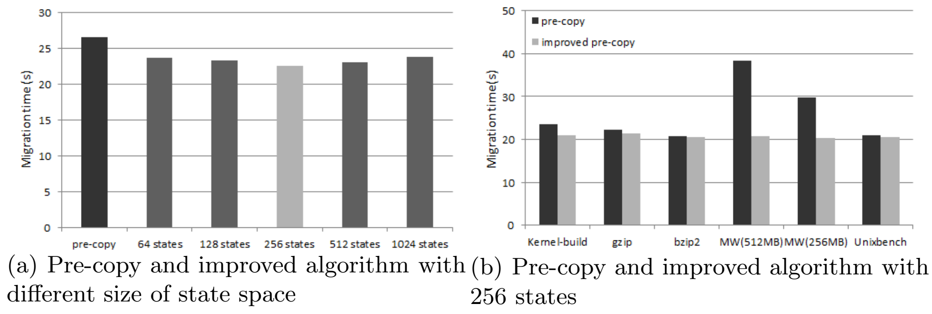
On the other side, the size of the state space should not be set too large. If the memory region covers few pages, more than one region may be full with dirty pages during the time duration of each state. At the other extreme, each region contains only one memory page. It is difficult for the forecasting algorithm to identify the real memory write state in this situation. Therefore, the forecasting accuracy will be reduced and the performance will be worse.

The following experiments are focused on comparison between the pre-copy algorithm and the improved algorithm with 256 states.



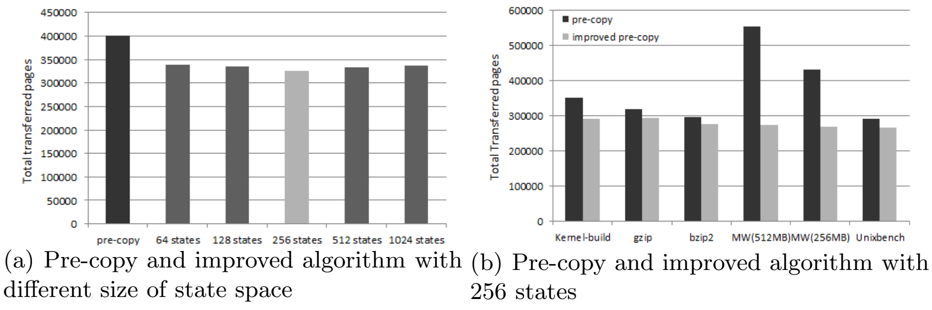
(a) Pre-copy and improved algorithm with different size of state space (b) Pre-copy and improved algorithm with 256 states

Fig. 4. Comparison of downtime



(a) Pre-copy and improved algorithm with different size of state space (b) Pre-copy and improved algorithm with 256 states

Fig. 5. Comparison of migration time



(a) Pre-copy and improved algorithm with different size of state space (b) Pre-copy and improved algorithm with 256 states

Fig. 6. Comparison of total transferred pages

5.4 Downtime

Figure 4 (b) shows the migration downtime for the six remaining workloads, where "MW" refers to the memory write loop workload. First we notice that in the memory write loop workloads, the downtime of the two algorithms is nearly the same. Compared to approximately 8 seconds and 4 seconds downtime, a difference of dozens of milliseconds can be ignored.

Among the other four workloads, the improved algorithm has more downtime than the original pre-copy algorithm. More precisely, the downtime of the improved algorithm increases by 5.5% on average with respect to the pre-copy algorithm. This shows the data filter efficiently maintains the same level of downtime. In the worst case (the kernel-build workload), the improved algorithm has more 21ms downtime than the original pre-copy algorithm. Compared to the total 305ms downtime achieved by the pre-copy algorithm, we can still consider that the two algorithms have the same level of downtime. Therefore, the downtime is not significantly increased because of the data filter.

5.5 Migration Time and Amount of Migrated Data

Figure 5 (b) and 6 (b) show the migration time and the total transferred pages for the six workloads run. For the memory write loop workloads, we see excellent performance of the improved algorithm: both the duration and the bandwidth resources are significantly reduced with nearly no increase in the downtime. In fact, the artificial workloads have regular memory write patterns, which are easy to be captured by the data filter. Although their constant and high dirty rates cause the poor performance of the pre-copy algorithm, they are the ideal workloads for the data filter to show its efficiency.

For the rest of the four workloads, the improved algorithm achieves 73.5MB to 236.1MB reduction in the amount of migrated data compared to the pre-copy algorithm. When it comes to the migration time, the improvement is 0.2 to 1.6 seconds. This is because the data filter spends time in forecasting and filtering dirty pages, as shown in table 1.

6 Conclusion

In this paper we propose and implement an improved algorithm of the pre-copy. When encountered with write-intensive workloads, the improved algorithm is able to provide efficient live migration of VM.

In the iteration phase, the improved algorithm employs a new data filter at the beginning of each round to filter the pages which are likely to be modified in the subsequent rounds. To do this, a lightweight forecasting algorithm is proposed to forecast dirty pages generated during the iteration phase. We extend the concept of the writable working set to the local writable working set, and propose a state transition model. This model largely reduces the algorithm cost, while ensuring the forecasting accuracy. With the data filter, the original send list of each round is filtered. After that, the pages of the new send list are sent to the destination. Unlike any other optimization methods, the improved algorithm still tries to send as many pages as possible in the iteration phase, instead of simply postponing to send them in the stop-and-copy phase.

In the future, we plan to find a way to efficiently shorten the migration time of the live VM migration. On the other side, we also plan to find a more sophisticated model than the Markov model. Based on the complex model, the same level of the migration time and the bandwidth resource consumption as the stop-and-copy algorithm, and the same level of the downtime as the pre-copy algorithm can be achieved.

References

1. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: NSDI, pp. 273–286 (May 2005)
2. Elmore, A.J., Das, S., Agrawal, D., El Abbadi, A.: Zephyr: live migration in shared nothing databases for elastic cloud platforms. In: SIGMOD, pp. 301–312 (June 2011)
3. Hines, M.R., Deshpande, U., Gopalan, K.: Post-copy live migration of virtual machines. SIGOPS Oper. Syst. Rev. 43(3), 14–26 (2009)
4. Jin, H., Deng, L., Wu, S., Shi, X., Pan, X.: Live virtual machine migration with adaptive memory compression. In: Cluster, pp. 1–10 (August-September 2009)
5. Song, X., Shi, J., Liu, R., Yang, J., Chen, H.: Parallelizing live migration of virtual machines. In: VEE, pp. 85–96 (May 2013)
6. Liu, H., Jin, H., Liao, X., Hu, L.: Live migration of virtual machine based on full system trace and replay. In: HPDC, pp. 101–110 (June 2009)
7. Deshpande, U., Wang, X., Gopalan, K.: Live gang migration of virtual machines. In: HPDC, pp. 135–146 (June 2011)
8. Ma, Y., Wang, H., Dong, J., Li, Y., Cheng, S.: Efficient Live Migration of Virtual Machine with Memory Exploration and Encoding. In: CLUSTER, pp. 610–613 (September 2012)
9. Liu, Z., Qu, W., Liu, W., Li, K.: Xen live migration with slowdown scheduling algorithm. In: PDCAT, pp. 104–107 (December 2010)
10. Nicolae, B., Cappello, F.: A Hybrid Local Storage Transfer Scheme for Live Migration of I/O Intensive Workloads. In: HPDC, pp. 85–96 (June 2012)
11. Shetty, J., Anala, M.R., Shobana, G.: A Survey on Techniques of Secure Live Migration of Virtual Machine. International Journal of Computer Applications 39(12), 34–39 (2012)
12. Nagarajan, A.B., Mueller, F., Engelmann, C., Scott, S.L.: Proactive Fault Tolerance for HPC with Xen Virtualization. In: ICS, pp. 23–32 (June 2007)
13. Nathuji, R., Schwan, K.: VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. SIGOPS Oper. Syst. Rev. 41(6), 265–278 (2007)
14. Jhawar, R., Piuri, V., Santambrogio, M.: Fault Tolerance Management in Cloud Computing: A System-Level Perspective. IEEE Syst. J. 7(2), 288–297 (2013)
15. Nelson, M., Lim, B.H., Hutchins, G.: Fast Transparent Migration for Virtual Machines. In: USENIX ATC, pp. 391–394 (April 2005)
16. Database Test Suite, `http://sourceforge.net/apps/mediawik-i/osdldbt/`

17. Kumar, S., Schwan, K.: Netchannel: A VMM-level Mechanism for Continuous, Transparent Device Access During VM Migration. In: VEE, pp. 31–40 (March 2008)
18. Shea, R., Liu, J.: Performance of Virtual Machines Under Networked Denial of Service Attacks: Experiments and Analysis. IEEE Syst. J. 7(2), 335–345 (2013)
19. de Gooijer, J.G., Hyndman, R.J.: 25 years of time series forecasting. Int. J. Forecast. 22(3), 443–473 (2006)
20. Jo, C., Gustafsson, E., Son, J., Egger, B.: Efficient Live Migration of Virtual Machines Using Shared Storage. In: VEE, pp. 41–50 (May 2013)
21. Enwiki Dump Progress, `http://dumps.wikimedia.org/enwi-ki/`