Journal of Network and Computer Applications 34 (2011) 1088–1096

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

Optimizing the live migration of virtual machine by CPU scheduling

Hai Jin [a,*], Wei Gao [a], Song Wu [a,*], Xuanhua Shi [a], Xiaoxin Wu [b], Fan Zhou [a]

[a] Services Computing Technology and System Lab., Cluster and Grid Computing Lab., School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
[b] Intel China Lab., No. 2 Kexueyuan S. Road, Beijing 100191, China

ARTICLE INFO

Article history:
Received 20 November 2009
Received in revised form
30 May 2010
Accepted 7 June 2010
Available online 9 July 2010

Keywords:
Live virtual machine migration
Pre-copy algorithm
CPU scheduling
Dirty rate

ABSTRACT

Live migration has been proposed to reduce the downtime for migrated VMs by pre-copying the generated run-time memory state files from the original host to the migration destination host. However, if the rate for such a dirty memory generation is high, it may take a long time to accomplish live migration because a large amount of data needs to be transferred. In extreme cases when dirty memory generation rate is faster than pre-copy speed, live migration will fail. In this work we address the problem by designing an optimization scheme for live migration, under which according to pre-copy speed, the VCPU working frequency may be reduced so that at a certain phase of the pre-copy the remaining dirty memory can reach a desired small amount. The VM downtime during the migration can be limited. The scheme works for the scenario where the migrated application has a high memory writing speed, or the pre-copy speed is slow, e.g., due to low network bandwidth between the migration parties. The method improves migration liveness at the cost of application performance, and works for those applications for which interruption causes much more serious problems than quality deterioration. Compared to the original live migration, our experiments show that the optimized scheme can reduce up to 88% of application downtime with an acceptable overhead.

1. Introduction

Virtualization abstracts the resources such as CPU and memory through generating virtual machines (VM) to better support resource assignment (Rosenblum and Garfinkel, 2005). In modern data center (DC) or Cloud environment, virtualization has been considered to be by fault the basic resource management technology because through virtualization the resources can be easily consolidated, partitioned, and isolated. In particular, VM migration has been applied for flexible resource allocation or reallocation, by moving applications from one physical machine to another for stronger computation power, larger memory, fast communication capability, or energy savings.

Primary migration relies on process suspend and resume. This forces the migrated application to stop until all the memory states have been transferred to the migration destination where it is resumed, which severely undergrads user experience in particular for applications that cannot be interrupted. To reduce the migration downtime, live migration has been proposed (Nelson et al., 2005; Clark et al., 2005) where pre-copy mechanism (Theimer et al., 1985) has been used to support seamless process transfer. In each round of pre-copy, the original host machine copies the memory data of the VM that will be migrated, and sends the data to the receiving machine (i.e., destination host). In the mean time, the host records the changed bits of memory during that copy/send phase. These dirty bits are generated by the running applications in the VM and have to be delivered to the destination in next rounds. By repeatedly pre-copying these dirtied bits, if data delivery is faster than new dirty bit generation, after a number of rounds, there will be only a small amount of dirty memory left. The host machine then can suspend the applications running in the VM and deliver all the remaining dirty data to the destination. Because the amount of memory data to be delivered in this final round is small, the delivery takes little time before the VM (including the applications) can be resumed at the destination machine. The overall downtime of the VM or applications, including dirty data delivery time and process resume time, is short and may not be perceived by users.

The key to the success of a live migration is that at the last round of pre-copy the outstanding dirty memory size should be small. The speed of dirty memory generation then becomes the most important parameter that determines the performance of a live migration. If the dirty memory generation rate is high, at each round of pre-copy a large amount of new dirty memory is generated. This will lead to an increased number of rounds of pre-copies, and consequently much more data to be transferred between migration parties. In cases where dirty data generation rate is faster than memory copy, e.g., either because the application is memory writing intensive or the network between

* Corresponding authors.
E-mail addresses: hjin@hust.edu.cn, jinhust@gmail.com (H. Jin), wusong@hust.edu.cn (S. Wu), Xiaoxin.wu@intel.com (X. Wu).

1084-8045/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2010.06.013

the host and destination has a low bandwidth, a large amount of dirty memory will remain no matter how many rounds of pre-copy has been executed. This results in a relatively long time for the final round of pre-copy and normally causes perceivable interruption for the migrated applications. The live migration thus fails.

To mitigate the problem, in the work we propose to optimize live migration through reducing the rate of dirty memory generation (or dirty rate). This can be achieved by adjusting VCPU, i.e., the CPU allocated to the VM during live migration, to control the memory writing speed. When the VM writes memory too fast, the VCPU will be slowed down. The amount of newly generated dirty bits during each round of pre-copy can be reduced, and the overall data to be transferred at the final round of pre-copy can be controlled under a desired value. Thus a technique not only can reduce the interruption time of a migrated application, but also can reduce the overall live migration time because the overall memory to be delivered between migration parties will be reduced as well.

Reducing dirty memory by trading off computation power does result in application performance degradation. Our targeted applications are those of which moderate performance degradation is tolerable, while interruption may cause much more serious performance issues. Typical applications that may apply the proposed live migration optimization technique are visual applications, e.g., game application. Slowing down VCPU to reduce rendering frame rate can only impact visual results for while. However, if the game has to be interrupted for a relative long time, e.g., hundreds of millisecond due to migration, the game results may change.

Our major contributions are summarized as follows:

- Based on the pre-copy for live migration we proposed mechanism to reduce the downtime of VM migration with memory intensive application or low network bandwidth condition, through slowing down the application memory writing.
- We provide analysis model for analysing key parameters that affect the performance of migration.
- We implemented a prototype in Xen environment. Experiments show that by making the memory's dirty rate in a proper range, this method can expand the pre-copy algorithm's applicable range and improve the performance of live migration in restricted condition. We provide experimental data for performance evaluation in real applications and explored trade-off of this algorithm with different setups.

This paper is organized as follows. After introducing related works and giving motivation of our work in Section 2, we describe the pre-copy model and analyse the factors that influence the live migration in Section 3. In Section 4, we propose our optimization method. The experiments and results are presented in Section 5. We conclude the paper in Section 6.

2. Related work and motivation

VM migration is a hot topic of computing system virtualization. Many systems (Sapuntzakis et al., 2002; Kozuch and Satyanarayanan, 2002; Whitaker et al., 2004) just pause the VM and copy the state data, then resume the VM on the destination host. These methods cause the application to become unavailable during the migration process. ZAP (Osman et al., 2002) could achieve lower downtime of the service by just transferring a process group, but it still uses stop-and-copy strategy. To move the VM between hosts in local area network without disrupting it, VMotion (Nelson et al., 2005) and Xen (Clark et al., 2005) use pre-copy algorithm to perform live migration. Based on their works, Sapuntzakis et al. (2002) and Travostinoa et al. (2006) tried migrating running VM on a wide area network.

There are many approaches to improving the performance of service live migration. The most popular technique is Memory balloon (Waldspurger, 2002; Bradford et al., 2007), which could eliminate unused memory (reduce the total transferring memory M) to save time for the first pre-copy round. Hash-Based Compression (Sapuntzakis et al., 2002) sends the hash of memory page before transfer the memory data; if the same page exists on destination host, this page needn't be copied. Opportunistic replay (Surie et al., 2008) minimizes the overhead of VM migration by recording user actions and replaying them on the destination host. Clark et al. (2005) mentioned some other approaches, such as Dynamic Rate-Limiting and Rapid Page Dirtying, to improve live migrating VMs on Xen. However, most of them just passively adapt to the memory writing behaviour, except Stunning Rogue Processes, which is an active approach that delays the execution of processes that write memory is too fast. But it needs to set up a 'stub handler' into the OS running service, and it is too simple to deal with complicated environments. The same strategy was used in (Bradford et al., 2007) for transferring persistent data.

Keeping service running without much performance loss while the VM is being moved to another physical machine was considered as a primary goal to achieve for live migration. However, many applications are more sensitive to the execution breaking than efficiency. For example, a web server would rather reduce respond latency than lose connections already built on; desktop end users or online game players may feel more comfort with a slow experience than with a frozen one. This paper tries to discover the key factors and limits of the live migration process and then to optimize its performance and available range. We focus on the downtime of migration process under severe environments that the traditional pre-copy algorithm does not work well.

Our solution uses a similar mechanism as the process-pausing methods, whereas our methods do that at the hypervisor level, out of VM, and give more dynamic strategy based on the status of VM and the migration environment. Though scheduling the whole VM's execution is not accurate as doing it to the process, the reasons why we chose not to implement our strategy inside the VM are:

- We want to avoid modifying the software environment in the VM, which may impact application's performance and constrain the optimizing method into the VMs pre-installed specific program.
- The VM is generally used for isolating independent applications. That means a VM's running processes are tight coupling. When a memory-over-writing process is paused, the other processes running in the same VM waiting for which would slow down the whole application's performance. So compared to our solution, the benefit of handling processes is not significant in practice.

3. Pre-copy model

The basic idea of iterative pre-copy algorithm is, by transferring dirty data repeatedly, changed memory could be decreased round by round, which then eventually reaches an acceptable size to move quickly, so that the downtime of service is negligible. As a mainstream algorithm used for live migration, pre-copy has been described in many papers before, yet there is no formal definition

given. This section presents the basic pre-copy model, and then analyses the parameters that impact the live migration's performance.

We firstly define parameter M as the total memory of the VM, and parameter B as the network bandwidth assigned to the migration task. To describe the speed of memory changing in iteration i, we introduce a parameter p_i as the dirty rate of round i. Then the time for transferring memory in round i can be calculated from every dirty rate of previous iterations according to the following equation:

$$t_i = \frac{p_{i-1} \times t_{i-1}}{B} = \frac{M \times \Pi_{k=1}^{i-1} p_k}{B^i} \tag{1}$$

When the dirty data of a pre-copy iteration is small enough or too many rounds have been done, the migration process transfers the remaining changed memory for the last time. In the final round, VM is stopped and its memory is not writable to ensure that no more data have to be sent. We define the number of all iterations n as the following equation:

$$n = \min(\{i+1 | p_i \times t_i \le h\}, N) \tag{2}$$

where h is the threshold value to begin the last round, and N the maximum number of iterations.

To describe the performance of the migration, there are two important parameters usually referred to. One is the total migration time that counts the time lasting from the beginning of the migration process to the end of it; the other one is downtime, which represents the period that service is pausing before it is resumed at the destination host. The total migration time indicates the general performance of migration, while in some circumstances users may prefer the shortest downtime with tolerant overhead.

From Eq. (1) we can see that the pre-copy mechanism works if $p_i < B$ so the time spent to send dirtied memory could be reduced during every round. When one round's dirty rate exceeds the transfer speed, the time spent for next pre-copy iteration will be longer than this one's. To analyse the relations between each parameters and the performance of the whole process, we sum the transfer time of all rounds as the following equation:

$$T_c = M \times \sum_{i=1}^{n} \frac{\Pi_{k=1}^{i-1} p_k}{B^i} \tag{3}$$

This equation shows the total pre-copying time T_c would be longer while the migrated VM has more total memory and/or less available network bandwidth. When M QUOTE QUOTE and B are fixed, T_c could be decreased by slowing down the dirty rates of every transfer round. For a given p_k, when it is increased, the copy time of all rounds after current one will be extended. In other words, the dirty rates of earlier rounds affect more rounds than the latter ones.

As for the downtime, since t_n depends on the ratios of dirty rates to assigned bandwidth, p_k/B, to make the downtime negligible, it is expected that most of the dirty rates are small enough, so that remained changed memory reach the threshold h in Eq. (3) within the $N-1$ rounds pre-copy, then the time spent for last round could be limited to be no more than h/B. Apparently an iteration would get close to this goal only if its ratio p_k/B is less than 1, the iterations with ratio equal or more than 1 would be wasted, then postpone or even draw back the progress. If the dirtied memory left by the $N-1$ round of pre-copy stays over the threshold h, it means that this algorithm does not work well, and the last iteration has to transfer them suspending more time. In the worst case, when the memory writing speed is too fast to reduce the dirtied pages during all iterations, the pre-copy algorithm is totally failed, and the last round will take M/B downtime to finish the process as well as a non-live migration.

The actual data transferred during the migration process is usually more than the total memory of VM because of the dirtied part. We use R as the redundancy ratio to describe the trade-off of the pre-copy algorithm and it could be calculated by the following equation:

$$R = \frac{T_c \times B}{M} = \sum_{i=1}^{n} \frac{\Pi_{k=1}^{i-1} p_k}{B^{i-1}} \tag{4}$$

here R indicates the extended transferred memory when using pre-copy. Bigger the ratio is, more overhead caused by iteratively pre-copy dirtied memory.

Memory dirty rate is a key parameter to the performance of pre-copy mechanism, especially when the bandwidth used for VM migration is poor. Theoretically, when the VM's memory writing speed is stable, say $p_1 = p_2 = \ldots = p_i = p$, then the dirty rate p must be less than B to make dirtied memory decreasing. Furthermore, to ensure the downtime of migration acceptable, p needs to be

$$p \le \sqrt[N-1]{h/M} \times B \tag{5}$$

so that the dirtied memory can be reduced to h before the Nth pre-copy iteration. Generally the pre-copy algorithm set $N=30$ and $h=256$ kB. Fig. 1 shows the boundary of p/B that makes pre-copy effective with different total memory size M. In most of the circumstances, p should be less than 80% of B to achieve short enough downtime. To explain how dirty rate p affects the downtime when it exceeds that boundary, we calculated downtimes of a virtual machine with different memory writing speeds. The conditions are set as an 800 MB memory VM migrated on a 200 Mbit/s network, and the results are given in Fig. 2. We can see that when p reaches 151.4 Mbit/s, which is the biggest dirty rate that the pre-copy algorithm works well with, the downtime t_n starts its exponential increase. We call this downtime's fast growing phenomenon the "migration barrier", it constraints the available range of live migration. When p is raised to as fast as the network bandwidth B, the pre-copy is totally failed and causes downtime equal to the time that transfers all the memory.

Though pre-copy algorithm keeps VM running during its live migration, the performance of applications in it will be undermined somehow. To quantitatively analyse the pre-copy algorithm's trade-off, we introduce a parameter F representing the numbers of instructions executed by CPU per second, it is a variable only relative to the main frequency of processor.

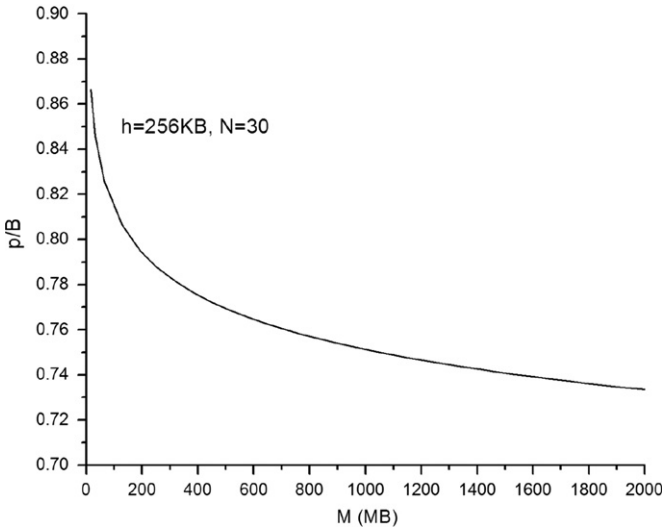

Fig. 1. Effective boundary of p/B with different memory sizes M.

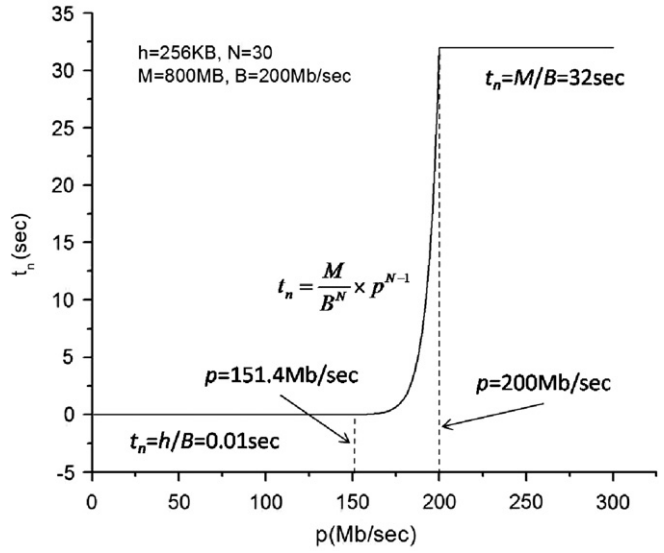

Fig. 2. Downtime t_n of migrations as the dirty rate p changes.

The VM's executed instructions during migration are represented as follows:

$$I = F \times U \times T \tag{6}$$

where U is VM's average CPU usage and T the total time of migration. This parameter I can be used to compare the two migration methods' impacts on the performance of application running in VM. Suppose the destination host uses the same CPU as the source host, the VM's efficiency difference of two migration methods could be

$$I_L = \left[U_1 \times T_1 - U_2 \times T_2 - U_{des} \times (T_1 - T_2)\right] \times F \tag{7}$$

where U_1, U_2, T_1 and T_2 are, respectively, the CPU usage and migration time of methods 1 and 2, and U_{des} represents the migration's destination host's available CPU usage. Specially, when $T_2=0$, the I_L is the efficiency loss of the migration 1.

4. Optimizing live migration

Based on above analysis, we propose an optimized iterative pre-copy algorithm, which could limit the dirty rate of VM, thus help to improve the performance of live migration process.

The main idea of the algorithm is, when a VM writes memory too fast to perform the pre-copy, it schedules the CPU time for this VM to a proper percentage, so that the dirty rate could be adjusted to a small enough value, then according to conclusions in Section 3, the downtime and the total time of the pre-copy process could be reduced to acceptable ranges.

4.1. Mechanism design

Our solution is based on an experiential rule that a certain VM's memory dirty rate is approximately linear increase with the growing speed of the VM's execution by host CPU. Assuming a VM running programs, which have x percent of the instructions to write the memory, and the VM is writing y MB data per second to the memory. When we set the CPU timeslice occupied by the same VM from 100% to 50%, because the percentage x does not change, the VM will write the memory with the speed of $y/2$ MB/s. In fact, there are other factors that affect the relationship between dirty rate and execution speed, such as process priority. Because the operation system would prefer to schedule the processes with higher priority (kernel processes or others), many processes with normal priority will not be executed when there is not enough CPU timeslice left. That means the dirty rate of high priority memory writing process will decrease slower than the low priority one if VM's execution speed keeps reducing. Nevertheless, such processes exist in the application programs with same lower priority, so in most situations, we can consider that approximately linear relationship is reasonable.

We use e_i and e_{i+1} as the percentages of processor quantum allocated for a migrated VM before and after optimization, p_i and p_{i+1} as the dirty rates generated by e_i and e_{i+1} execution. The relationship among those parameters would be

$$\frac{e_i}{e_{i+1}} = \frac{p_i}{p_{i+1}} \tag{8}$$

When e_i and p_i are known, and if we want to adjust p_{i+1}/B to a given value C, e_{i+1} should be

$$e_{i+1} = \frac{C \times B \times e_i}{p_i} \tag{9}$$

The parameter C is an expectant ratio that determines the proper dirty rate to achieve good downtime in migration process with certain bandwidth. As Eq. (5) and Fig. 1 implied, C is preferred to the effective boundary of p_i/B, which can be calculated from known parameters (VM's memory size, available network bandwidth and pre-copy algorithm's last round threshold). However, since C is one of the key parameters in our algorithm that determines the target VM's running speed and the memory writing speed, its value should balance those two factors. A larger C leads to less efficiency loss of application while may not help to a successful live migration; and if C is too low, the performance of application will be unbearable even the VM is moved quickly. Though the actual dirty rate will vary during the migration, most of the applications running in VM have memory writing patterns. For example, some applications keep dirty rates in narrow ranges, some write the memory with stable frequency. In order to guarantee p_{i+1}/B stays in the area close to C, we use the dirty rate of current round as reference to determine the next round's quantum to be allocated. Our algorithm will update the dirty rate at the end of each pre-copy round and re-calculate e_{i+1} by replacing pi with new dirty rate.

As the result of Eq. (9), e_{i+1} is a value relative to the current CPU scheduling percentage e_i. The original proportion allocated to the migrated VM should be saved on first pre-copy round and restored at the end of migration before the VM resumed on destination host.

4.2. Optimized pre-copy algorithm

We modified the basic pre-copy algorithm to which adjusts certain timeslices for the VM being migrated in each pre-copy round. Algorithm 1 presents the optimized pre-copy algorithm. It needs max round limit N and threshold of doing last round H as input parameters. For simplicity here set the expectant ratio C as a constant. The main steps of the algorithm are as follows:

At first, the data to be transferred M is initialized to the VM's whole memory and the round index R is initialized to 1. Lines 1 and 2 complete this initialization.

Line 3 starts the main loop of the algorithm that iterative pre-copy VM's dirtied memory, it continues until dirtied memory reaches the threshold or the next round is the last one. Lines 4–6 perform the traditional pre-copy algorithm that save changed memory and count the time lasting in current round while sending the previous round's dirtied memory. After calculating the transfer speed of last round in line 7, line 8 moves the content of *TempM* to M to copy in the next iteration, and line 9 gets the

dirty rate of current round. Then we do our optimization on lines 10–12. After acquiring currently assigned scheduling parameter, the algorithm calculates the new execution percentage according to Eq. (9). Line 12 resets the new value to the parameter of VMM's scheduling strategy to make prospective percentage of physical CPU time available to the appointed VM. At the end of the iteration, add 1 to the round counter R on line 13.

In the end, lines 15 and 16 send the remained memory in the last stop-and-copy round. Then finish the process by resuming VM on line 17.

Algorithm 1. OptimizedPreCopy(*VM*, *N*, *H*)

1: $M \leftarrow VM$'s memory
2: $R \leftarrow 1$
3: While size of $M > H$ and $R < N$ do
4: Transfer M to destination host
5: Record *VM*'s dirtied memory to *TempM*
6: Count the transferring time T
7: $B \leftarrow M/T$
8: $M \leftarrow TempM$
9: $P \leftarrow M/T$
10: Read current CPU time allocated for *VM* to E
11: $E \leftarrow C^*B^*E/P$
12: Schedule E% of the CPU time to execute *VM*
13: $R \leftarrow R+1$
14: End while
15: Pause *VM*
16: Transfer M to destination host
17: Resume *VM* on destination host

Our algorithm only introduces a scheduling strategy to help improve the performance of VM live migration; it could cooperate with most of the other known optimization methods compatibly. One shortcoming of the algorithm is that it works by sacrificing the performance of application running in VM, which makes user uncomfortable. We will discuss this problem in Section 5. In fact, this algorithm will be the last attempt behind other approaches. It effects only if the previous methods failed to achieve the expected downtime.

4.3. CPU scheduling implementation

We chose the open source Xen virtual machine monitor (Barham et al., 2003) and its live migration tool (Clark et al., 2005) as basic environment to implement the above-optimized pre-copy algorithm on it. Since Xen is an open source virtualization platform, it is easy to get every round's needed values and calculate the target scheduling percentage. We inserted several codes into the "xc_domain_save.c" file, which does most of the work on VM live migration. Fig. 3 shows the main part of our improvement to the original pre-copy algorithm.

The SetSched() function is given in Fig. 4, which controls the execution of target VM's CPU. VM monitor attains global load

```
if (ts_dirty_rate > 0) {
    if (ts_send_rate > MAX_MBIT_RATE) {
        ts_send_rate = MAX_MBIT_RATE;
    }
    ts_setValue = ts_send_rate * TS_C_VALUE / ts_dirty_rate;
    SetSched(ts_setValue);
}
```

Fig. 3. The code that schedules CPU in each iteration. ts_send_rate and ts_dirty_rate are the memory data transfer and changing speed of current round, parameter C is assigned by the constant TS_C_VALUE.

```
void SetSched(int i) {
    int rInt, paramIndex;
    virSchedParameterPtr paramUnit;

    /* Initialize sched */
    if (i < 1) i = 1;
    ts_sched = ts_sched * i / 100;
    if (ts_sched < TS_MIN_SCHED) ts_sched = TS_MIN_SCHED;
    if (ts_sched > TS_MAX_SCHED) ts_sched = TS_MAX_SCHED;

    /* Set Sched Parameters */
    if (ts_params != NULL) {
        for (paramIndex=0; paramIndex<*ts_nParams; paramIndex++) {
            paramUnit = ts_params + paramIndex;

            if (!strcmp(paramUnit->field, "cap")) {
                paramUnit->value.ui = (unsigned int)ts_sched;
            }
        }

        rInt = virDomainSetSchedulerParameters(ts_dom, ts_params, *ts_nParams);
        if (rInt != 0) {
            SchedError();
            return;
        }

        paramUnit = NULL;
    }
    else {
        if (ts_fp != NULL) fprintf(ts_fp, "[Sched Task] params is NULL\n");
    }
}
```

Fig. 4. The SetSched() function—it sets the percentage of target VM's CPU execution by invoking libvirt API, the Credit Scheduleris used.

balancing on all the managed VMs by performing CPU scheduling strategies. Our solution only works on the specified VM when it is migrating, so it is not necessary to substitute the existing scheduler or undermine it by significantly impacting other VMs' running. Also, to avoid the VM running too slowly to keep the service response, we set the CPU time allocated to the VM not be lower than 20%.

Xen provides three optional CPU schedulers: Simple Earliest Deadline First (SEDF) (Leslie et al., 1996), Borrowed Virtual Time (BVT) (Duda and Cheriton, 1999) and latest Credit Scheduler. They all assign each VM a separate weight representing the proportion of CPU share. Weight is a relative value, which means the actual CPU allocated to a VM is decided by its weight compared to that of the other VMs. The scheduler will allocate more CPU time to the one with higher weight. So the migrated VM's executing speed could be controlled by adjusting its weight as the following equation:

$$w_0 = \frac{e}{1-e} \times \sum_{i=1}^{n} w_i \tag{10}$$

where e is the expectant percentage of CPU time allocated to the VM and $w_1, \ldots, w_n$ are the weights of VMs running on host except the one being migrated. When one VM's weight changed, the whole host's CPU distribution will be reset, even other VMs keep the weight unaltered. To implement absolute VM executing speed, we should modify w_0 whenever there is variation with any VM's weight.

Besides the weight-based scheduling, the SEDF and Credit Scheduler provide another CPU allocation mechanism named non-work-conserving mode (Cherkasova et al., 2007). It supports limiting each VM's CPU usage, which makes our scheduling avoid complicated weight operation. Just assign the value e to the target VM's cap parameter, then the CPU time allocated to the VM will not exceed e percent. By using cap mechanism, the target VM's weight need not be frequently calculated and modified; however, when migration begins, the current CPU scheduler in use must be detected for choosing the right method to control VM execution.

5. Evaluation

The basic pre-copy has been proven to work well on VM with normal workloads in the environments that have sufficient resources. In this section we describe the experiments undertaken to evaluate the total time and downtime of the optimized pre-copy algorithm in tough circumstances. We also discuss the trade-off of our approach and compare the performance to the basic pre-copy algorithm.

5.1. Experimental setup

We built our experimental environment on a pair of two-socket servers, each socket has 4 Intel Xeon 1.6 GHz CPUs. Both servers were 4 GB DDR RAM and connected by a 1000 Mbit/s Ethernet network. We used Linux 2.6.18 with Xen 3.1.0 installed as the operation system. The Xen was default set Credit as the CPU scheduler. Storage was exported to the migrated VM from a file system image, which was accessed via the NFS protocol. We also pre-installed Red Hat Enterprise Linux 5 as guest OS in VMs.

5.2. Performance comparison

Firstly, we compare the general performance between original and optimized pre-copy algorithms. We used a program, memWriter, to generate workloads during our experiments. This program writes the given-sized memory with a stable and controllable speed. We performed a series of migrations with the bandwidth limited to 1000, 500, and 200 Mbit/s. Workloads write 800 MB memory with different speeds; the range is from 41 Mbit/s (naked guest OS) up to 4798 Mbit/s. Every workload is migrated 5 times, by separately using original Xen live migration and our optimized live migrations with the parameter C assigned to 0.6, 0.7, 0.8, and 0.9 for comparison. In all cases the VM was configured to have a single CPU with 1 GB of RAM.

Fig. 5 shows the downtimes of both the original migration and the optimized migration with different memory writing speeds. The two methods perform well when memory dirty rates are less than the memory transfer speed that most of the downtimes are kept in 0.1 s. As the memory dirty rate increases close to the available network bandwidth, original live migration makes VM downtime rapidly rise and then stays at a very high position while the dirty rate keeps increasing. This is in accord with the migration barrier mentioned in Section 3. It can be caused by the limit of network bandwidth assigned for the migration process. When a traditional live migration meets the migration barrier, the pre-copy algorithm fails to reduce the changed memory size and the migration process has to send the whole memory in stop-and-copy iteration. This phenomenon also follows the model presented in Section 3.

Compared to the original pre-copy algorithm, the optimized pre-copy performs better. All the optimized migrations loosen the migration barrier and keep downtime of migration short enough while the memory writing speed even exceeds the data transfer bandwidth. The extended effective range of the optimization is affected by the parameter C. As it decreased, the migration barrier moves more. Because of the failure of pre-copying or optimized pre-copying, the experiments with dirty rates exceed their migration barriers and achieve similar worse performance.

When the 1 Gbit/s network bandwidth is assigned to the migration process and the expectant ratio C is set to 0.9, the optimized algorithm extends the traditional pre-copy mechanism's migration barrier; here it is represented by the first dirty rate that makes downtime reach 1 s, from nearly 600 bit/s to about 1.2 Gbit/s. In the best situation (dirty rate=1943 Mbit/s, C=0.6), the optimized approach reduces the downtime to 0.026 s, only 0.4% of the original migration's 6.435 s. As the network bandwidth is limited to 500 and 200 Mbit/s, the migration barriers of traditional mechanism correspondingly shrank under the available network bandwidths. Due to the reduced memory transfer speed, in situations that pre-copy does not work, the largest downtimes are higher than those under 1 Gbit/s transfer speed.

From Fig. 6 we can see that under 1 Gbit/s available network bandwidth, optimized mechanism with C=0.6 uses least time to perform the whole VM migration than other pre-copy processes when memory writing speed does not exceed the migration barrier. While the (C=0.9) optimized migration makes longest total time, even higher than the original process. As dirty rates

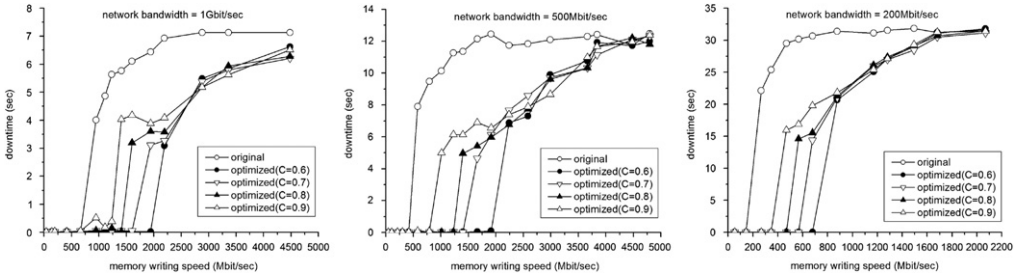

Fig. 5. The down times of original and optimized migrations for different workloads.

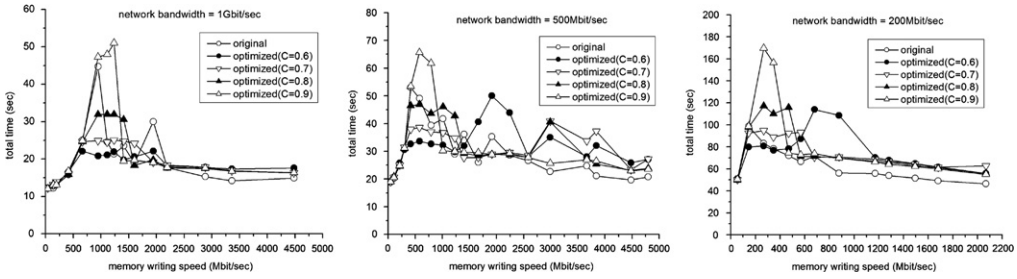

Fig. 6. The total times of original and optimized migrations for different workloads.

rise up to more than the migration barrier, the optimized methods do not work well, leading the total migration time to rise, while the original mechanism detects that the dirty rate is too large and finishes migration by skipping early iterations and directly going into stop-and-copy round.

In conditions that 500 and 200 Mbit/s network bandwidths are allocated to the process, results follow a similar way, except a peak of total time for the ($C=0.6$) optimized migration close to the migration barrier, while others are more smooth and stable.

5.3. Dynamic web application

To evaluate the effect of optimized pre-copy algorithm in practical circumstances, we tried both approaches on a dynamic web application. A client sends 100 requests per second to access the '/jsp-example/dates/date.jsp' page provided by tomcat running in the migrated VM. The VM is configured with 8 CPUs and 512 MB of RAM. We limit the network bandwidth for migration to 200 Mbit/s, and set the parameter C to 0.8.

Fig. 7 illustrates the detailed migration process of the experiment. The x-axis indicates time elapsed since the start of migration, while the y-axis shows the speed of memory changing (illustrated by black box) or sending (illustrated by box in write grid). We also put a gray line into this figure for showing the CPU cap assigned during the process. The VM has been in three phases through the migration. Phase 1 was the earlier 2 rounds lasting about 38 s. In this long phase, the memory is transferred with a low speed of 104 Mbit/s, and the CPU allocation is not changed yet. After the memory dirty rate reaches 160 Mbit/s in the 2nd round, the optimization starts to adjust the VM's processor quantum to 52%. Phase 2 includes rounds 3–8 between the 39th second and the 43rd second that CPU scheduling strategy is working. The dirty rate is slowed down to 135 Mbit/s in round 3, while the pre-copy mechanism raises the transfer rate up to 209 Mbit/s. Though the distance of two rates makes the CPU cap set back to 62% in the next round, the allocated processor quantum of each round is eventually reduced rapidly in the following few rounds, as well as decrease in dirtied memory, leading to performing the last phase. In phase 3 the CPU scheduling is terminated and the cap parameter is set back to the initial state. The VM is stopped at round 9, and then does the last iteration to finish the migration. The whole process takes total 43.986 s, and downtime is shrunk to only 288 ms.

We also moved the same workload without using optimization. It spends 42.968 s total time and 2491 ms downtime. The throughputs achieved by both mechanisms are shown in Fig. 8. During the most of the migration times, the optimized migration does not sacrifice the efficiency of application. The throughput is reduced by CPU scheduling strategy for no more than 4 s. As a conclusion, we can say that comparing those two processes, though spends total time about 1 more second, using our optimized pre-copy algorithm reduces downtime by over 2.2 s, 88% of the original algorithm.

5.4. Overhead analysis

We have mentioned that a side-effect of our optimized algorithm is, when the CPU time allocated to the migrated VM is constrained, the applications running in this VM would be slowed down at the same time. It seems that the algorithm obtains the small downtime by sacrificing performance of service. Now we discuss this problem based on our experimental results.

Fig. 9 shows the VM's average CPU usages of every migration in all the previous experiments on memWriter workloads. When the original migration is performed, the VM uses as much CPU as possible to execute workload, while our algorithm schedules the VM's CPU allocation as the memory writing speed increases. To draw the growing dirty rates back to the value below the available network bandwidth, the processor quanta for their workloads are gradually decreased. Both mechanisms make stable CPU usages

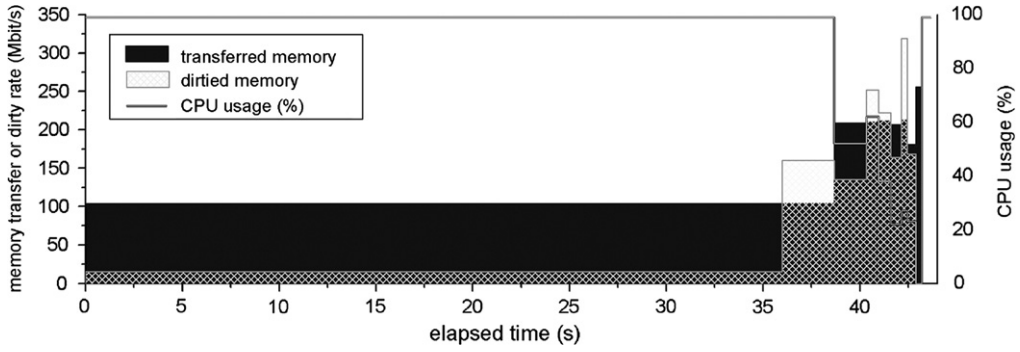

Fig. 7. Results of migration a VM running tomcat application.

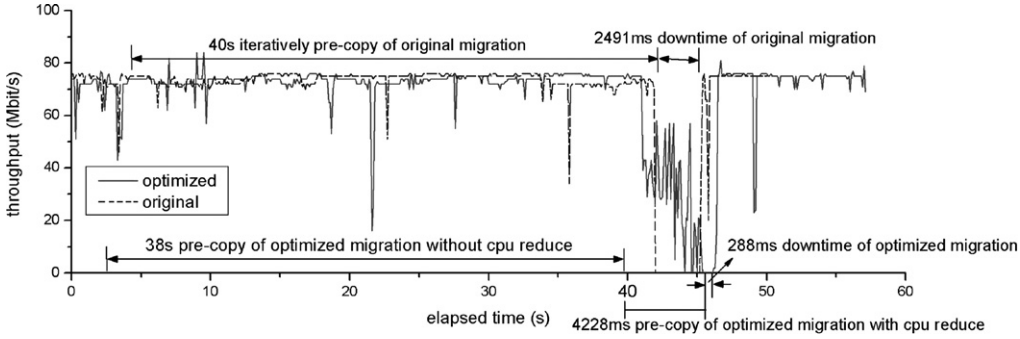

Fig. 8. Throughput comparison of original and optimized migration for tomcat application.

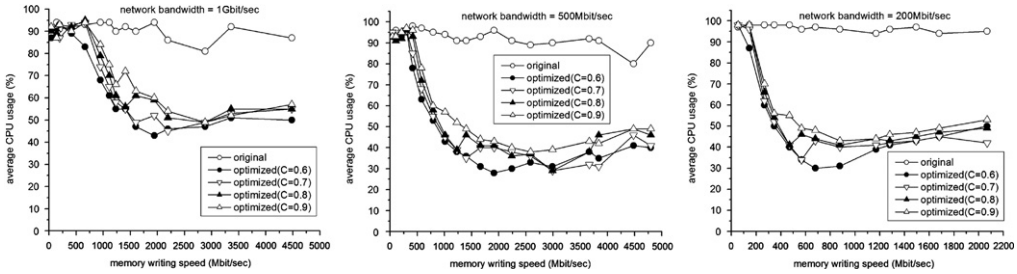

Fig. 9. The average CPU usages of original and optimized migrations for different workloads.

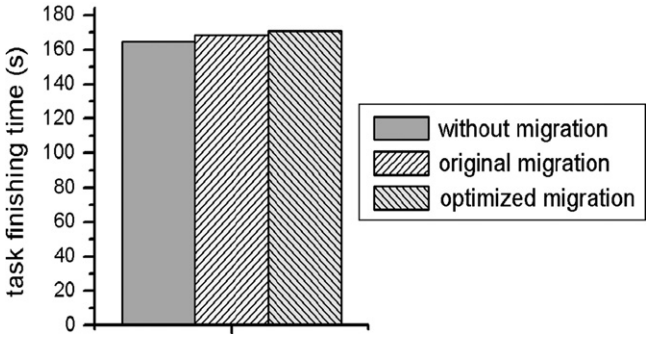

Fig. 10. Results of 800,000 web app. requests task.

when they meet their own migration barriers. We can also see that the difference in average CPU usage among different C parameters is not too distinct. Though the target VM's CPU frequency is reduced, the released timeslices are not wasted as they are allocated to other running VMs on the same host by Credit Scheduler.

To evaluate the general performance overhead of original and optimized migrations, we designed a task that sends a total of 800,000 access requests to the previous dynamical web application.

From Eq. (7) and the experiment results collected in the last subsection, we can calculate the efficiency loss of the optimized algorithm as

$$I_L(\text{mig-tomcat}) = 0.99 \times 42.968 - 0.94 \times 43.986 - 0.99 \times (42.968 - 43.986)F = 2.2F$$

It means that the optimized algorithm will spend $I_L(\text{mig-tomcat})/(U_{des}F) = 2.22$ s more than the original migration.

The results of finishing time for the task performed with different situations are given in Fig. 10. The whole task takes 165 s when it is performed without migration, while there is another 3.5 s time spent on original migration, and 6 s spent on optimized migration as well. Since both the migrations spend no more than 44 s, we can get the overheads as the efficiency lost rates; hence, the original algorithm makes 8% overhead while the optimized algorithm makes 6% more efficiency loss.

Though would make extra efficiency lost to the migrated VM, our solution could avoid large downtime. In fact our mechanism could be explained as it divides the application's unavailable time due to the last stop-and-copy iteration into tiny execution intervals and spreads them over the whole live migration process. Besides, by taking some timeslices from the migrating VM, other VMs can acquire more CPU resource, which improves the overall CPU utility. So it is a choice between slow service with short break and fast service with long break. It was thought that live migration should prefer to ensure the service's performance, but when service does not respond the requests for a long time, even losing the connections, we should consider balancing its downtime and efficiency with the approach proposed in this paper.

Another by-effect of our approach is that no matter how many processes write the memory, all processes in the VM will be punished, which is unfair to those innocent processes, and loses efficiency of the applications in the VM as a whole. However, for isolating different applications from each other, users will normally create VMs as many as they needed, and each VM contains one application. If the application's processes are tight coupling, then the benefit of just punishing one process is not very significant. So as mentioned in our motivation, it is acceptable to gain our optimization's scalability on different guest OS by sacrificing this rare potential efficiency loss.

6. Conclusions

In this paper, we presented the basic pre-copy model of VM live migration and proposed an optimized algorithm to improve the performance of live migration. Iterative pre-copy mechanism is sensitive to the VM's memory writing speed; when the dirty rate exceeds the transferring bandwidth, the downtime meets the migration barrier, which it will enlarge quickly. Our approach tries to limit the speed of changing memory through controlling the CPU scheduler of the VM monitor. We described the design and implementation of our optimized pre-copy algorithm in details. VMs with different memory writing speeds have been migrated in our experiments. Results show that by using the optimized algorithm, the migration barrier has been loosened up to 4 times. Comparing the migrations of the same workload with and without optimization, our solution can dramatically lower the VM's downtime, with the acceptable overhead.

In the future, we plan to integrate more parameters, such as writable working set (Clark et al., 2005), into the model presented in this paper. We are going to add network bandwidth controlling and memory writing patterns to our optimized pre-copy algorithm, using many strategies together to allocate proper resources to perform VM live migration.

Acknowledgments

This work was supported by National 973 Key Basic Research Program under Grant no. 2007CB310900, Program for New Century Excellent Talents in University under Grant NCET-07-0334, NSFC under Grant no. 60973037, Information Technology Foundation of MOE and Intel under Grant MOE-INTEL-09-03, Important National Science & Technology Specific Projects under Grant 2009ZX03004-002.

References

Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, et al. Xen and the art of virtualization. In: Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19). ACM Press; 2003. p. 164–77.

Bradford R, Kotsovinos E, Feldmann A, Schioberg H. Live wide-area migration of virtual machines including local persistent state. In: Proceedings of the 3rd international conference on virtual execution environments (VEE), 2007. p. 169–79.

Clark C, Fraser K, Hand S, Hanseny J-G, July E, Limpach C, et al. Live migration of virtual machines. In: Proceedings of the 2nd ACM/USENIX symposium on networked systems design and implementation (NSDI), 2005.

Credit Scheduler. 〈http://wiki.xensource.com/xenwiki/CreditScheduler〉.

Cherkasova L, Gupta D, Vahdat A. Comparison of the three CPU Schedulers in Xen. ACM SIGMETRICS Performance Evaluation Review (PER) 2007;35(2):42–51.

Duda K-J, Cheriton D-R. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In: Proceedings of the 17th ACM symposium on operating systems principles (SOSP), 1999. p. 261–76.

Kozuch M, Satyanarayanan M. Internet suspend/resume. In: Proceedings of the IEEE workshop on mobile computing systems and applications, 2002. p. 40–6.

Leslie I-M, Mcauley D, Black R, Roscoe T, Barham P-T, Evers D, et al. The design and implementation of an operating system to support distributed multimedia applications. IEEE Journal of Selected Areas in Communications, 1996.

Nelson M, Lim B-H, Hutchins H. Fast transparent migration for virtual machines. In: Proceedings of USENIX '05, 2005.

Osman S, Subhraveti D, Su G, Nieh J. The design and implementation of zap: a system for migrating computing environments. In: Proceedings 5th USENIX symposium on operating systems design and implementation (OSDI), December 2002. p. 361–76.

Rosenblum M, Garfinkel T. Virtual machine monitors current technology and future trends. IEEE Computer 2005:39–47.

Surie A, Cavilla A-L, Lara E-D, Satyanarayanan M. Low-bandwidth VM migration via opportunistic replay. In: The 9th workshop on mobile computing systems and applications (HotMobile), 2008. p. 74–9.

Sapuntzakis C-P, Chandra R, Pfaff B, Chow J, Lam M-S, Rosenblum M. Optimizing the migration of virtual computers. In: Proceedings of the 5th symposium on operating systems design and implementation (OSDI), December 2002. p. 377–90.

Theimer M, Lantz K, Cheriton D. Preemptable remote execution facilities for the V-system. In: Proceedings of the 10th symposium on operating system principles, December 1985. p. 2–12.

Travostinoa F, Daspit P, Gommans L, Jog C, Laat C-D, Mambretti J, et al. Seamless live migration of virtual machines over the MAN/WAN. Future Generation Computer Systems 2006;22(8):901–7.

VMware Infrastructure 3 Documentation, 〈http://www.vmware.com/support/pubs/〉.

Waldspurger C-A. Memory resource management in VMware ESX server. In: Proceedings of the 5th symposium on operating systems design and implementation (OSDI), ACM operating systems review, 2002, Special issue. p. 181–94.

Whitaker A, Cox R-S, Shaw M, Gribble S-D. Constructing services with interposable virtual hardware. In: Proceedings of the 1st symposium on networked systems design and implementation (NSDI), 2004. p. 169–82.