2010 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010)
IPCSIT vol. 53 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V53.No.2.61

Application of A Disk Migration Module in Virtual Machine live Migration

Song Wei[+] and Gao Bao

Department of Information Science and Technology, Beijing Forestry University

Beijing, China

Abstract—In the paper, we first analyse the live migration of Kernel based Virtual Machine (KVM), and then implement a Linux virtual disk driver called DM (Disk Migration). During the live migration, KVM will make use of the DM module to migrate the hard disk, so KVM live migration is feasible when there is no shared disk between the source and the destination. The DM module uses "bitmap" algorithm to record disk changes, and uses "request-reply" algorithm to safely migrate the dirty disk blocks, and deploys "pre-copy" algorithm to reduce the downtime of KVM live migration. The experiments show that the downtime of live migration is less than 100ms, which means these algorithms the DM uses can effectively limit the interrupt time of service.

Keywords-live migration; virtual disk driver; KVM; DM

1. Introduction

Nowadays, several types of virtual machine monitors support the live migration, such as Xen, VMware, KVM and so on, the virtual machine is preferred to be used to realize the live migration for the reason that migrating an entire operating system and all of its applications as one unit allows us to avoid lots of dif┐ culties faced by deploying process-level migration approaches[1].

Most research focuses on migrating only memory and CPU states of the virtual machine assuming that the source and the destination machine can share disk storages. So this kind of live migration can not be used under the circumstance where disk storage sharing is impractical. In order to make the live migration feasible under these circumstances, we propose and implement a Linux virtual disk driver called DM, which is realized as a kernel module. The DM will migrate the whole disk data from the source machine to the destination machine by using the "bitmap" algorithm to track dirty disk blocks during the live migration and using the "pre-copy" algorithm to shorten the interrupt service time, which is also called the downtime, and using the "request-reply" algorithm to migrate dirty disk blocks and deal with the problem of data loss when transmitting on the network. Therefore, when there are no shared disks, the virtual machine monitor can lively migrate the guest OS which means the virtual machine to the peer virtual machine monitor by using this DM module.

2. KVM Live Migration

The virtual machine live migration refers to transferring run time data of a guest OS from source machine to destination machine. After the live migration, the guest OS continues to run on the destination virtual machine monitor. The live migration is a process during which the guest OS seems running all the time due to the short downtime and provides almost ceaseless services to the users [2].

[+] Corresponding author.
E-mail address: beijing@gmail.com

The live migration of KVM also focuses on transferring memory and CPU states of the guest OS, it can not be applied in non-shared disks environment. In KVM, a new execution mode called "guest mode" is added into Linux, joining the existing kernel mode and user mode. The kernel allocates continuous virtual address to form the guest address space where the guest OS is running. So the major job of live migration is to migrate the guest address space from the source to the destination. KVM uses the "pre-copy" algorithm to lively migrate the guest OS.

1) The "pre-copy" step: in this step, KVM will iteratively migrate memory and CPU states from the source to the destination. The "iteratively" means that migrating occurs in rounds, in which the pages to be transferred during round n are those that are modified during round n-1. The iterative process will continue until a convergent point that only leaves a small amount of memory and CPU states needed to transfer.

2) The "stop and copy" step: this step will stop the source KVM machine and then transfer the remaining memory and CPU states to the destination KVM.

3) The "resume" step: since all necessary data of the guest OS has been transferred to the destination, the destination KVM machine can be resumed to provide services for the users.

3. Related Research On Live Migration With Disk Storage

Kozuch proposes the "on demand fetching" way[4], which first migrates memory and CPU states and delays disk storage migration. The VM immediately resumes on the destination after the memory and CPU states migration is completed. It then fetches storage data on demand over the network. One advantage of this way is that the downtime is as short as the downtime of live migration with shared disk storage. But it will incur residual dependence on source machine, the availability of this way will be restricted because the migrated guest OS depends on two machines for quite a long time.

Bradford and Kotsovino propose the way of pre-copying the control information regarding dirty blocks to the destination and fetching dirty blocks whenever needed on the destination machine[5]. This way first migrates the whole disk storage to the destination one time and then migrates the memory of guest OS. During the disk migration process, the system uses the way of "back-end driver" to track all disk write access requests and to record the information concerning the write requests into a special structure including the written data, the location of the data and the length of the data and then to transfer the structure to the destination machine. After the source machine completes the data migration, the destination machine will be resumed but all the write accesses must be blocked before all forwarded structures are applied. So one drawback of this way is that it may cause a long I/O block time for the synchronization

Zhang and Luo design the TPM (Three Phase Migration) to migrate the disk storage[2]. The TPM algorithm is proposed to minimize the downtime caused by migrating disk storage data. A block bitmap is introduced to track all write accesses to the local disk storage during the disk migration process, and then the system will conduct the migration of dirty blocks according to the the block bitmap. An incremental migration algorithm is applied to facilitate the migration back to initial source machine with less migrated data and shorter migration time. But this system does not take into account of the problem of data loss on the network transmission.

Lv and Li design a virtual disk driver named DiskMig which has "write-limit" and "read-limit" migration modes to guarantee the disk consistence between the source and the destination[6]. The DiskMig uses "write-limit" mode to record the information of source dirty blocks into a well-designed bitmap structure. When the guest OS is migrated to the destination, the destination machine will use "read-limit" mode to check whether the disk blocks involved in a read operation are related to the bitmap structure or not. If it is not, the read operation is a normal read operation; if it is, it means that this read operation is associated with dirty disk blocks, and then the system will use the way of "Transfer On Demand with Forward Ahead"(TOD&FA) algorithm to synchronize the dirty blocks quickly. But this system also fails to provide a secure migration policy for dealing with data loss problem.

4. System Flow

The whole system flow is illustrated in the Figure 1. The DM module is responsible for migrating the disk data, and the KVM uses signals to communicate with the DM module. At the same time, KVM is in charge of migrateing memory data and CPU states of guest OS.

1) Begin to migrate guest OS, source KVM sends a signal to the DM. When the DM receives the signal, it will start to migrate the disk.

2) KVM uses the "pre-copy" algorithm to iteratively migrate the memory data and data regarding CPU states of guest OS until dirty memory pages are small enough. When the changed pages are very small, KVM stops migrating the memory and jumps out the loop, and goes to the next step.

3) KVM checks whether it receives a signal from the DM module or not. As mentioned before, the DM module starts to migrate the disk data when it receives a signal from KVM. The DM module will iteratively migrate the disk data until dirty disk blocks are small enough, and then the DM module will send KVM a signal in order to notify KVM of its completion. Hence if KVM does not receive a signal, which means the DM module is in the process of migrating disk blocks and has not finished the designed task, and then the system will return to the process 2. But when there is a signal, the system will go to the next step.

4) Stop the source guest OS and send a signal to DM to migrate the remaining dirty disk blocks.

5) KVM starts to migrate the remaining dirty memory pages. When it is completed, the system will go to the next step.

6) KVM checks whether it receives a signal from the DM module or not. If it does, the system goes to the next step. If it does not, KVM checks it again.

7) The source KVM sends the destination KVM a signal to resume the running of the guest OS on the destination KVM for all memory pages and the disk blocks have been completely migrated to the destination machine.

Both step 3 and step 6 are associated with checking the receipt of a DM signal, but there are some differences between these steps. In the step 3, when there is no signal, KVM chooses to continuously send dirty memory pages. However, in the step 6, KVM repeatedly checks the receipt of the signal when no signal is found, the difference is because guest OS has been stopped before the step 6 and no dirty memory pages will be created again.

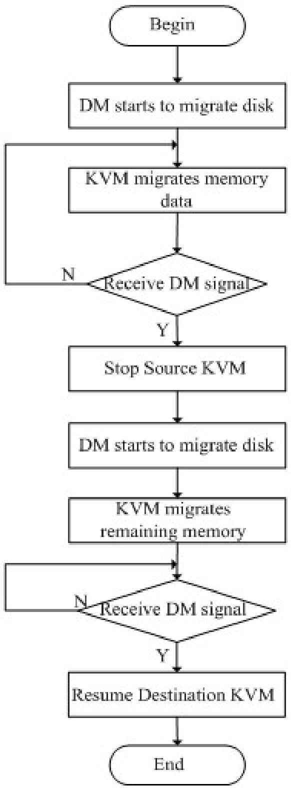

Fig 1. System Flow

5. DM Module Algorithm

The DM module is implemented as a Linux kernel module for the consideration of the efficiency of disk migration. As a linux kernel module, the DM module can get rid of the costs of copying data between kernel space and user space. The DM module lays at the bottom of the VFS and disk caches layers, and lays at the top of generic block driver layer. Therefore, the DM can make use of the APIs exported by the generic block driver layer to read and write the disk blocks.

5.1. Pre-copy algorithm

In order to provide the seamless services for users, the downtime of live migration should be considered as the first priority and should be manipulated as short as possible. The DM module uses the "pre-copy" algorithm, which is similar to the algorithm used in KVM memory migration, to shorten the downtime of the live migration. The "pre-copy" algorithm means that the DM module will migrate the whole disk storage to the destination DM module at the first time and then migrate the dirty disk blocks again and again until the dirty disk blocks are very small. After the guest OS is stopped, the DM is going to migrate the remaining disk blocks. Because the number of dirty blocks are so small at the time of stopping guest OS that these dirty blocks can be migrated to the destination in a very short time.

5.2. Bitmap algorithm

Since the guest OS is running when the source DM module migrates the disk blocks, a number of disk blocks may be modified during this time. So these dirty blocks must be syncronized in order to keep the migrated guest OS consistent. The DM module employs the way of "bitmap" to trace the dirty disk blocks. In the DM module implementation, it uses 1 bit in the bitmap to represent 4KB disk data. When some disk blocks are modified, those corresponding bits in the bitmap must be sat to 1. The DM module can find these dirty blocks by searching positive bits which mean the value of bit is 1 in the bitmap.

The structure block_bitmap is used to describe and manage the bitmap in DM module.

```
struct block_bitmap {
      unsigned long  *bp;
      spinlock_t     lock;
      unsigned long  bit_offset;
      unsigned long  total_set;
      unsigned long  total_bits;
}block_bitmap;
```

In the structure, the "bp" field points to the start address of bitmap data in the memory, this memory space is created through dynamic memory allocation and its size is decided by the size of the related disk device. And the "lock" field is a spin lock which is introduced to solve the problem of accessing the bitmap concurrently. The "bit_offset" field indicates the current position when searching the next positive bit. The "total_set" field represents the number of positive bits. The "total_bits" indicates the total number of bits.

1) set_bit(block_bitmap,sector, size,value)

This function sets values of related bits to "vaule" parameter, which can be 0 or 1. The positions of those bits can be calculated by the start sector and the size of the data, which can be specified separately by the parameter "sector" and parameter "size".

2) find_next_bit(bitmap)

This function is used to find the next positive bit from the current position in the bitmap. And the current position is indicated by the field "bit_offset" in the bitmap description structure.

5.3. Request-reply algorithm

As mentioned before, the "pre-copy" algorithm will repeatedly execute several times. In each time the DM module will use the "request-reply" algorithm to migrate the dirty disk blocks. This algorithm is illustrated in the Figure 2.

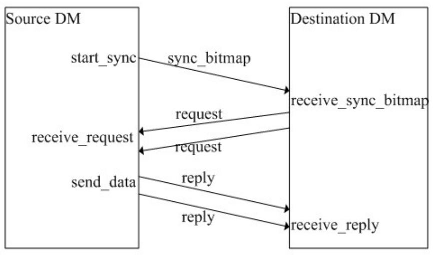

Fig 2. request-reply algorithm

- The source DM calls "start_sync" function to send a package, which has a "sync_bitmap" message header and a data body including the bitmap data, to the destination DM.
- The receiver thread of the destination DM will receive this package and call "receive_sync_bitmap" function to receive the bitmap data and execute the union operation with the local bitmap. The union operation is helpful to decrease the possibility of data loss, which will be explained later.
- In the "receive_sync_bitmap" function, it will create block request packages according to the positive bits in the new bitmap and send those request packages to the source DM, each request will ask for 4KB disk data. Searching the bitmap, the function will encapsulate and send one request package for each positive bit found in the bitmap.
- The receiver thread of the source DM module is responsible for receiving block request packages from the destination DM. When a new request package is received, the receiver process will call the "receive_request" function to deal with it, this function is going to read specific disk blocks into memory according to the information encapsulated in the request package.
- The sender thread of the source DM calls "send_data" function to send memory data which contains the data of disk blocks to the destination DM by encapsulating the data into a reply package, each reply package contains 4KB disk data.
- The receiver thread of the destination DM is in charge of receiving those reply packages. It calls "receive_reply" function to receive those disk data and write those data into relating blocks of the local disk. And then this function will clear corresponding bits in the local bitmap.

In the step 2, we mention the union operation, which is introduced to solve the problem of data loss on the network transmission. When a request or reply package is missed, the bit related to the package will not be cleared in the destination bitmap. So the value of bitmap after one round migration indicates which blocks fail to be synchronized in the last round. The union operation can force the DM module to re-send these lost request packages in the next round. Therefore, this union operation can help the disks to keep consistent after the live migration.

Due to there are lots of request packages sent from the destination DM, the DM module uses two threads instead of a single one to complete the migration of disk data. A receiver thread is used to deal with the requests and read disk blocks into memory by calling the "receive_request" function. A sender thread is introduced to migrate these memory data to the destination DM. So the receiver and sender threads can work concurrently. It is much more efficiently than the way in which a single thread processes reading disk data and sending disk data asynchronously. The communication between these two different threads is conducted by a queue, which stores disk blocks information and memory information. When the receiver thread finishes the read operation, it will append those information to the queue. If there is one or more than one elements in the queue, the sender thread will dequeue the element from the head of the queue and execute the send operation.

5.4. DM module architecture

According to the separate functions, DM module can be divided into 4 parts, as illustrated in the Figure 3. The local disk handle module is referenced by other modules, it can provide basic disk I/O services to other modules. As mentioned before, the data send module and the request handle module are designed as threads.

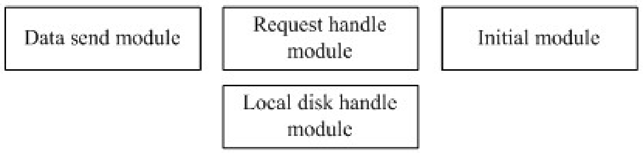

Fig 3. DM architecture

1) Initial module: this module configures the DM module based on the config files. It will build the network connection with the peer DM module and create 2 different threads, which are responsible for sending disk data and handling requests separately.

2) Local disk handle module: this module is responsible for reading and writing disk blocks. It provides some basic APIs for other modules to use. One of these basic APIs is "make_request" function, which uses the function
"generic_make_request" provided by the generic block device driver to complete the disk I/O operations.

```
make_request_common(bio, sector, size, rw)
{
    /*
      Initialize object "bio" using parameters provided
      in the function, such as "sector", "size" and "rw".
    */
    ......
    generic_make_request(bio);
}
```

The parameter "bio" is the object of structure bio, which is defined in the generic disk block driver and is used in several common APIs in this driver.

3) Request handle module: this module is created as a thread, which waits for the coming request packages. This module has to analyse the header of the package, and then call the message handler functions. In the DM, this receiver module can deal with 3 different kinds of packages: "sync_bitmap", "request" and "reply" packages. And the corresponding handler functions are "receive_sync_bitmap", "receive_request" and "receive_reply".

4) Data send module: this module is also created as a thread, which will send the data to the peer DM module. Its main API is "send_data". In this function, it will encapsulate a package, which has a header with "reply" message and disk data, and send this "reply" package to the peer DM module using socket.

6. Result and Data

In the experiment, we use two computers and they share the same configuration, which is Intel Core 2 Duo P8700 2.53GHZ CPU, 2GB Memory size and IDE disk. And they are connected by a Gigabit LAN. The software configuration is almost the same: host operating system is Ubuntu 9.04 Desktop Edition, the version of KVM is kvm-84, guest OS is Windows XP, the memory size of guest OS is 512MB and the disk size of guest OS is about 10GB.

We first measure the performance of KVM live migration based on share disks. KVM uses NBD to share disks with peer guest OS, so KVM live migration only migrates the memory data. The results are showed in the table I.

TABLE I. RESULTS FOR DIFFERENT OPERATIONS BASED ON NBD

	No operation	**Playing movie**
Time of pre-copy(ms)	1533.85	1756.49
Size of migrated data/MB	139.89	177.88
Downtime(ms)	2.80	1.99

When no disks can be shared between those two machines, KVM uses the DM module to migrate the disk data and makes the live migration available under this circumstance. The results are showed in the table II.

TABLE II. RESULTS FOR DIFFERENT OPERATIONS BASED ON DM

	No operation	**Playing movie**
Time of pre-copy(s)	192.85	195.02

Size of Migrated data/MB	10239.78	10240.06
Downtime(ms)	75.91	77.53

To keep the disk consistent after the live migration, the DM module deploys "request-reply" and "pre-copy" algorithms. Those algorithms can increase the total number of migrated data and the "pre-copy" time because lots of control messages and dirty blocks have been sent several times in order to make the disk consistent and the ensure the downtime short. Under some circumstances, the correctness of disk blocks and the shortness of downtime should be taken as the first priority, and it is worth increasing the "pre-copy" time a little to achieve this goal.

The DM module can effectively control the downtime by using the "pre-copy" algorithm. For example, if the DM module can set the threshold of dirty blocks to M, the estimate value of live migration downtime can be calculated by the equation (1) :

$$T = M / S. \tag{1}$$

T means the downtime and S means the speed of disk migration. And the value of S is mainly determined by the speed of network data transfer rates and that of disk I/O.

7. Conclusion and Future Work

In the paper, we design a Linux virtual disk driver named DM, which can migrate disk data to the peer DM module. So KVM can use this DM module to migrate the guest OS when there are no shared disks between the source machine and the destination machine. In the DM module, we deploy "bitmap", "request-reply" and "pre-copy" algorithms to effectively control live migration downtime and keep the disk data consistent. My future work focus on optimizing the KVM live migration and cutting down the time of "pre-copy", and reducing the total time of KVM live migration.

8. References

[1] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I.Pratt, and A.Warfield. Live Migration of Virtual Machines[J]. NSDI, 2005.

[2] Yingwei Luo, Binbin Zhang, etc. Live and Incremental Whole-System Migration of Virtual Machines Using Block-Bitmap[J]. In Proceedings of the 2008 IEEE International Conference on Cluster Computing, pages 99-106.

[3] A. Kivity, Y. Kamay, D. Laor, etc. KVM theLinux virtual machine monitor[J]. In OLS ’07: The 2007 Ottawa Linux Symposium, pages 225–230, July 2007.

[4] M Kozuch, M Satyanarayanan, T Bressoud, CHelfrich, S Sinnamohideen. Seamless mobile computing on fixed infrastructure[J]. IEEE Computer, 2004, 32(7):65-72.

[5] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, etc. Live Wide-Area Migration of Virtual Machines Including Local Persistent State. In Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments(VEE '07).

[6] Xiaolu Lv, Qin Li. Whole System Live Migration Mechanism for Virtual Machines[J]. Computer Science, 2009,7:256-261.