FITDOC: Fast Virtual Machines Checkpointing with Delta Memory Compression

Yunjie Du, Xuanhua Shi, Hai Jin, Song Wu
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, Chia
xhshi@hust.edu.cn

***Abstract*—Virtualization provides the function of saving the whole execution environment status of the running virtual machine (VM), which makes checkpointing flexible and practical for HPC servers or data center servers. However, the system-level checkpointing needs to save a large number of data to the disk. Moreover, the overhead grows linearly with the increasing size of virtual machine memory, which leads to disk I/O consumption disaster along with poor system scalability. To target this, we propose a novel fast VMs checkpointing approach, named *Fast Incremental checkpoinTing with Delta memOry Compression* (FITDOC). By studying the run-time memory characteristics of different workloads, FITDOC counts the dirty pages in a fine-granularity manner (the number of 8 bytes), instead of the conventional method (the number of pages). FITDOC utilizes dirty page logging mechanism to record the dirty pages; accordingly, a delta memory compression mechanism is implemented to eliminate redundant memory data in checkpointing files. To locate the dirty data in dirty pages, FITDOC utilize two mechanisms: by analyzing the distribution characteristics of dirty pages in dirty bitmap, we propose a fast dirty bitmap scanning method to locate the dirty pages, and take a multi-threading data comparison mechanism to locate the real dirty data in one page. The experimental results show that compared with Xen's default system-level checkpointing algorithm, FITDOC can reduce 70.54% of checkpointing time on average with 1GB memory size and achieve better improvement for VMs with larger memory configurations. FITDOC can reduce 52.88% of the checkpointing data size on average compared with Remus's incremental solution which is in page granularity. Compared with default dirty bitmap scanning method in Xen, the scanning time of FITDOC is decreased by 91.13% on average.**

Keywords-checkpointing; memory compression; dirty page; dirty bitmap

I. Introduction

More and more clusters and data centers use virtualization technology [1, 2] to run multiple OS instances concurrently on a single physical machine. The virtualization technology has brought lots of advantages, such as enhancing resources utilization ratio [3], facilitating management, reducing costs and energy consumption [4], improving service performance and availability [5, 6].

As an important feature of virtualization, checkpointing [7-11] is able to record the whole system state of a VM into a file for later restoration and ensure that the VM continues to run after completion. The file is generally saved in a shared file system for recovery on other physical machine. Just like migration, checkpointing is transparent to the OS and upper applications in VM. It provides great benefits for system maintenance, fault-tolerant, testing and debugging in modern clusters and data centers.

During checkpointing, the *virtual machine monitor* (VMM) stores the whole system-level state of a VM to non-volatile storage, including the state of VCPU, memory and all emulated devices. The memory state is stored in page granularity and usually equals to the memory size of the target VM. With the increasing growth of VM's memory configuration, checkpointing becomes more and more of a concern, which takes a long time and seriously affects the VM's QoS. For latency-sensitive applications, the long downtime even causes services failure [12].

In this paper, a novel checkpointing approach is proposed, named *Fast Incremental checkpoinTing with Delta memOry Compression* (FITDOC), to improve VM checkpointing performance. FITDOC utilizes the data distribution characteristics of the run-time memory in a fine-granularity manner, uses dirty page logging mechanism to record the dirty pages and introduces delta memory compression to eliminate redundant memory data in checkpointing files. To locate the dirty data in dirty pages, FITDOC utilizes two mechanisms: by analyzing the distribution characteristics of dirty pages in dirty bitmap, we propose a fast dirty bitmap scanning method to locate the dirty pages, and take a multi-threading data comparison mechanism to locate the real dirty data in one page. We carry out the implementation and evaluation of FITDOC based on Xen-4.0.1.

We summarize the main contributions of this paper as follows:

- We quantitatively analyze the data distribution characteristics of dirty pages and variation characteristics of dirty page numbers with time and memory size under different types of workloads, and checkpoint the dirty pages in a fine-granularity manner (the number of 8 bytes), instead of the conventional method (the number of pages) to reduce the size of the checkpointing files.
- We analyze the distribution characteristics of dirty pages of VMs and propose a quick scanning

978-1-4799-7981-3/14 $31.00 © 2014 IEEE
DOI 10.1109/CSE.2014.82

IEEE computer society

method of VM's dirty bitmap to locate the dirty pages, which effectively decreases the scanning time.

- We have designed and implemented a novel checkpointing approach with Xen-4.0.1 which introduces dirty page logging mechanism and delta memory compression into checkpointing to eliminate redundant memory data, which greatly reduces total checkpointing time.

The rest of this paper is organized as follows: Section II presents the background and quantitative analysis of run-time memory. The design and implementation of our system is described in section III. In section IV we present and analyze the experimental results. We introduce the related work in section V. Finally, section VI concludes our research and outlines the future work.

II. Background and Analysis

To study the run-time memory characteristics of the VM checkpointing, we select several representative applications as the workloads of target VM:

1) **underload**: An idle Linux Server.
2) **dbench**: dbench 4.0 [13] is an open source benchmark generating I/O workloads to file system.
3) **kernel compile (K-Compile** for short**)**: Linux Kernel Compile [14] is a balanced workload that stresses mainly CPU and memory, but doing a fair amount of disk I/O as well. We compile Linux kernel version 2.6.31.8 with default configuration.
4) **NPB**: *NAS Parallel Benchmarks* (NPB) [15] is a collection of computationally intensive parallel applications performing various scientific computations. We chose the programming model OpenMP and application LU.B.
5) **Hadoop**: Hadoop [16] is an open-source implementation of the MapReduce distributed data analysis tool, an emerging framework often used in cloud computing environments. We use a single node that hosts the JobTracker and the TaskTracker to execute the well-known wordcount tasks.
6) **TPC-W**: TPC Benchmark W (TPC-W) [17] is a transactional web server and database performance benchmark. The workload is performed in a controlled internet commerce environment that simulates the activities of a business oriented transactional web server.

A. *System-level Checkpointing Time*

We conduct some experiments on a server-class machine to test the time to save a VM with different memory sizes and workloads respectively through NFS. The result is shown in Figure 1. In Figure 1, the X-axis represents the memory size of the target VM and the Y-axis represents the whole system-level checkpointing time.

As can be seen from Figure 1, the VM checkpointing time increases linearly with the increasing memory size, and almost has nothing to do with the workload within the VM. This is because the whole system-level checkpointing needs to record all the pages, regardless of whether the page is used or not.

TABLE I. The statistics of variation in byte granularity

	underload	Dbench	K-Compile	NPB-LU.B	Hadoop	tpcw
zero pages	1303	220807	2001	2389	13239	4157
[0]	261342	239183	234930	131122	94927	251882
(0,1]	23	40	93	16	900	176
(1,2]	22	289	65	26	1126	317
(2,4]	18	384	531	23	1606	573
(4,8]	59	588	253	54	1627	646
(8,16]	60	570	237	58	1316	558
(16,32]	76	1000	398	71	1024	426
(32,64]	41	887	415	43	593	383
(64,128]	37	1921	334	44	553	385
(128,256]	33	4821	200	55	859	446
(256,512]	85	5514	425	93	1119	515
(512,1024]	26	5080	903	208	2903	567
(1024,2048]	1	691	4258	1654	12432	1367
(2048,4096]	1	854	18186	128357	139486	3579
dirty pages	482	22639	26298	130702	165544	9938
total pages	261824	261822	261228	261824	260471	261820
dirty bytes	67660	11475425	78848380	450396692	536244749	12407963
8B dirty bytes	40888	4043960	68428456	108190080	416419512	3745856
bytes of dirty pages	1974272	92729344	107716608	535355392	678068224	40706048
bytes of total pages	1072431104	1072422912	1069989888	1072431104	1066889216	1072414720
dirty page rate	0.18%	8.65%	10.07%	49.92%	63.62%	3.80%
dirty bytes / bytes of dirty pages	3.43%	12.38%	73.20%	84.13%	80.03%	30.48%
dirty bytes / bytes of total pages	0.01%	1.07%	7.37%	42.00%	50.92%	1.16%
8B dirty bytes / dirty bytes	60.43%	35.24%	86.78%	24.02%	77.65%	30.19%

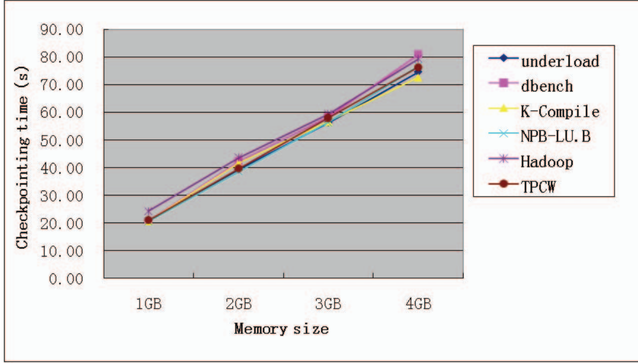

Figure 1. Time to save VMs with different memory sizes and workloads

In the case of a fixed bandwidth and disk I/O speeds, the checkpointing time T_c can be expressed as:

$$T_c = S/R \propto S_m \quad (1)$$

where S represents the total state size, S_m represents the memory size and R represents the data transmission rate. Therefore, the key to reduce checkpointing time is to reduce the memory state data.

B. Data Distribution Characteristics Study of Dirty Pages

Traditional checkpointing method saves memory state in pages, but how many bytes become dirty in a page over a period of time is another matter of concern. To measure this in the context of our benchmarks, we have run them while continuously taking 5 checkpoints with 3 minute intervals for 1GB VMs. Then we select one checkpoint and analyze the variation characteristics in byte granularity. The result is shown in Table I. In Table I, the variables are explained as follows:

zero pages: count of the pages that full of zero.

[0]: count of the pages that change nothing compared with previous checkpointing.

(x, y): count of the pages whose dirty bytes are in this range compared with previous checkpointing.

dirty pages: count of the pages whose dirty bytes are in (0, 4096).

total pages: count of the total pages, which is usually about 262144 (1GB/4KB) for 1GB VM.

dirty bytes: the total amount of changed bytes compared with previous checkpointing.

From statistics in Table I, we can get the following viewpoints: 1) Most of the memory pages have changed nothing, which follows the incremental checkpointing discussions in [25][26]; 2) There are many dirty pages which only change few bytes. The average dirtiness rates of dirty pages under our six workloads are 3.43%, 12.38%, 73.20%, 84.13%, 80.03%, and 30.48%, respectively; 3) As for the whole system-level checkpointing, the actual dirty bytes are only 0.01%, 1.07%, 7.37%, 42.00%, 50.92%, and 1.16%, respectively, of the total bytes of memory pages. So if we only save the state of dirty bytes, we will largely reduce the amount of data to be transferred, thus reducing the checkpointing time to a large extent.

As described above, there are many dirty pages that only change a few of bytes. So how the dirty bytes distribute in dirty pages is also a concern. In x86_64 computer, we can conduct 8 bytes data once. As shown in Table I, there are many dirty bytes changed in 8 bytes unit. The average 8B dirtiness rates of dirty bytes under our six workloads are 60.43%, 35.24%, 86.78%, 24.02%, 77.65%, and 30.19%, respectively.

III. System Design and Implementation

This section describes the design and implementation of FITDOC. To shorten additional compression time, we employ multi-threading technique to parallelize the compression tasks. Furthermore, we propose a quick scanning method of VM's dirty bitmap aiming at finding out the dirty pages. To demonstrate the feasibility of FITDOC, we implement a prototype based on Xen-4.0.1.

The framework of FITDOC is shown in Figure 2. During VM running, we enable the log dirty mode and record dirty pages through dirty bitmap. When we make a checkpoint, FITDOC needs to go through the following steps: 1) suspend the VM and the VM stops running; 2) get dirty bitmap from Xen address space through hypercall; 3) fast scan VM's dirty bitmap to find out dirty pages; 4) compress the dirty pages using delta memory compression and multi-threading technology, and then transfer the compressed data to NFS server; 5) resume the VM and make it continue to run in the consistent state before checkpointing.

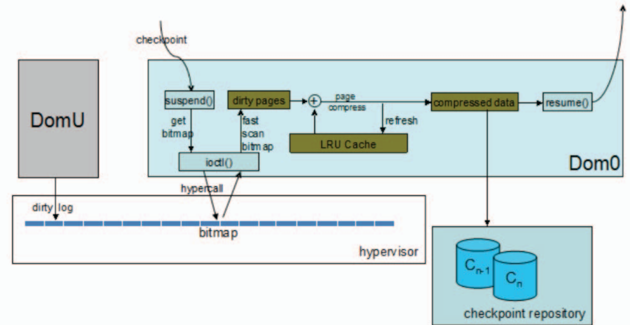

Figure 2. Framework of FITDOC

A. Dirty Page Logging Mechanism

Dirty page logging mechanism can record dirty pages using dirty bitmap and is firstly applied in live migration. The dirty bitmap in Xen address space maps the entire virtual address space of the VM in sequence and each bit corresponds to one of the VM's pages. At the beginning, all bits of dirty bitmap are set to 0, and all entries of the shadow page table are marked as read-only. When the guest operating system in VM performs a write memory operation, the VM will trap and give over the control rights to the VMM. Then the VMM will set the corresponding bit in dirty bitmap to 1 to mark the page as a dirty page, and give the corresponding entry in the shadow page table writable permission. Thus the VM will not trap when performing write operations on this page again. Log dirty mode can lead the VM to trap and will introduce some overhead, but this

overhead can be almost negligible because trap occurs only when the target page is write at the first time, as detailed in section IV.

B. Fast Scanning Dirty Bitmap

As described above, in order to find out the dirty pages, we need to scan the dirty bitmap firstly. In Xen environment, 32-bit VM has a 4GB virtual address space, i.e., a total of 1M (4GB/4KB) = 1048576 pages. Each page corresponds to one bit of the dirty bitmap in sequence, so we need 1048576 bits = 32 pages (1048576/8/4086) to save dirty bitmap, and thus each page saving dirty bitmap corresponds to a 128MB (4GB/32) virtual address space. Traditional method scans the dirty bitmap bit by bit to find out the bits whose value is 1 and then records the corresponding PFNs (*page frame number*), which usually takes several tens of milliseconds. It is worth noting that the VM is in stopped state during scanning, which would have a more serious impact on the performance.

Figure 3 illustrates the addressing mode in dirty bitmap, i.e., how PFN corresponds to its dirty bit. As shown in Figure 3, Xen uses four-level page table to index dirty bitmap. L4 ~ L3 entries store the MFNs (*machine frame number*) of the next level page table, and the L2 entries store the MFNs of the dirty bitmap pages. The L1 refers to the base address of the dirty bitmap page. Xen separates a PFN into four parts to index L4 ~ L1 respectively, 41 ~ 33 bits for L4, 32 ~ 24 bits for L3, 23 ~ 15 bit for L2 and low 14 ~ 0 bits for indexing the dirty bit where the PFN corresponds. Most important of all, whether the address in L2 entry is valid is an important basis for judging whether there are dirty pages distributed in the dirty bitmap page which corresponds to a 128MB virtual address space.

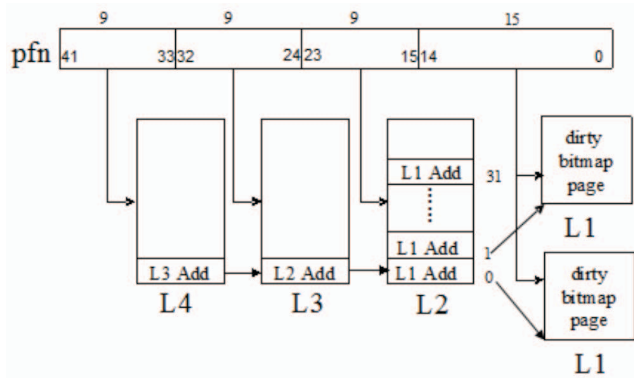

Figure 3. Addressing mode in dirty bitmap

In order to get dirty bitmap, Xen needs to scan 0 ~ 31 L2 entries in sequence to get the base address of dirty bitmap pages. If the address stored in the L2 entry is valid, then we can get a valid dirty bitmap page from the Xen address space; otherwise, we make a zero page to act as the dirty bitmap page. Finally, 32 valid or zero dirty bitmap pages compose a continuous dirty bitmap which maps the entire 4GB virtual address space. Traditional method scans the dirty bitmap bit by bit, which does not effectively take advantage of the characteristics of zero pages, and thus takes up a lot of redundant time.

How dirty pages distribute in the 4GB virtual address space is also a matter of concern. We record dirty pages for 317856 times with 1 second interval for a 512MB VM running kernel compilation to test the distribution of dirty pages using log dirty mode. Result shows that there are only 2 checkpoints where dirty pages distributed in the high 3.5GB (PFN>131072) space and that dirty pages of all the remaining checkpoints only distributed in the low 512MB (PFN⩽131072) space. That is to say, there are almost no dirty pages in the high 3.5GB space, and thus the corresponding dirty bitmap pages of high 3.5GB space are almost all zero pages. We also test other workloads, and the result is also the same. However, traditional method still scans these continuous zero pages bit by bit.

Taking advantage of above dirty pages distribution characteristics, we can speed up scanning dirty bitmap using flags. FITDOC sets up two level flags, L1 flags and L2 flags, in Xen address space for the entire 4GB VM virtual address space, as illustrated in Figure 4. Each L1 flag corresponds to a dirty bitmap page and thus corresponds to a 128MB virtual address space in sequence. Therefore, we require a total of 32 (4GB/128MB) L1 flags. In addition, because there are almost no dirty pages in the high 3.5GB space, we set up a L2 flag for this space. The two level flags are initialized to 0.

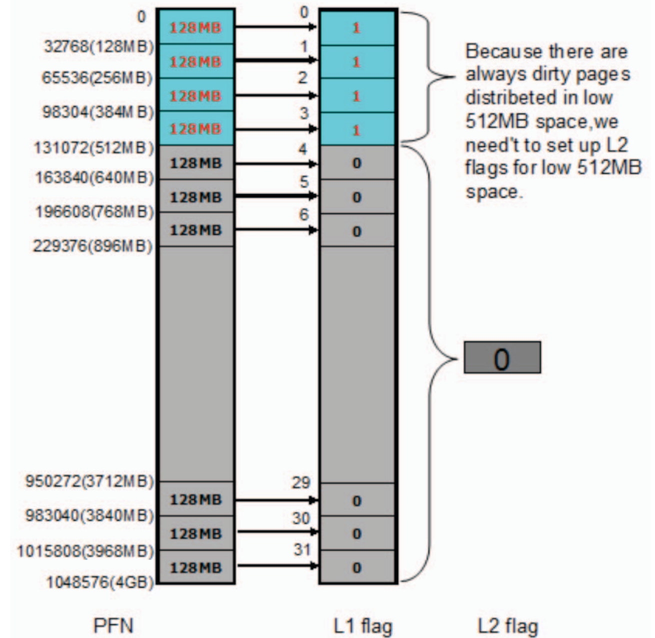

Figure 4. The mapping diagram from 4GB virtual address space to two level flags

The fast scanning method contains two stages: 1) setting the flags; 2) judging the flags. The former is located in Xen space. When application in Domain0 calls for dirty bitmap from Xen space through hypercall, we set the values of two level flags according to the validity of addresses in L2 entries, and then return the dirty bitmap together with the two level flags to the upper layer application. The latter is located in

Domain0. After the application get the dirty bitmap and two level flags from Xen space, we first judge the values of flags in sequence and then decide whether or not to scan the corresponding parts in dirty bitmap, and thus save a lot of time.

Since most of the flags are zero, we can skip the scan of the corresponding parts in the dirty bitmap, and therefore effectively save the scanning time. The result is shown in section IV.

C. Delta Memory Compression

Seen from Table I, there are a lot of dirty pages which only change a few bytes. But traditional checkpointing method needs to save the complete 4KB page data, which take up more time.

Delta memory compression provides an effective solution to address this problem. To compare with previous pages, we save the dirty pages of previous checkpoints into a memory buffer. According to the time locality principle of program, a previous dirty page will likely continue to be changed later, and therefore we adopt the LRU (*Least Recent Used*) page replacement algorithm. FITDOC borrows ideas from RLE [18] data compression method and improves it. The schematic diagram of delta memory compression is illustrated in Figure 5. We compare the changes between pre-page and post-page and get the dirty lines in 8B granularity (8B/line), then use line number to mark it.

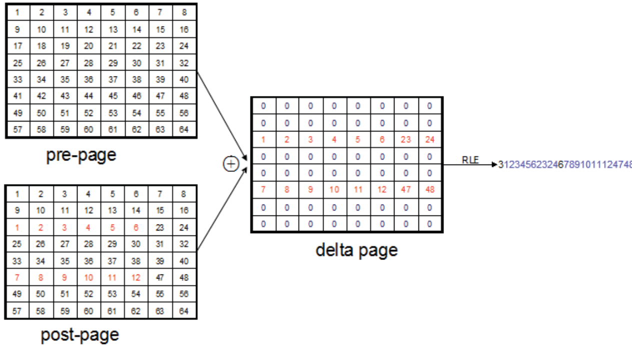

Figure 5. Schematic diagram of delta memory compression

The basic process is as follows:

First, for a target dirty page, we first judge the hit or miss in the memory buffer. If miss, we add the target dirty page into memory buffer according to the LRU page replacement algorithm, and then turn into processing next dirty page; otherwise, go to next step.

Second, we make a generalized XOR operation for the hitting page (pre-page) and the target dirty page (post-page). Since the dirty bytes in a page also distribute locally, as described in section II, we make the XOR operation in 8 bytes in X86_64 computers for speed up. Then we will get a sparse page which contains the delta information.

Finally, we record the indexes and data of non-zero units in the sparse page, which is named compressed data and then update the memory buffer.

Because of the independence among compression tasks on each page, we can use multi-threading technology to speed up memory compression. FITDOC adopts the classic producer/consumer model: producer thread is responsible for compressing target dirty pages and then putting the compressed data into a memory buffer; consumer thread is for reading compressed data from the buffer and then transferring them to NFS server. The experiments show that two producer threads and a consumer thread can achieve better compression performance.

IV. PERFORMANCE EVALUATION

In this section, we evaluate FITDOC with various workloads detailed in section II, then present and analyze performance improvement compared with Xen's default whole system-level checkpointing algorithm and Remus's incremental solution [8]. We primarily care about the impact on VM's performance using dirty page logging mechanism, the time spent on scanning dirty bitmap, the checkpointing time, the checkpointing size and the compression overhead.

A. Experimental Environment

Our experiment platform is composed by two identical computer servers. One server works as the storage server providing shared storage by NFS protocol to the other server, which acts as the physical machine running VMM. For each server, its configuration includes eight Intel Xeon E5620 quad-core CPUs running at 2.40GHz, 16GB DDR RAM. Two servers are connected by a Gigabit LAN. We use CentOS 5.4 as the guest OS (DomainU) and the host OS (Domain0). The version of VMM is Xen-4.0.1 and dom0 Linux kernel is 2.6.31.8. The unprivileged VM is configured with 2 VCPUs and 512MB RAM except where noted otherwise. Full-virtualized VMs are used in our tests.

B. Overhead of Log Dirty Mode

As described above, using log dirty mode would introduce some overhead. We make experiments to test the completion time of NPB-LU.B and kernel compile in the ON and OFF mode respectively, and the result is illustrated in Figure 6. Seen from the figure, the completion time in ON mode only increases 0.52% and 0.26% compared to OFF mode, that is, log dirty mode almost has no effect on the performance of the VM. This is mainly because that the VM will trap only when the target page is written at the first time, which will introduce some overhead, but subsequent write operations on the same target page will not lead the VM into a trap. Due to time locality principle of program, a target page generally tends to be written repeatedly, and therefore the overhead introduced by the first write on the target page can be almost negligible.

C. Scanning Time of Dirty Bitmap

Figure 7 illustrates the time to scan dirty bitmap under several different workloads. Experimental results show that compared with Xen's default scanning algorithm, FITDOC can reduce scanning time by 91.13%, 91.24%, 91.23%, 91.03%, 91.13%, 91.05%, and in average of 91.13%, which agrees with the statistics in section II. We can reduce more with small memory size. The substantial effect is caused by following reasons. Firstly, by using L1 flag, FITDOC only

scans the dirty bitmap page where dirty pages distribute. Second, there is generally almost no dirty page in the high 3.5GB space, we can usually skip to scan the dirty bitmap for this space using L2 flag. In addition, what we can also see from the figure is that the scanning time is almost independent of the workload ran in the VM.

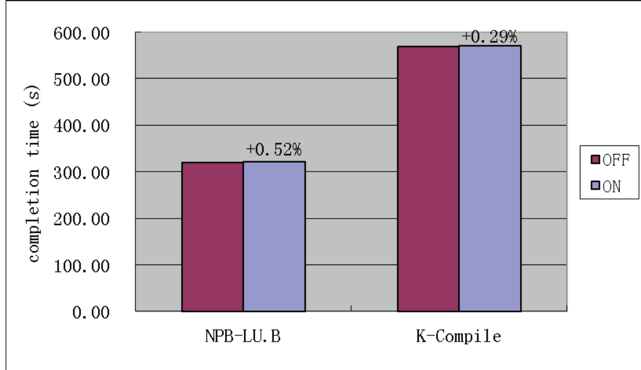

Figure 6. The impact on VM's performance using log dirty mode

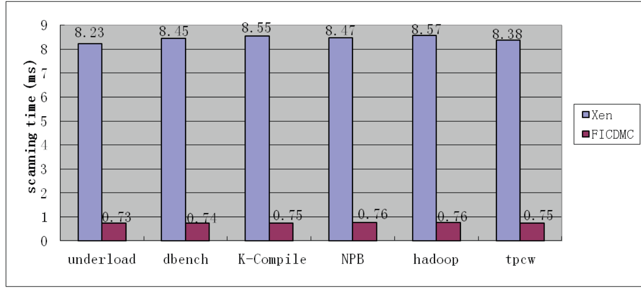

Figure 7. Time to scan dirty bitmap

D. *Checkpointing Time*

An important concern is the checkpointing time and Figure 8 illustrates it under different workloads and memory sizes. Experimental results show that compared to Xen's default checkpointing algorithm, FITDOC can reduce checkpointing time by 91.90%, 84.52%, 77.26%, 45.52%, 40.41%, 83.81%, and 70.54% on average with 1GB memory size, and the checkpointing time increases by only a small amount with the increasing memory size. This is mainly because: 1) different workloads have different *dirty page set* sizes. FITDOC only records the dirty pages using log dirty mode. 2) FITDOC only transfers the dirty bytes in dirty pages in 8B, which eliminates a large number of redundant state data. 3) The fast method to scan VM's dirty bitmap. 4) The *dirty page set* sizes do not change much despite the different configurations of memory size.

E. *Checkpointing Size*

Another important aspect is the checkpointing size and Figure 9 illustrates the size of one checkpointing under different workloads with 1GB memory size. Experimental results show that compared to Xen's system-level checkpointing algorithm, FITDOC can reduce checkpoint size by 99.99%, 98.93%, 92.63%, 58.00%, 49.74%, 98.84%, and in average of 83.02%. As we known, Remus [8] incremental solution only saves the dirty pages to eliminate redundant memory data and speed up checkpointing. But the dirty pages also contain lots of unchanged bytes, as detailed in section II. Experimental results show that compared to Remus's incremental checkpointing solution, FITDOC can reduce checkpoint size by 96.57%, 87.62%, 26.80%, 15.87%, 20.92%, 69.52%, and in average of 52.88%. This is because FITDOC saves the dirty data in a fine-granularity, while Remus works with the page granularity.

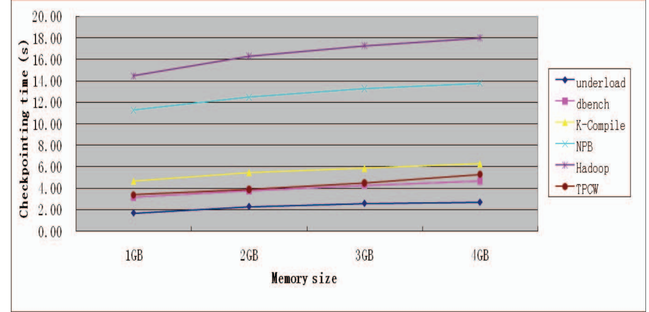

Figure 8. Checkpointing time under different workloads and memory sizes

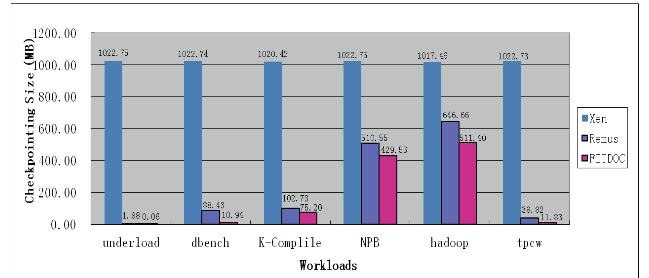

Figure 9. The size of one checkpointing under different workloads compared with Remus's incremental solution (Memory size: 1GB)

F. *Compression Overhead*

Compared to Xen's default checkpointing algorithm, the most overhead FITDOC brings is the compression overhead. Figure 10 illustrates the compression time under different workloads and memory sizes. Seen from Figure 10, with 1GB memory configuration, the compression time accounts for 34.71%, 32.19%, 33.40%, 38.23%, 35.66%, 31.18% respectively of the total checkpoint time, and in average of 34.23%. Because of the basically fixed *dirty page set*, the compression time also increases by only a small amount with the increasing memory size. Although memory compression brings additional compression time, it eliminates a large number of redundant state data that need to be transferred through network, and therefore greatly shorten the checkpointing time overall.

V. RELATED WORK

As important features of virtualization technology, physical memory related functionalities such as checkpointing, live migration and fault-tolerance have received a lot of attention in academia and industry.

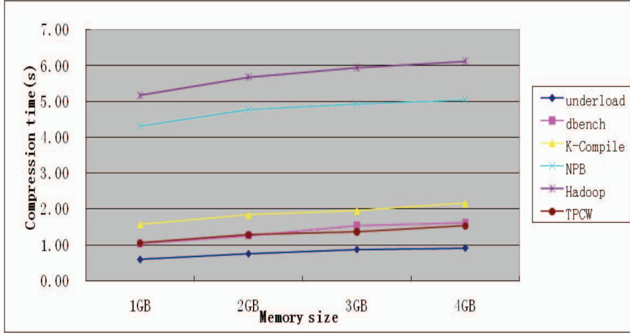

Figure 10. Compression time under different workloads and memory sizes

VM live migration [12, 19, 20] can migrate a running VM from one physical machine to another without stopping the execution of the VM. Pre-copy is the default migration algorithm for Xen, which transfers memory pages iteratively. But Pre-copy has to transfer too much memory data, and then a great total migration time is brought about. Post-copy [21] is proposed to solve this problem and ensures that each memory page is transferred at most once. However, Post-copy does not have the same level of reliability as Pre-copy.

To improve migration performance, Deng et al. [22] introduce compression into migration to decrease the amount of transferred data and achieve better effect. But the normal compression also compresses and transfers much duplicated data, which would affect performance. Trace-replay [23] is also introduced into migration innovatively. The amount of traces is much smaller than that of modified memory pages, so trace-replay can improve migration performance significantly. However, in multi-processor environment, trace-replay based migration is not that suitable.

Speeding up checkpointing has already become a research focus in many years. One of the effective methods to accelerate checkpointing is to improve network bandwidth. *InfiniBand* (IB) is indeed popular in modern clusters and data centers. We can transfer VM's state data through RDMA [24] and achieve good results. However, it is expensive and complex to manage.

For a fixed bandwidth, reducing the amount of transferred data is a good choice. A common technique is self-ballooning [21]: the VMM communicates with the ballooning driver running inside the VM. There are many unused pages in free state among the VM's occupied physical memory. Before the *save* operation, self-ballooning releases the unused pages and returns them to VMM, thereby reducing VM's memory and the total amount of transferred data. Self-ballooning can only release the unused pages, but help nothing for those pages containing useful information. Memory compression [11] is another effective solution to overcome the bottleneck of network bandwidth, which is recommended for migration to decrease the amount of transferred data, and improve migration performance greatly. But it must compress all the pages including unchanged pages which account for an important proportion. Moreover, a compression algorithm which achieves high compression ratio generally takes long time to compress. Suppose that compression brings too much time overhead, it will get just the opposite to the ideal result that we have expected.

Park et al. [10] present a technique for fast and space-efficient checkpointing of virtual machines. Modern operating systems use the better part of the available memory for a page cache that caches data recently read from or written to disk. Through transparent I/O interception at the VMM level, they track I/O requests and maintain an up-to-date mapping of memory pages to disk blocks in a page_to_block map. At checkpointing time, they exclude those pages from the memory images written to disk, thereby saving a considerable amount of disk space and time.

The study most relevant to ours is [12]. In this paper, Zhang et al. study the characteristics of run-time memory image data during migration, and find that there is a high similarity among transferred pages in each iteration. They utilize the self-similarity of run-time memory image, use hash based fingerprints to find identical and similar memory pages, and employ RLE to eliminate redundant memory data during migration. But it is a big overhead to find similar pages in each iteration and they do not study the similarity of pages in different iteration. In [27], Zou et al. explore several potential data-analytics placement strategies along the I/O path and propose a flexible data analytics (FlexAnalytics) framework. FlexAnalytics enhances the scalability and flexibility of current I/O stack on HEC (*High-End Computing*) platforms and is useful for data pre-processing, runtime data analysis and visualization, as well as for large-scale data transfer.

Delta memory compression has been introduced into live migration effectively. In [18], Svard et al. study the application of delta compression during the transfer of memory pages in order to increase migration throughput and thus reduce downtime, and they achieve better performance. But they do not quantitatively analyze the changed bytes in dirty pages. What's more, they need to compress a page byte by byte, which introduces more additional overhead.

VI. CONCLUSION AND FUTURE WORK

In this paper, we first study the variation characteristics of run-time memory in byte granularity, and find that too much redundant data has been transferred. Then the design and specific implementation of FIDOC is in deep discussion, which introduces dirty page logging mechanism as well as delta memory compression into checkpointing. Furthermore, we also propose a fast method to scan VM's dirty bitmap aiming at find out the dirty pages. Finally we present the evaluation of FITDOC. Experiments demonstrates that compared with Xen's default system-level checkpointing algorithm, FITDOC can reduce 70.54% of checkpointing time on average with 1GB memory size and achieve better improvement for VMs with larger memory configurations. FITDOC can reduce 52.88% of the checkpointing data size on average compared with Remus's incremental solution which is in page granularity. Besides, compared with default dirty bitmap scanning method in Xen, the scanning time of FITDOC is decreased by 91.13% on average.

A fast method for saving full-virtualized VM has been proposed and the experimental results have verified its high

efficiency. In the future, we concentrate on how to restore a VM quickly with compressed data involved, and we will also extend support for para-virtualized VM intensively. Actually, diminishing compression overhead is full of challenge which is our main occupation in the future work.

Acknowledgement

This paper is partly supported by the NSFC under grant No.61133008 and No.61370104, National Science and Technology Pillar Program of China under grant No.2012BAH14F02, MOE-Intel Special Research Fund of Information Technology under grant MOE-INTEL-2012-01, and Chinese Universities Scientific Fund under grant No. 2014TS008.

References

[1] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer*, vol.7, pp.34-45, 1974.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003, pp.164-177.

[3] R. Nathuji and K. Schwan, "VirtualPower: coordinated power management in virtualized enterprise systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, 2007, pp.265-278.

[4] S. Jan, C. Lang, and F. Bellosa, "Energy management for hypervisor-based virtual machines," in *Proceedings of the USENIX Annual Technical Conference*, 2007.

[5] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *Proceedings of 21st ACM International Conference on Supercomputing (ICS'07)*, 2007, pp.23-32.

[6] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the performance of hypervisor-based fault tolerance," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'10)*, 2010, pp.1-10.

[7] B. Nicolae and F. Cappello, "BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011, pp.1-12.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, 2008, pp.161-174.

[9] B. Gerofi, Z. Vass, and Y. Ishikawa, "Utilizing Memory Content Similarity for Improving the Performance of Replicated Virtual Machines," in *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC'11)*, 2011, pp.73-80.

[10] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11)*, 2011, pp.75-86.

[11] L. Deng, H. Jin, S. Wu, X. Shi, and J. Zhou, "Fast saving and restoring virtual machines with page compression," in *Proceedings of the 2011 International Conference on Cloud and Service Computing (CSC'11)*, 2011,pp.150-157.

[12] X. Zhang, Z. Huo, J. Ma, and D. Meng, "Exploiting data deduplication to accelerate live virtual machine migration", in *Proceedings of the 2010 IEEE International Conference on Cluster Computing (Cluster'10)*, 2010, pp.88-96.

[13] DBENCH, http://dbench.samba.org/.

[14] The Linux Kernel Archives, http://www.kernel.org/.

[15] NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html.

[16] Welcome to Apache Hadoop, http://hadoop.apache.org/.

[17] TPC-W, http://www.tpc.org/tpcw/.

[18] P. Svard, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11)*, 2011, pp.111-120.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the second USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, 2005, pp.273-286.

[20] M. Nelson, B. Lim, and G. Hutchines, "Fast transparent migration for virtual machines," in *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*, 2005, pp.391-394.

[21] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*, 2009, pp.51-60.

[22] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive memory compression," in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'09)*, 2009, pp.1-10.

[23] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *Proceedings of the 18th International Symposium on High Performance Distributed Computing (HPDC'09)*, 2009, pp.101 – 110.

[24] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07)*, 2007, pp.11-20.

[25] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems", in *Proceedings of 18th ACM International Conference on Supercomputing (ICS'04)*, 2004.

[26] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments", in *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08)*, 2008.

[27] H. Zou, Y. Yu, and W. Tang, "FlexAnalytics: A Flexible Data Analytics Framework for Big Data Applications with I/O Performance Improvement", *Big Data Research* 1 (2014), pp.4-13.