MiyakoDori: A Memory Reusing Mechanism for Dynamic VM Consolidation

Soramichi Akiyama*, Takahiro Hirofuchi†, Ryousei Takano† and Shinichi Honiden‡
*Graduate School of Information Science and Technology,
The University of Tokyo, Tokyo, Japan 113–8656
Email: akiyama@nii.ac.jp
†National Institute of Advanced Industrial Science and Technology, Japan
Email: {t.hirofuchi, takano-ryousei}@aist.go.jp
‡The University of Tokyo, Japan / National Institute of Informatics, Japan
Email: honiden@nii.ac.jp

***Abstract*—In Infrastructure-as-a-Service datacenters, the placement of Virtual Machines (VMs) on physical hosts are dynamically optimized in response to resource utilization of the hosts. However, existing live migration techniques, used to move VMs between hosts, need to involve large data transfer and prevents dynamic consolidation systems from optimizing VM placements efficiently. In this paper, we propose a technique called "memory reusing" that reduces the amount of transferred memory of live migration. When a VM migrates to another host, the memory image of the VM is kept in the source host. When the VM migrates back to the original host later, the kept memory image will be "reused", i.e. memory pages which are identical to the kept pages will not be transferred. We implemented a system named MiyakoDori that uses memory reusing in live migrations. Evaluations show that MiyakoDori significantly reduced the amount of transferred memory of live migrations and reduced 87% of unnecessary energy consumption when integrated with our dynamic VM consolidation system.**

***Keywords*-virtualization; live migration; IaaS cloud;**

I. Introduction

Many companies use Infrastructure-as-a-Service datacenters instead of having their own servers to build their software systems with the help of the recent emergence of virtualization techniques [1]. One of main features of an IaaS datacenter is that the virtual machines (VMs) can easily move between physical machines (hosts) by using live migration techniques [2], [3]. Using live migration techniques, a VM can migrate from one host to another without any interruption. One possible application of live migration is for management purposes [4], [5]. Several studies have been conducted on optimal placements of VMs in a datacenter. For example, Goiri *et al.* developed a VM placement policy that reduces the energy consumption of a datacenter by consolidating VMs into a small number of hosts by using a live migration technique [6]. Another example is a *dynamic VM consolidation* system developed by us [7]. In this system, we use post-copy live migration [8] to achieve a quicker consolidation and better energy efficiency than with pre-copy live migration.

This work was supported in part by KAKENHI (23700048) and JST/CREST ULP.

Live migration techniques are used to optimize VM placements. However, these techniques tend to lead heavy network traffic, as existing live migration techniques transfer the entire memory image of the target VM. This makes a migration to take long time and delays the completion of the VM placements optimization. The execution of live migration is considered as a penalty parameter in an optimization problem of VM packing [6]. Our dynamic VM consolidation system does not impose such a delay as it uses post-copy live migration. However, a post-copy live migration slows down the performance of a target VM for a certain time after a migration.

In this paper, we propose a *memory reusing* mechanism to reduce the amount of transferred data in a live migration. In a dynamic VM consolidation system, a VM might migrate back to the host it has once executed. The memory image of the VM is left on the host when it migrates out from the host and the image will be reused when the VM migrates back to the host later. The fewer amount of data contributes to a shorter migration time and greater optimization by VM placement algorithms. For example, our experiments show that the energy consumption of a dynamic VM consolidation system is reduced by integrating memory reusing.

The rest of this paper is organized as follows. In Section II, we explain how live migration techniques are used and their problems from the view point of dynamic VM consolidation. Section III explains the concept of memory reusing. Section IV describes our implementation of memory reusing, called MiyakoDori. In Section V we evaluates MiyakoDori with realistic applications. Section VI and Section VII include a further discussion and review of related work. Section VIII concludes this paper.

II. Live Migration for Dynamic VM Consolidation

A. Dynamic VM consolidation

Our dynamic VM consolidation system reduces the energy consumption of datacenters by using post-copy live

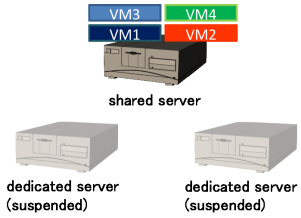

Figure 1. Dynamic VM consolidation: when idle

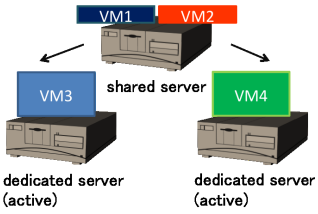

Figure 2. Dynamic VM consolidation: when busy

migration technique [7], [8]. In this system a datacenter consists of two types of physical hosts: *shared servers* and *dedicated servers*. The VMs that are not performing heavy computations are consolidated into a small number of shared servers. In the other case, VMs that are active are distributed to dedicated servers to leverage the performance of these VMs. Figure 1 and Figure 2 illustrate a datacenter that has four VMs. When all the VMs are idle, they are consolidated into a shared server, shown in Figure 1. Figure 2 shows the result of VM3 and VM4 requiring a higher performance: they are both migrated to dedicated servers. Note that *dedicated* means this scheduling algorithm explicitly assigns a physical host to an active VM to ensure VM performance.

B. Problems of Existing Live Migration Techniques

In this section, we explain the problems of existing live migration techniques when applied to dynamic VM consolidation. The main problem of existing live migration techniques [2], [3] is that they transfer large amount of data to migrate a VM. For example, to migrate a VM with 1GB memory, we need to transfer at least 1GB of data. Transferring large amount of data causes two problems:

1) The migration causes memory accesses that degrade the performance of applications running in the VM by 8% to 30% [3], [9].
2) Consolidating many VMs into a host at once congests the network of the host [10] and delays the consolidation.

A dynamic consolidation aggressively optimizes VM placements with frequent live migrations. The data transfer cost of a live migration will make a significant impact on the efficiency of dynamic consolidation systems. Therefore, we try to reduce the amount of transferred data in a live migration.

III. Memory Reusing during Migrations

In this section, we propose *memory reusing*, a mechanism to reduce the amount of transferred data of a live migration in dynamic VM consolidation. As described in Section II, a dynamic VM consolidation system executes live migrations many times. During these migrations, a VM might migrate back to a host on which it has been executed before. When a VM transit from an idle state to a busy state, the VM must

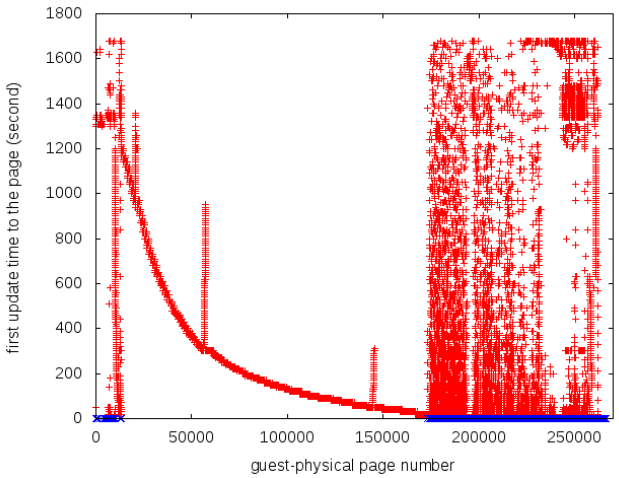

Figure 3. Write access pattern of TPC-C workload: X-axis and Y-axis represent the guest-physical page number and the time when the page is updated for the first time, respectively. Points with a Y value of 0 are blue, points with values larger than 0 are red. Note that 0 for Y value means that the memory page was never updated during the 30 minutes of workload.

be moved from a shared server to an idle server. On the other hand, when the VM becomes idle again, it is moved to the shared host. For instance, VM3 in Figures 1 and 2 makes a round trip between the shared server and its dedicated server. By caching the memory image of the VM beforehand, the entire memory image transferring can be avoided.

A. Write Access Pattern of a Compound Workload

We assume that even when a VM is actively running, a substantial number of memory pages will remain unchanged. Unused memory pages and the ones keeping read-only data of workloads will not be updated. In order to ensure this assumption, we executed the TPC-C [11] workload in a VM and recorded the memory write accesses pattern. TPC-C is a workload that simulates transaction processing system and includes CPU utilization, memory access, and IO.

We modified the QEMU/KVM [12], [13] to record the write access pattern to the memory. QEMU has a functionality named *dirty page tracking*, which detects the first update of a memory page and sets the corresponding bit of the *dirty page table* to 1. Our modified version of QEMU records the time when each memory page is updated for the first time by reading the dirty page table every T seconds.

We executed the workload for 30 minutes on a VM with 1GB memory and 1 virtual CPU, and set T to 10. Figure 3 shows the results of this experiment. For example, a red point with a Y value of 600 indicates that the page was updated for the first time between 590 seconds and 600 seconds after the beginning of the workload.

The results show that Y values are scattered in a wide range, which indicates that memory reusing would successfully reduce the overhead of a live migration. For example,

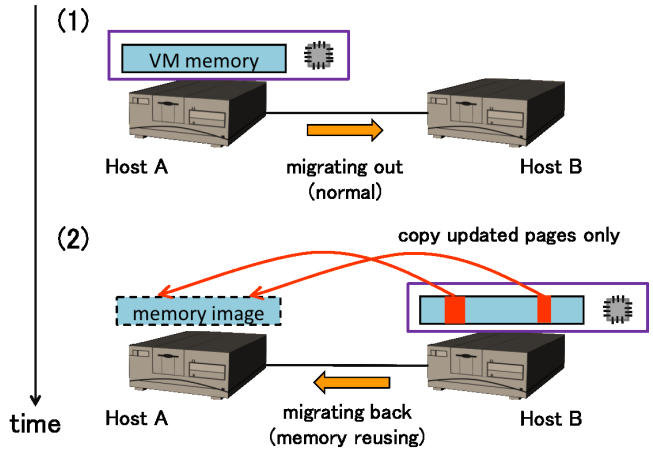

Figure 4. Basic idea of memory reusing: The upper row illustrates the first migration from host A to host B, in which memory reusing cannot be applied. The lower row illustrates a migration back from host B to host A, in which the memory image of the VM is kept on host A and unchanged memory pages are not transferred.

suppose the VM migrates to another host just after the beginning of the workload and migrates back to the original host 600 seconds later. In this case, all the memory pages with Y values between 600 and 1800 need not to be transferred: they are not updated until at least 600 seconds after the beginning of the workload. For example, in this evaluation 17% of the points are between Y=600 and Y=1800.

B. Basic Idea of Memory Reusing

The basic idea of memory reusing is to keep the memory image of a VM on a host in order to reuse it when the VM migrates from another host at a later stage.

Figure 4 illustrates the basic idea. (1) The upper row illustrates a VM that migrates from host A to host B. Memory reusing is not applied to this migration because it is the first migration of the VM to host B. Therefore, all the memory pages of the VM must be transferred on the network. At this time, we keep the memory image of the VM on host A in order to reuse it later. (2) The lower row illustrates the VM when it migrates back from host B to host A. Some memory pages of the VM have been updated since the memory image has been kept on host A. However, many memory pages remain unchanged (as shown before in this section) and they are not transferred from host B to host A.

IV. Implementation: MiyakoDori

We implemented the memory reusing mechanism and integrated it into QEMU. We call this system MiyakoDori[1]. In this section, the design policy and the implementation of MiyakoDori are described.

[1]MiyakoDori is a migrating bird in Japanese.

A. Live Migration with Memory Reusing

Each memory page of a VM has a *generation*, which indicates how many times the page has been updated since the boot of the VM. When a VM boots, generations of all the memory pages are set to zero. We call the set of all generations of a VM as the *generation table* of the VM. A *generation server* manages the generation tables of all the VMs. We use the tuple (V, A) to refer to the generation table of VM V associated with host A.

Figure 5 illustrates the behavior of MiyakoDori when a VM V is migrated from host A to host B for the first time. Note that, at this stage memory reusing is not utilized because no memory image is kept yet.

1) Stop temporarily VM V on host A.
2) Detect memory pages that have been updated since the boot time of VM V by using dirty page tracking.
3) Send updated page numbers to the generation server, which propagates the changes to (V, A).
4) Resume VM V. This pause is less than a second because the data transferred in (3) is quite small, as describe in Section VI
5) Migrates VM V to host B by using a normal live migration mechanism, but keeps the memory image on host A for possible later reuses.

Figure 6 illustrates the behavior of MiyakoDori when a VM V is migrated back from host B to host A. The differences from the previous procedure is **emphasized**. Figure 5 does not include explanations for the same procedure as in Figure 6.

1) Stop temporarily VM V on **host B**.
2) Detect memory pages that have been updated since **the last migration to host B** by using dirty page tracking.
3) Send updated page numbers to the generation server, which propagates the changes to **(V, B)**.
4) **The generation server compares (V, A) with (V, B) to find reusable pages, i.e. the pages that have the same generations in the two generation tables.**
5) **The generation server sends reusable page numbers to the hosts.**
6) Resume VM V.
7) Migrates VM V to **host A, without transferring reusable memory pages**.

B. Design Policy

The design of MiyakoDori is based on a client-server architecture. A central management server collects generation values from all physical hosts and controls VM placement of a datacenter. We call the central management server as generation server in above explanations of our implementation. This architecture allows us to develop advanced placement algorithms, which exploit cache availability on each host as an input parameter. Further discussion will be made in our future work.

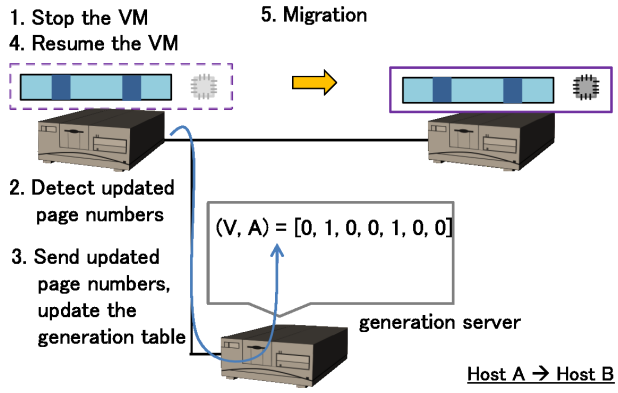

Figure 5. First migration of a VM: The memory image is cached on the source host. The generation table is updated and MiyakoDori knows later which pages can be reused.

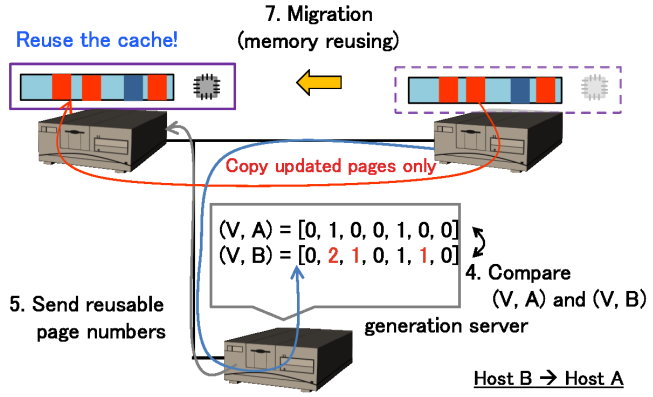

Figure 6. Migration back to a host on which the VM has once been executed: All the memory pages that have different generations in the source and the destination are copied. Other pages are reused and do not need to be copied.

C. Dirty Page Tracking

Dirty page tracking is a functionality provided by the original QEMU/KVM. A user-level function of QEMU detects via dirty page tracking which memory pages are updated. In x86 and x64 architectures, when a memory page is updated, the CPU sets the dirty bit of the corresponding page table entry (to 1). Dirty page tracking acquires this bit and makes it available to user-level functions. Since the page table entry is touched only once when the first update to the page occurs, there is practically negligible overhead in using dirty page tracking permanently, as describe in Section VI.

V. Experiments

We evaluated MiyakoDori with application benchmarks and integration with our dynamic VM consolidation. In the application benchmarks, we used four benchmarks and evaluated the amount of data transferred in a pre-copy live migration with/without MiyakoDori. In addition, we integrated MiyakoDori into our dynamic VM consolidation system [7] and estimated the reduction of energy consumption achieved by memory reusing.

Table I
Specifications of the hardware and the software

CPU	Intel Xeon X5460 (4 cores)
Memory	8 GB
Disk	250GB HDD
Network	GigaBit Ethernet NIC
OS	Debian GNU/Linux 6.0
kernel	Linux 2.6.32
KVM	kvm-kmod-2.6.38.6
QEMU	0.13.0

Table II
Workloads for application benchmarks

Name	Intensity	Detailed description
Busy Loop	CPU	Infinite busy loop
Apache	Network, IO	Read 1,024 static HTMLs (256KB each) with 100Mbps
Video	IO	Transcode mpeg2 video into ogg theora format using vlc media player
TPC-C	(Compound)	DB access benchmark that simulates transactions of an online shop

A. Application benchmarks

In this benchmarks, we evaluated to what extent MiyakoDori can reduce the amount of transferred memory in a single live migration. We used three servers for two hosts and a generation server. The specifications of used hardware and software are shown in Table I. We set up a VM with 1GB of RAM and one virtual CPU. We used Ubuntu 10.10 server as a guest OS. Disk images are located on the generation server, accessible from the VM on any host via Network File System (NFS).

The evaluations are performed as follows:

1) Launch a VM V on host A
2) Run a workload on VM V for N minutes
3) Migrate VM V from host A to host B
4) Run the workload on VM V for *another* N minutes
5) Migrate VM V from host B to host A

We executed this procedure with/without MiyakoDori and compared the amount of transferred memory in the step 5). Using MiyakoDori, the memory pages updated in the latter N minutes are transferred. However, all the used memory pages are transferred without MiyakoDori. For each test case, we used one of the workloads described in Table II and set variable N to either 5, 10, or 15.

The amount of transferred memory in each test cases is shown in Figures 7, 8, 9, and 10. The X-axis is the interval between two migrations ($=$ N minutes) and the Y-axis is the amount of transferred memory in MB. Note that the maximum values of the Y-axis are not the same in all the figures. A bar in right-hand side in each interval (labeled

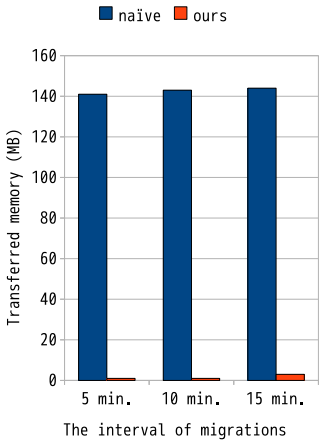

Figure 7. Results for Busy Loop

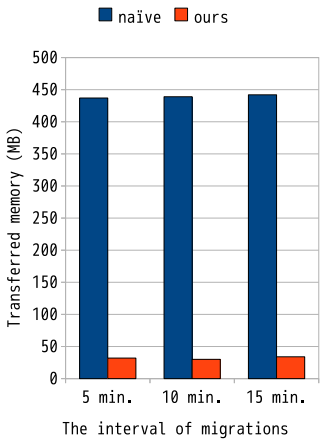

Figure 8. Results for Apache

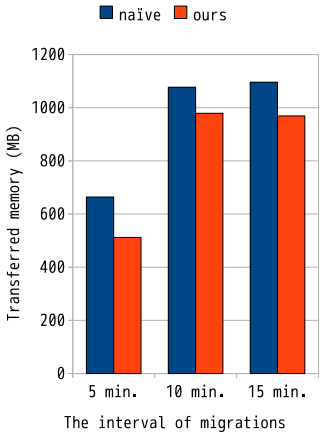

Figure 9. Results for Video

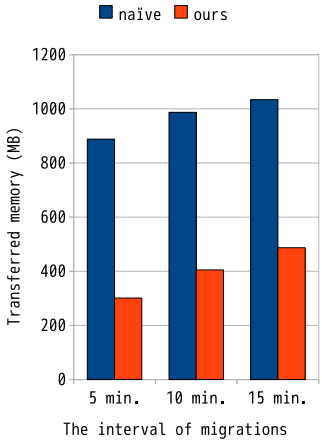

Figure 10. Results for TPC-C

as naive) describes the result with memory reusing and left-hand side (labeled as ours) is for the case without memory reusing. In the following, we discuss the results of each test case:

Busy Loop: In this workload only a small number of memory pages were updated. Therefore, more than 95% of memory pages could be reused.

Apache: This workload consumes more memory than Busy Loop but the ratio of reused memory pages to non-reused pages is similar to Busy Loop. The reason is that reading files does not updates many memory pages, even if the total size of read files is large. The guest operating system kept all static HTML files during this experiment. The memory pages used for this file cache were not updated. The results show that memory reusing is suitable for a workload that has large static data.

Video: In this test set, the largest amount of memory is consumed compared with the other three workloads. The difference between Video and Apache is that Video writes large amount of data (i.e. converted video). The operating system caches the written file in the memory, therefore many memory pages were updated in this workload. The results show that memory reusing cannot be utilized well in workloads that produce large output data in the memory. Though the merit of memory reusing was small in Video workload, some memory pages were still reused. These pages are used by the operating system (e.g. text segment of loaded program).

TPC-C: This workload is a compound workload of CPU, memory and IO. The memory access patterns of the workload are complex, as mentioned in Section III. Even though 79% of the memory are used for the workload (that is, 79% of points have larger values than 0 in Figure 3), more than 50% of the pages were reused. The results clearly indicate that memory reusing is beneficial for compound workloads similar to TPC-C (i.e. transaction system).

Table III
TOTAL MIGRATION TIME IN THE BUSY LOOP TEST CASE

N	naive	ours
5 min.	7 sec	<2 sec
10 min.	7 sec	<2 sec
15 min.	7 sec	<2 sec

Table IV
TOTAL MIGRATION TIME IN THE APACHE TEST CASE

N	naive	ours
5 min.	16 sec	2 sec
10 min.	16 sec	2 sec
15 min.	16 sec	2 sec

Table V
TOTAL MIGRATION TIME IN THE VIDEO TEST CASE

N	naive	ours
5 min.	22 sec	16 sec
10 min.	34 sec	31 sec
15 min.	35 sec	31 sec

Table VI
TOTAL MIGRATION TIME IN THE TPC-C TEST CASE

N	naive	ours
5 min.	28 sec	10 sec
10 min.	32 sec	13 sec
15 min.	33 sec	15 sec

Tables III, IV, V, and VI show total migration time. The expression "<2 sec" in Table III means that the results differ from case to case but less than 2 seconds. Note that a total migration time is the time period between the start of the migration and the time when all the states of the VM are successfully transferred. The total migration time is proportional to the amount of transferred data, since transferring the memory image account for the largest part of a live migration.

The evaluations show that:

1) Memory reusing can reduce the amount of transferred memory in a live migration with all of the four workloads.
2) Workloads that have small working set (e.g. Busy Loop) or large read-only data (e.g. Apache) are especially suitable for memory reusing.
3) Workloads that produce large data (e.g. Video) does not take advantage of memory reusing.
4) Total migration time is also reduced in proportion to the amount of transferred data.

Table VII
TOTAL MIGRATION TIME AND REUSED MEMORY SIZE IN DYNAMIC VM CONSOLIDATION

Migration	1	2	3	4	5	6
w/ MiyakoDori (seconds)	18	17	18	17	18	17
w/o MiyakoDori (seconds)	19	5.6	0.1	0.3	0.1	0.5
Reused (bytes)	0	1.3G	1.9G	1.9G	1.9G	1.9G

B. Integration with our dynamic VM consolidation system

We integrated MiyakoDori into our dynamic VM consolidation system [7] and evaluated how memory reusing contributes to a real situation.

We used three hosts and four VMs for the evaluation. The hosts are a dedicated server, a shared server and a management. A management node gathers load information of all the VMs and hosts and commands VM replacement to the hosts if needed. The specifications of physical hosts are the same with the micro evaluation (Table I). Three of four VMs are *static VMs* consume 80% of the CPU core assigned to it. The last VM is a *migrating VM* and configured to have 1900MB of memory. The migrating VM runs Apache web server and has 5,000 static HTML files with 256KB each. We applied dynamic VM consolidation in following settings.

Workload: The HTML files are read with two different speeds, 60 files/sec ($\approx$ 120Mbps) and 10 file/sec ($\approx$ 20Mbps). Two speeds are alternately applied with 5-minute interval. That is, first the speed is 10 files/sec, and then it is switched to 60 files/sec 5 minutes later, and this cycle is repeated for 30 minutes (3 times).

Scheduling Policy: Three static VMs never change their CPU usage nor the host on which they run. The migrating VM changes its CPU usage in response to the speed of the read and moves as follows:

1) to the dedicated server if its CPU usage is over 80%
2) to the shared server if its CPU usage is under 30%

The reason why we move the migrating VM with 80% of CPU usage is that the total CPU usage of four VMs reaches 320%, which is 80% of the total CPU capacity of the shared server (= 400%, as the shared server has 4 CPU cores).

The migrating VM makes a round trip in response to switches of reading speed of the HTML files. We evaluated the total migration time and the amount of reused memory on each live migration in the 30-minute workload, with and without MiyakoDori. The results are shown in Table VII. We executed the workload 3 times and took the average value.

In migrations 2, 4, and 6, the migrating VM moves from the dedicated server to the shared server and the dedicated sever can be suspended after the migrations. The expression "after the migrations" means that the dedicated server must be powered on until all the state of the VM is transferred to the shared server. Suppose an optimal situation where live migration has no time overhead and the dedicated server can be powered off just after the migration started. In this situation, the time duration in which the dedicated server must be powered on is 15 minutes (the half of 30 minutes workload). We can estimate the amount of unnecessary energy consumption E (i.e. extra energy consumption compared to the optimal situation) by:

$$E = \alpha \sum_{i \in \{2,4,6\}} T_i \tag{1}$$

where T_i is the total migration time of the i^{th} migration and α is energy consumption per second of the dedicated server. The reduction ratio of the unnecessary energy consumption by memory reusing, R, is calculated by:

$$R = \frac{E_{without} - E_{with}}{E_{without}} \tag{2}$$

$$= \frac{51 - 6.4}{51} \tag{3}$$

$$\approx 87\% \tag{4}$$

where $E_{without}$ and E_{with} are estimated by equation (1) and Table VII.

In this evaluation, we integrated MiyakoDori into our dynamic VM consolidation system and estimated the reduction ratio of the unnecessary energy consumption. The result show that memory reusing significantly reduces the unnecessary energy consumption by shortening total migration time of live migrations.

VI. DISCUSSION AND FUTURE WORK

A. Discussion about Overhead

Even though MiyakoDori introduces two types of overhead into existing systems, we claim that in the real world this overhead is negligible for the following reasons:

Memory Access Slowdown: Setting the dirty bit of a memory page requires the page table to be checked in each memory access even if the virtual address of the page is cached on the Translation Look-aside Buffer. Therefore, dirty page tracking slows down memory accesses. In our experiments, dirty page tracking slows down only the first access to a page for approximately 20%, but no slowdown was observed in the completion time of the CG benchmark from NAS Parallel Benchmarks [14].

Network Overhead: MiyakoDori requires extra data to send updated page numbers to the generation server and to send reusable page numbers to the hosts. This overhead is very small because we only need 1 bit to represent whether a page is updated or not. Suppose that a VM has 4GB of memory, then we need to send only 4 GB $\div$ 4 KB/page $\times$ 1 bit/page = 1 Mbits for updated page numbers (the same calculation is also applied for reusable page numbers).

CPU Downtime: The virtual CPU of a VM needs extra suspension by MiyakoDori, as described in Section IV. However, this suspension is less than a second because the

data transferred during the suspension is enough small, as described above.

B. Future Work

Further Analysis of Memory Access Patterns: As described in Section III, write access patterns to the memory have certain characteristics that depend on the workload. We believe that by analyzing memory access patterns, including reads and writes, live migration techniques can be further improved. For example, analysis of memory access patterns of a workload indicates the assumed size of the working set of the workload [15].

Memory Cache Management: Current implementation leaves all the pages of all the memory images, except when the host OS swaps them out. However, a memory image includes many unnecessary pages. For instance, a page that is updated on another host is no longer needed because the page cannot be reused. Other types of unnecessary pages can be detected by further analysis of memory access pattern of the workload. For example, a memory page that has strong possibility of update in a near future does not need to be cached.

VII. Related Work

In this section, we briefly review related approaches that try to reduce the amount of transferred data of live migration. In [16], the authors use run-length compression to reduce the amount of transferred data. In a pre-copy live migration, a page could be transferred more than twice because of memory updates during the transfer. This study claims that even if a page is *dirtied* the difference between the current version and the previous version of the page is not large. A bit-wise xor of two versions is a bit sequence that includes long running 0s. Run-length compression technique is used to compress the bit sequence because it shows a high compression rate against a long running 0s and calculation cost for the compression is very small. The authors of study [17] also compress the memory pages but utilize similarity of two different pages, as well as two versions of an identical page. According to [17], in the first phase of a pre-copy live migration 25% of used pages have an identical page to it and thus does not need to be transferred.

Study [18] transfers execution logs of a target VM instead of the its memory image. This reduces the amount of transferred data in a live migration because a execution log does not contain unnecessary information to restore the state of the VM.

Post-copy live migrations [8], [19], [20] address overhead of a live migration in a different way. In this kind of migration, the states of the virtual CPU are migrated before transferring its memory image. Memory pages of the VM are copied on-demand when they are read/written by the already migrated CPU. A post-copy migration is superior to pre-copy one in following two aspects.

1) A memory page is transferred only once because there is no memory update during the transfer.
2) Transferring the virtual CPU is done immediately while it waits for transferring the whole memory image in a pre-copy live migration.

A drawback of a post-copy live migration is that it degrades performance of the VM due to page faults occurred by on-demand memory copies. Countermeasures for this drawback are page pre-fetching using locality of memory accesses [8], [19] and clustering of memory page characteristics enabled by analyzing page table entries and OS specific information [21].

We used our dynamic VM consolidation system [7] to evaluate memory reusing mechanism in a real setting. More aggressive relocations to reduce energy consumption are studied in AASH [22] or HASS [23]. These studies change assignments of processors to processes using modified kernel scheduler. When a process is in active phase a high performance and energy-consuming processor is suitable while low performance processor is appropriate when the process waits for memory accesses or IO completions a lot. This idea can be extended to assignments of hosts to VMs using live migration technique, if a live migration can be executed with a very low cost.

VIII. Conclusion

In this paper we proposed memory reusing, which reduces the amount of transferred memory of a live migration in the context of dynamic VM consolidation systems by keeping memory images of VMs in hosts and skipping transfer of unchanged memory pages. To verify the feasibility of memory reusing, we implemented a prototype system named MiyakoDori. Evaluations showed MiyakoDori reduces the amount of transferred memory and total migration time of a live migration and thus reduces the energy consumption of a dynamic VM consolidation system. Future work includes further analysis of memory access patterns and memory cache management in order to improve the memory reusing mechanism.

References

[1] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, "The evolution of an x86 virtual machine monitor," *SIGOPS Opereting Systems Review*, vol. 44, pp. 3–18, Dec. 2010.

[2] 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 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005, pp. 273–286.

[3] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005, pp. 391 – 394.

[4] V. Soundararajan and J. M. Anderson, "The impact of management operations on the virtualized datacenter," in *Proceedings of the 37th annual International Symposium on Computer Architecture*, 2010, pp. 326–337.

[5] Distributed resource scheduling, distributed power management of server resources. [Online]. Available: http://www.vmware.com/products/drs/

[6] I. Goiri, F. Julia, R. Nou, J. L. Berral, J. Guitart, and J. Torres, "Energy-aware scheduling in virtualized datacenters," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, 2010, pp. 58–67.

[7] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Reactive consolidation of virtual machines enabled by postcopy live migration," in *Proceedings of the The 5th International Workshop on Virtualization Technologies in Distributed Computing*, 2011, pp. 11–18.

[8] T. Hirofuchi, H. Nakada, S. Itoh and S. Sekiguchi, "Enabling instantaneous relocation of virtual machines with a lightweight vmm extension," in *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2010, pp. 73–83.

[9] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang, "Live migration of multiple virtual machines with resource reservation in cloud computing environments," in *Proceedings of the 2011 IEEE Conference on Cloud Computing*, 2011, pp. 267–274.

[10] S. Kikuchi and Y. Matsumoto, "Performance modeling of concurrent live migration operations in cloud computing systems using prism probabilistic model checker," in *Proceedings of the 2011 IEEE Conference on Cloud Computing*, 2011, pp. 49–56.

[11] Tpc-c. [Online]. Available: http://www.tpc.org/tpcc/

[12] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005, pp. 41 – 46.

[13] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the 2007 Linux Symposium*, 2007, pp. 225 – 230.

[14] NAS PARALLEL BENCHMARKS. [Online]. Available: http://www.nas.nasa.gov/Resources/Software/npb.html

[15] A. Garg, "Looking inside memory - tooling for tracing memory reference patterns," in *Proceedings of the 2010 Linux Symposium*, 2010, pp. 63–74.

[16] P. Svärd, 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*, 2011, pp. 111–120.

[17] 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*, 2010, pp. 88–96.

[18] 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 ACM International Symposium on High Performance Distributed Computing*, 2009, pp. 101–110.

[19] 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*, 2009, pp. 51–60.

[20] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: Rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009, pp. 1–12.

[21] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara, "Kaleidoscope: Cloud micro-elasticity via vm state coloring," in *Proceedings of the Sixth Conference on Computer Systems*, 2011, pp. 273–286.

[22] V. Kazempour, A. Kamali, and A. Fedorova, "AASH: an asymmetry-aware scheduler for hypervisors," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2010, pp. 85–96.

[23] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: a scheduler for heterogeneous multicore systems," *SIGOPS Operating Systems Review*, vol. 43, pp. 66–75, April 2009.