JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 36, 423-440 (2020)
DOI: 10.6688/JISE.201903_36(2).0017

Live Save: An Efficient Snapshot Mechanism for Virtual Machines in Xen

PO-JEN CHUANG AND YEN-CHIA HUANG
Department of Electrical and Computer Engineering
Tamkang University
Tamsui, New Taipei City, 25137 Taiwan
E-mail: pjchuang@ee.tku.edu.tw

We present an efficient snapshot mechanism, briefed as Live Save, to make real time backup of the virtual machine state to a local host. Live Save will iteratively send the state data, store the snapshot file in a local host and, when necessary, send the entire file directly to a remote host – to reduce network bandwidth consumption in existing snapshot practices. In iterative transmissions, when dirty frequency rises to a certain high level, iterations may fail to reduce dirty pages and hence become redundant. To handle the problem, we add a simple judging measure to Live Save to form an Improved Live Save which will adaptively adjust the maximum number of iterations. By maintaining the maximum number of iterations, Improved Live Save can avoid futile iterations – saving unnecessary iterative computation in Live Save. Extended simulation runs are conducted to evaluate the performance of various snapshot mechanisms (Xen Save, Xen Live Migration, Live Migration Save (VNsnap), Live Save and Improved Live Save) in terms of downtime, total execution time and total pages sent. The results show that, in performing the snapshot practice, both Live Save and Improved Live Save are more resource-conserving than the other mechanisms.

Keywords: cloud computing services, virtualization, Xen, virtual machines, state snapshot mechanisms, dirty frequency, performance evaluation

1. INTRODUCTION

Virtualization techniques [1-5] are important for cloud services because they help to build fully functional virtual machines (VMs) on limited physical resources to maximize hardware utilization and system performance. Current virtualization techniques tend to use such physical resources as CPU, memory, network interface cards or storage to build fully functional VMs which can execute operating systems and applications like real computers. Aided by the techniques, cloud service providers can now turn one physical server into a number of VMs to maximize hardware utilization and conserve physical resources [6-11]. In practice, when hardware or software errors occur in a computer with multiple VMs, it may cause VMs to become faulty and consequently affect cloud systems and services. To maintain proper system performance, a cloud service provider must back up VMs in advance so that he can recover a faulty VM – when it happens – at the shortest time. Traditionally, we can back up a system in the shutdown period. But it is infeasible to shut down a cloud system for VM backup because the impact on services can be serious. To lessen the impact, we can involve the snapshot technique [12] to back up virtualization systems at minimized downtime. The snapshot of VMs, which usually

Received August 15, 2018; revised November 18, 2018 & April 2, 2019; accepted June 10, 2019.
Communicated by Jan-Jan Wu.
[*] A preliminary version of this paper was presented at the 10th IEEE International Conference on Service-Oriented Computing and Applications, Kanazawa, Japan, Nov. 2017.

contains the backup of hard disks and system states (memory + vCPU), can handily help us to restore a faulty system to the checkpoint.

Xen [13-15] is a popular virtualization platform without the snapshot function. To obtain the snapshot of hard disks and system states, it depends on the Logical Volume Manager in the Linux system [16-18] and the **Xen Save** function. To generate the image file of the VM state, **Xen Save** must shut down the VM first and the generation time of a snapshot (*i.e.*, the system downtime) is subject to the memory size of the VM. It may take tens of seconds during which cloud services must totally stop. To shorten the downtime, **VNsnap** [19, 20] builds a **Live Migration Save** mechanism based on **Xen Save** and **Xen Live Migration** commands for distributed systems [21, 22]. **Live Migration Save** (**VNsnap**) uses a host as the target to receive and store the VM state snapshots. It successfully reduces the system downtime in **Xen Save**. The reduction is nevertheless attained with considerable network bandwidth consumption – because the **VNsnap Live Migration Save** needs a daemon to receive and store the state data of VMs and has to send the memory data continuously in the Internet.

To reduce the obvious bandwidth consumption in **Live Migration Save**, we set up a new snapshot mechanism – briefed as **Live Save** – in this investigation. Different from **Live Migration Save**, the proposed **Live Save** sends the VM state data in iterations, stores the snapshot file in a local host and, will directly transmit the complete snapshot file to a remote host only when necessary. Note that, in iterative transmissions, when dirty frequency (*i.e.*, the number of dirty pages generated per unit time) reaches a certain high level, iterations may find no page for transmission and become resource-wasting redundant iterations. To handle the situation, we set up a simple judging measure in the original **Live Save** to form a modified **Improved Live Save** mechanism which can adjust the allowed maximum number of iterations *adaptively*, as follows. In the iterative practice, when **Improved Live Save** detects dirty frequency rises to such a high level that iterations cannot effectively reduce dirty pages, it will decrease the maximum number of iterations by 1 after each iteration to prevent redundant iterations. On the other hand, when dirty frequency drops below the level, the modified mechanism will increase the maximum number by 1 – without exceeding the preset threshold, to maintain proper performance. The simple judging measure enables **Improved Live Save** to avoid insignificant iterations and, as a result, to perform better in total execution time and transmitted memory data than the original **Live Save**.

Extensive simulation runs are conducted under different CPU cap values and Domain-U settings, to evaluate the performance of various snapshot mechanisms (**Xen Save**, **Xen Live Migration**, **Live Migration Save** (**VNsnap**), **Live Save** and **Improved Live Save)** for a number of parameters (*dirty frequency*, *numbers of dirty pages*, *iteration time*, *numbers of iterations*, *downtime*, *total execution time* and *total pages sent*). The obtained results show that, in contrast to other mechanisms, the proposed **Live Save** and **Improved Live Save** both perform with reduced *network bandwidth consumption* and *downtime*. **Improved Live Save**, in particular, performs better in *numbers of iterations*, *total execution time* and *total pages sent*. The performance gain comes mainly from the following key designs: (1) *live* backing up VM states to the local host ("*live* backup" indicates to back up the VM state in a minimal shutdown period); (2) without sending the memory data iteratively in Internet; and (3) *adaptively* adjusting the maximum number of iterations to avoid futile iterations.

2. BACKGROUND STUDY

2.1 Xen Save [13]

Originally, **Xen** uses the **Save** command to direct a VM into the standby mode and uses the **Restore** command to wake it up into the operation mode. When **Save** is issued, the VM will shut down its operation to generate the image file of its state. The size of the image file, as mentioned, equals the memory size of the VM and the file generation time – which depends also on the memory size – may range as long as tens of seconds. The standby VM will wake up upon the issue of **Restore** to resume its operation according to the generated image file (which functions as the state snapshot of the VM).

2.2 Migration [21, 22]

We can perform VM migration – offline or live – to move VMs from a physical source host to another physical target host with identical virtualization environments.

Offline migration: A VM will be shut down completely before migrating from the source to the target and will restore the operation only after the migration ends (*i.e.*, the original service will totally halt during the migration).

Live migration: A VM can move from the source to the target without being shut down. It will maintain the operation during most of the migration, cease only briefly at the very end (when to leave the source to the target) and instantly revive the operation when the migration ends.

2.3 Live Migration [21-27]

In contrast to offline migration, live migration is a more feasible VM migration mode for cloud computing and virtualization practices because it helps to maintain almost all services in the migration (so that cloud servers can upgrade, replace or repair devices without interrupting routine services). Live migration can be post-copy or pre-copy – the latter is currently the mainstream application in commercial software, including **Xen**. VM live migration usually works by the following three phases, but different migration modes may involve varied phases. For instance, post-copy migration works by the second and third phases, whereas pre-copy migration (such as **Xen Live Migration**) involves the first and second.

The push phase: When live migration starts, the VM will maintain its operation at the source host and simultaneously keep sending data to the target. To preserve data consistency in both ends, it must re-send all changed data in the process.

The stop-and-copy phase: The VM will briefly stop the service only at the instant to leave the source host. It will keep sending the remaining memory state data to the target in this brief suspension. After all state data are sent to the target host, the VM – which has been officially migrated from the source to the target now – will instantly resume the service.

The pull phase: After the migration, when the VM in the target host accesses uncopied memory data to result in page faults, the memory data will be sent from the VM in the source to the VM in the target.

2.4 Xen Live Migration [22]

Xen Live Migration moves a VM from the source host to the target in the following steps.

Pre-migration: The source host first sends a migration inquiry to the target to check the compatibility before migration starts.

Reservation: With compatibility checked, the source will issue the migration request and, after receiving confirmation from the target, start to transfer the data.

Iterative pre-copy: The source will copy the VM memory data and send the copy to the target in iterations. In the iterative process, all memory data will be sent in the first iteration. In following iterations before each ends and begins, call XEN_DOMCTL_SHADOW_OP_CLEAN to get the bitmaps of dirty pages (changed memory pages) and copy the two bitmaps to to_send and to_skip. Clear the two bitmaps for re-recording in subsequent iterations. The two bitmaps (which respectively indicates the pages turning dirty in the previous and current iterations) will reveal the pages to be sent in the current iteration: *i.e.*, the dirty pages which were generated in the previous iteration but are not changed (not turning dirty again) in the current iteration. Then check at the end of each iteration to see if it reaches the stop condition: If yes, go to the next step (**stop-and-copy**); if no, keep sending the memory data in iterations.

Stop-and-copy: Set last_iter to 1. The source will shut down the VM, copy the dirty page bitmap to to_send, and send the memory pages with to_send or to_fix = 1 (*i.e.*, the dirty pages generated in the last iteration or the frequent dirty pages) – along with the state of CPU – to the target.

Commitment: After receiving the complete and consistent data of the VM, the target will notify the source to shut down the VM.

Activation: The VM which is now completely migrated from the source to the target will instantly activate the service.

2.5 Live Migration Save (VNsnap) [12, 19, 20]

Snapshot studies used to be hard disk oriented, *i.e.*, we can use the snapshot technique to fix the hard disk and restore the VMs. In cloud services, it is obvious that we can use the VM state snapshot backup file to restore a faulty VM *more instantly*. Recall that, during VM migration, **Xen Live Migration** will keep sending the VM's memory data iteratively until there remain only a few highly modifiable memory pages. It will shut down the VM and transfer the remaining data (with the CPU state) to the target at

the end of the migration to retain as much service as possible. To further reduce the shutdown duration in **Xen Live Migration**, **Live Migration Save** (**VNsnap**) uses a physical host as the target to receive and store the state snapshot file of a migrating VM. The new design shortens the required system downtime and preserves more routine services in the snapshot process.

Table 1 lists the basic features of **Xen Save**, **Xen Live Migration** and **Live Migration Save** to assist understanding.

Table 1. The features of Xen Save, Xen Live Migration and Live Migration Save (VNsnap).

	Xen Save	Xen Live Migration	Live Migration Save
Live	no	parameter "–l" is live	yes
VM *downtime*	the entire backup period	the stop-and-copy phase	the stop-and-copy phase
Destination of VM migration	specified file location in the host	the target host (the file not retained)	the target host (the file retained)
VM state at the end of backup	when "-c" backup ends, VM resumes the operation	the source stops and the target starts to work	resume the operation when backup ends

3. THE PROPOSED SNAPSHOT MECHANISMS

To maintain proper cloud system performance, this paper presents an efficient new snapshot mechanism – **Live Save** – to minimize system downtime when backing up the VM states. In contrast to **Live Migration Save** which needs an additional daemon to receive the memory data of VMs, **Live Save** is more resource conserving as it *live* backs up the VM state to a local host without iteratively sending memory data in the Internet.

3.1 Live Save

Built also over **Xen Save**, the proposed **Live Save** aims to reduce the network bandwidth consumption in existing **Live Migration Save**. Recall that **Live Migration Save** uses an extra daemon host to receive and store the VM state data, which is quite bandwidth consuming as it must send the memory data (including those being overwritten by the newly transferred data) iteratively and continuously in the Internet. To fix the situation, our **Live Save** sends the VM state data in iterations and stores the file in a local host instead of a remote one – with *no* network bandwidth consumption as shown in Fig. 1. When it is necessary to send the snapshot file to a remote host (*e.g.*, when a VM gets faulty), **Live Save** can also work out with *less* complexity. It will *directly* send the generated complete snapshot file to the remote host after iterations end, consuming obviously less network bandwidth than **Live Migration Save** which sends memory data (including redundant ones) *continuously* in the Internet.

To sum up, **Live Save** works differently from **Live Migration Save** in (1) sending the VM state data in iterations and storing the snapshot file in a local host, and (2) directly sending the generated complete snapshot file to a remote host only at need. Both practices help to bring down the required network bandwidth consumption. (Table 2 gives the pseudo code of our **Live Save**.)

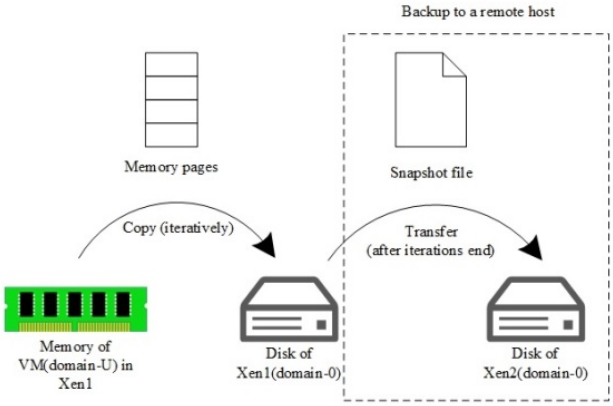


Fig. 1. The architecture of live save.

Table 2. The pseudocode of our live save.

```
1  // start live backup by iteratively transmitting
2  // the VM's memory data to the local hard disk
3   While (not any of the 3 stop conditions) {
4     // the stop conditions:
5     // (1) the number of iterations exceeds MAX_iterations (to be Xen's
6     //          MAX_iterations, i.e., 30, initially)
7     // (2) dirty pages generated in the previous and current iterations are
8     //          less than 50
9     // (3) total pages sent exceed the triple of memory size
10          if (first iteration)
11             copy all the VM's memory data into the local hard disk
12          else   // subsequent iterations
13             send the dirty pages to the local hard disk
14               // the dirty pages to be transmitted in this iteration are the
15               // dirty pages each of which has been generated in
16               // the previous iteration but has not become dirty again
17               // in this iteration
18  }
19  // stop and copy
20  shut down the VM and stop service briefly
21  copy remaining dirty pages and CPU state into the local hard disk
22  activate the VM and resume service
```

`

3.2 Improved Live Save

Experimental evaluation shows that system *downtime* in our snapshot operation will culminate when dirty frequency reaches a certain high level, and that *the numbers of iterations* and *total pages sent* will affect the length of *downtime*. Based on the results, we set up a simple judging measure to manage the two parameters in order to reduce *downtime* and preserve system performance.

3.2.1 The problem of dirty frequency

Like **Xen Live Migration**, our **Live Save** will terminate the iterations on three stop conditions:
(1) when the number of iterations exceeds 30 (preset in **Xen**)
(2) when dirty pages generated in the previous and current iterations are less than 50
(3) when total pages sent exceed the triple of the memory size

Following the three stop conditions and the simulation results, we attain three dirty frequency phases – **low**, **medium** and **high**, each corresponding to one of the stop conditions:

low → the generated dirty pages in the previous and current iterations are less than 50
medium → the total pages sent exceed the triple of the memory size
high → the number of iterations exceeds 30

To solve the dirty frequency problem in **Live Save**, we will focus on the first iteration stop condition – the maximum number of iterations – based on the following findings.

According to simulation results, the highest *downtime* tends to fall at the second half of the **medium** dirty frequency phase – indicating iterative transmissions after this second phase no longer support the generation of dirty pages and subsequent iterations may find no page for transmission. As mentioned, the page to be sent in a new iteration will be the dirty pages generated in the previous iteration which do not turn dirty again in the new iteration. Hence, with high dirty frequency, iterations may fail to reduce dirty pages effectively. As simulation results show that system *downtime* usually arrives at the peak at CPU cap25, we assume such high dirty frequency appears after CPU cap25 – *i.e.*, after cap25, the iteration speed will drop below dirty frequency. (Note that the CPU cap value of 25 indicates 25% of CPU usage.)

The results of *total pages sent* and *numbers of iterations* show that iterations will stop when total transmitted memory data multiply the memory size between CPU cap13 and 31. In this cap interval, it is still possible for iterations to reduce dirty pages, *i.e.*, it remains likely to reduce *downtime*. Then, after CPU cap34 when dirty frequency rises to such a high level, it turns unlikely to reduce dirty pages or *downtime* – because hardly any page transmission will take place now: even if there is some page to transmit, it will soon turn dirty again. Iterations hence become redundant and resource-wasting.

3.2.2 The improved design

To solve the dirty frequency problem in **Live Save**, we insert a simple judging measure into original **Live Save** to form a modified **Improved Live Save**. With the judging measure, the modified mechanism can instantly adapt the dirty frequency levels to lessen the high dirty frequency impact. **Improved Live Save** works as follows. When dirty frequency rises to such a high level (say *H*) where iterations cannot effectively reduce dirty pages, the modified mechanism will decrease the maximum number of iterations (MAX_iterations) by 1 after each iteration, to maintain appropriate performance. When

dirty frequency drops below *H*, it will increase MAX_iterations by 1 – but not to exceed the preset MAX_iterations in **Xen**. By the simple practice, we can avoid insignificant iterations in **Live Save** to save overall execution time and total transmitted memory data.

In **Improved Live Save**, we can define *H* as the dirty frequency of an iteration which rises to such a high level that the iteration may produce no less dirty pages than the previous iteration. In fact, such a high level *H* will lead the system to the processing limit and cause iterations unable to reduce dirty pages effectively. That is, with *H*, the current iteration will likely become a useless iteration with no page for transmission. For improvement, when dirty frequency reaches *H*, our modified mechanism will cut MAX_ iterations by 1 at the end of the iteration to avoid such useless iterations. On the other hand, when dirty frequency drops below *H* and there remain chances to reduce dirty pages, it will increase MAX_iterations by 1 at the end of the iteration – not to exceed the preset threshold. (Table 3 gives the pseudocode of the judging measure which can be readily inserted between lines 17 and 18 in Table 2.)

As mentioned in Section 3.2.1, dirty frequency levels (low, medium, or high) are differentiated on the basis of iteration stop conditions, *i.e.*, dirty frequency levels are not fixed and will vary with different disk-writing speed. For instance, in a system with very low disk-writing speed, a moderate dirty frequency may become *H* simply because the system may not catch up with even such a moderate dirty frequency. Note that in a system with fixed disk-writing speed, *H* can be computed initially and be fixed (as the value of *H* results from the dirty frequency of an iteration which rises to such a high level that the iteration may produce more/equal dirty pages than the previous iteration and hence find no page for transmission).

Table 3. The pseudocode of the added judging measure.

1	if dirty frequency > system process limit
2	MAX_iterations = MAX_iterations – 1
3	else if MAX_iterations < Xen's MAX_iterations
4	MAX_iterations = MAX_iterations + 1

4. PERFORMANCE EVALUTION

4.1 The Simulation Environment

Extended simulation runs are conducted to check the performance of various snapshot mechanisms, including **Xen Save**, **Xen Live Migration**, **Live Migration Save (VNsnap)**, **Live Save** and **Improved Live Save**. We carry out the simulation in 2 Xen hosts with Intel Quad-core CPU: 3.30GHz, 4GB memory, Xen-4.5.1 and 3.16.0 Linux kernel (hosted virtualization with para-virtualization) and 1 Network File System (NFS) host [28] with Intel CPU: 3.30GHz, 2GB memory and 3.13.0 Linux kernel. Note that we use 2 Xen hosts and 1 NFS host to run the simulation for **Xen Live Migration** and **Live Migration Save** (as Fig. 2 shows) but only 1 Xen host to run the simulation for **Xen Save**, **Live Save** and **Improved Live Save**.

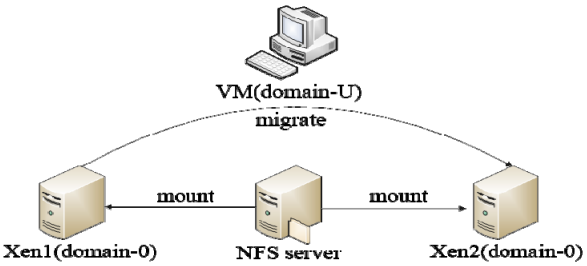


Fig. 2. The migration environment.

In the simulation, domain-U keeps writing data into the memory to make it dirty and the Credit Scheduler [11, 29] will set up CPU cap levels for domain-U. By setting up the CPU cap levels, we can limit domain-U's CPU usage to yield different levels of dirty frequency: CPU cap0 indicates unlimited (100%) CPU usage, whereas CPU cap1~99 indicates 1%~99% CPU usage. When issuing the Save and Migration commands, we use command parameter −vvv to attain more detailed messages and wanted data from the system, including the system downtime, execution time and numbers of data sent. Domain-U has two settings: VM memory 1G (with 1GB memory – data written *randomly* and *continuously* to its 512MB memory) and VM memory 2G (with 2GB memory – data written *randomly* and *continuously* to its 1GB memory).

Regarding simulation results, all data are the average value of 20 runs and the CPU cap interval is set to be 3 (0 (idle), 1, 4, 7…100 (0)). For instance, *downtime* for **Improved Live Save** at CPU cap28 is 6050.55 (Fig. 6 (a)). Given 95% of confidence, the calculated confidence interval half-width over the 20 replications is 168.7258964 – indicating we are 95% confident that the true result will fall between 6050.55±168.7258964 or equivalently 6050.55±2.79% (with less than 3% error). It shows that the simulation results are reasonably accurate, *i.e.*, we can practically exclude possible deviations due to simulation scenarios generated by random numbers.

4.2 Simulation Results

4.2.1 Dirty frequency vs. CPU caps

Fig. 3 (a) gives dirty frequency at different CPU cap values (0%~100%) for the five mechanisms. As we can see, **Save** is without dirty frequency because it directly shuts down the system in the snapshot process. Dirty frequency for the other four mechanisms grows with CPU cap values in a similar rising trend. (Note that in later simulations, we will adjust the CPU caps to get varied dirty frequency in order to observe the corresponding performance and overhead).

4.2.2 Dirty pages in each iteration vs. CPU caps

Figs. 3 (b) and (c) give the numbers of dirty pages generated in each iteration at different CPU caps, respectively for VM memory 1G and 2G. The results show that, for the four mechanisms, dirty pages generated in the iterations all reach the peak after CPU cap 31. We also find that, after this cap value, the obtained dirty pages are close to the

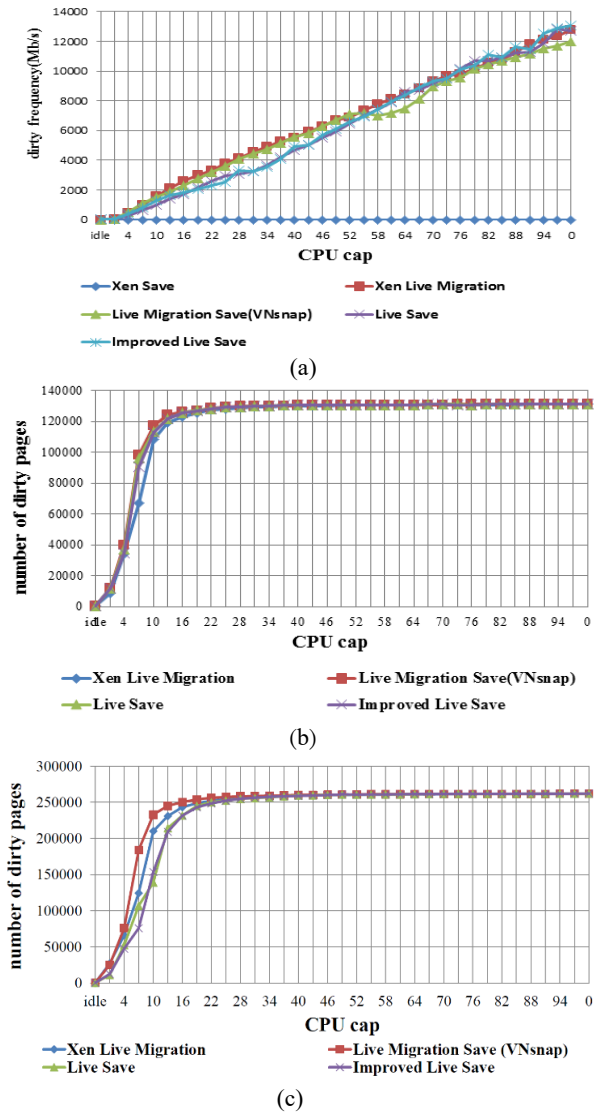


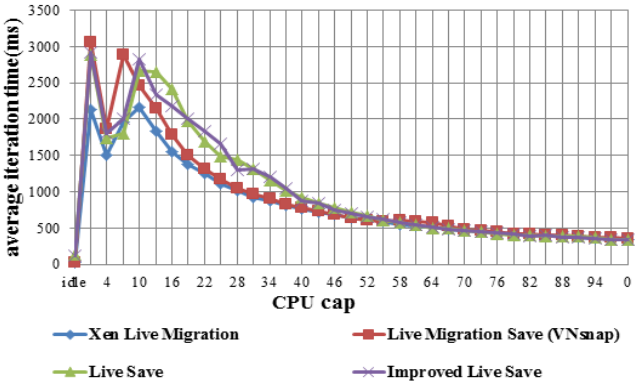
Fig. 3. (a) Dirty frequency vs. CPU caps; (b) Dirty pages/each iteration vs. CPU caps for VM memory 1G; (c) Dirty pages/each iteration vs. CPU caps for VM memory 2G.

size of data written into the memory (131072 pages – close to 512M memory and 262144 pages – close to 1G memory), indicating nearly all memory pages turn dirty in iterations after CPU cap 31.

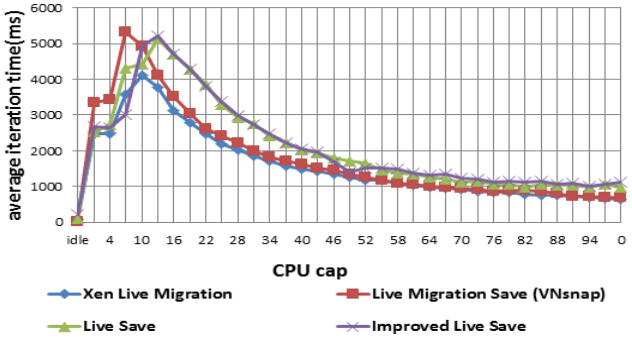
4.2.3 The average iteration time vs. CPU caps

Fig. 4 exhibits the average iteration time, from the second to the last second iterations, at different CPU caps for VM memory 1G (a) and 2G (b). (The first and last itera-

tions are excluded because the first sends all memory data and the last is the stop-and-copy phase.) We observe that the average iteration time rises sharply before CPU cap10 for all mechanisms. This is because, when dirty pages increase in the memory, mechanisms need to transmit increasing pages in each iteration. After CPU cap10, the average iteration time starts to decline. This is because, after the cap value, iterations tend to handle fewer pages at less transmission time. Recall that the pages to be sent in a new iteration are the dirty pages from the previous iteration which do not turn dirty again at the beginning of the new iteration. Hence, when high dirty frequency produces more dirty pages in the memory, it will cause subsequent iterations to skip more dirty pages and meanwhile to transmit fewer pages.



(a) VM memory 1G.



(b) VM memory 2G.

Fig. 4. The average iteration time vs. CPU caps.

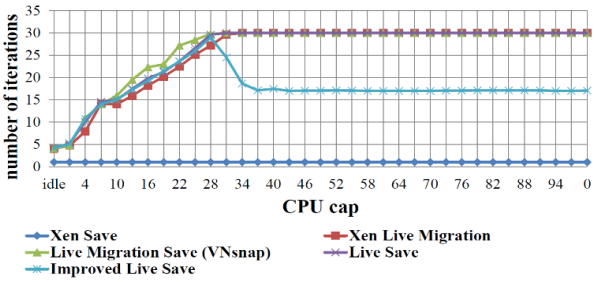
4.2.4 The average number of iterations vs. CPU caps

Fig. 5 depicts the number of iterations at different CPU caps for VM memory 1G (a) and 2G (b). As it shows, the numbers of iterations for **Live Migration**, **Live Migration Save** and **Live Save** all grow with cap values before CPU cap37. After cap37, the three mechanisms maintain a steady trend of 30 iterations to the end (*i.e.*, cap0 (100%)). As for **Improved Live Save**, the number of iterations rises from the beginning, takes a decending turn at cap28 (a) and cap31 (b), and then stays around 17 until the end. (By con-

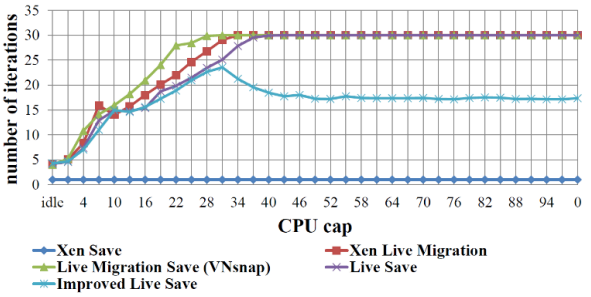
trast, **Save** runs only one iterative transmission.)

We also find the numbers of iterations ascend before CPU cap7. This is because, when dirty frequency rises, we need more iterative transmissions to reduce dirty pages in order to reach the stop condition for the **low** dirty frequency phase: dirty pages from both the previous and current iterations must be less than 50 (Section 3.2.1). The numbers of iterations increase between CPU cap7 and 37 for a different reason: When rising dirty frequency results in more skipped pages and fewer transmitted pages, we must carry out more iterations to get the needed pages in order to meet the stop condition for the **medium** dirty frequency phase: the total transmitted pages exceed the triple of the memory size. After CPU cap37, the number of iterations reaches 30 which is the stop condition for the **high** dirty frequency phase.

Fig. 5 (b) exhibits that the number of iterations for our **Improved Live Save** starts to decrease after CPU cap31. It happens as dirty frequency has now reached such a high level that iterative transmissions can hardly keep up with generation of dirty pages, *i.e.*, it becomes difficult to reduce dirty pages. **Improved Live Save** hence starts to decrease the number of iterations (between cap31 and 40) to 17 – the smallest number of iterations.



(a) VM memory 1G.



(b) VM memory 2G.

Fig. 5. The number of iterations vs. CPU caps.

4.3 Downtime

The results of *downtime* obtained at different CPU caps are shown in Fig. 6 (a) / Table 4 for VM memory 1G and Fig. 6 (b) / Table 5 for VM memory 2G. (Both tables show the results from cap1 to 25 and at idle.) As observed, *downtime* grows with CPU caps before cap25 for all mechanisms except **Xen Save**, with a similar trend: growing slowly before cap4, abruptly between cap4 and 10, and slightly from cap 10 to 25. The

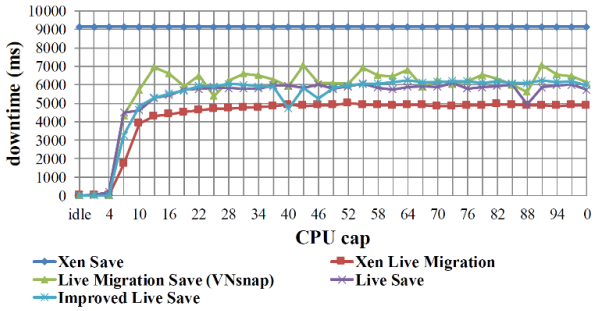
fact that *downtime* reaches the highest after cap25 and remains so to the end (with sporadic variations) reveals that iterations after cap25 can hardly reduce dirty pages.

Table 4. *Downtime* from CPU cap1 to cap25 and at idle (VM memory 1GB).

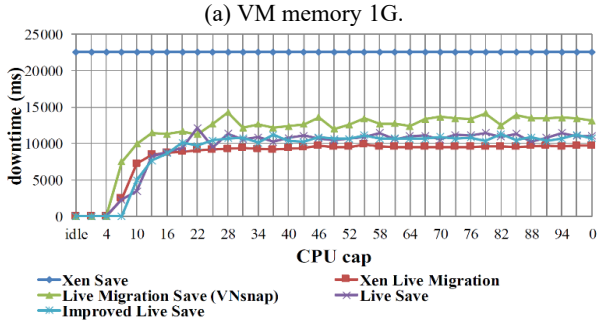
downtime	idle	1	4	7	10	13	16	19	22	25
Save	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5
Live Migration	3.9	4.8	5.1	1715.2	3922.8	4290.35	4413.7	4520.4	4631.45	4698.65
Live Migration Save	3.85	51.1	8.55	4343.15	5728.5	6964.8	6609.7	5909.85	6495.65	5394.4
Live Save	9.7	31.55	192.4	4495.7	4641.45	5279.05	5443	5730.95	5781.55	5855.2
Improved Live Save	7.5	6.9	15.2	3272.05	4825.15	5301.4	5525.35	5677	5928	5919.8

Table 5. *Downtime* from CPU cap1 to cap25 and at idle (VM memory 2GB).

downtime	idle	1	4	7	10	13	16	19	22	25
Save	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5	9151.5
Live Migration	3.9	4.8	5.1	1715.2	3922.8	4290.35	4413.7	4520.4	4631.45	4698.65
Live Migration Save	3.85	51.1	8.55	4343.15	5728.5	6964.8	6609.7	5909.85	6495.65	5394.4
Live Save	9.7	31.55	192.4	4495.7	4641.45	5279.05	5443	5730.95	5781.55	5855.2
Improved Live Save	7.5	6.9	15.2	3272.05	4825.15	5301.4	5525.35	5677	5928	5919.8



(a) VM memory 1G.



(b) VM memory 2G.

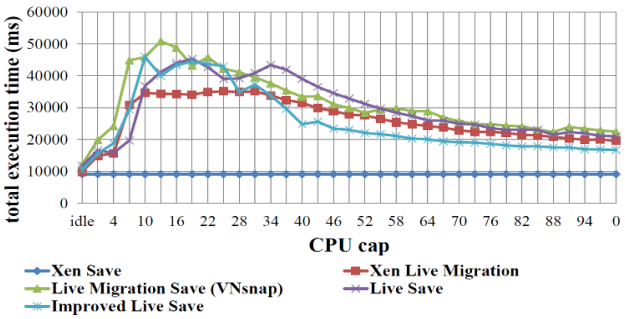
Fig. 6. Downtime.

Among the mechanisms, **Xen Live Migration** produces the shortest *downtime*, followed by **Live Save**, **Improved Live Save**, **Live Migration Save** and **Xen Save**. Unlike other mechanisms which have to generate a snapshot of the VM state data and write it to the hard disk, **Xen Live Migration** sends the state data directly to the memory of the target host. It hence takes less execution time in the stop-and-copy phase, to produce the

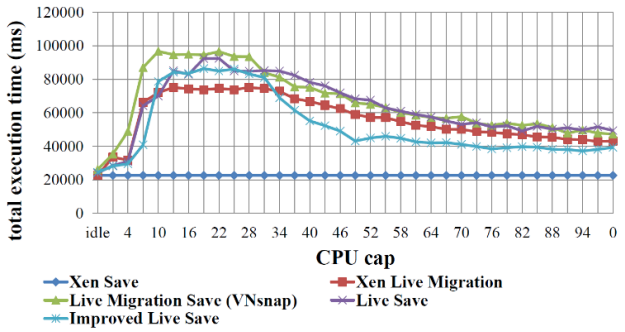
shortest *downtime*. Our **Live Save** and **Improved Live Save** take less *downtime* than **Live Migration Save** (as much as 4000 ms less in Fig. 6 (b)) because we write the VM state data directly to the hard disk of a local host, whereas **Live Migration Save** sends the data to the snapshot daemon of the target host and later writes the received data from memory to the hard disk. **Xen Save** needs the longest *downtime* because, instead of iteratively sending the state data in advance, it will shut down the VM first before engaging VM migration.

4.4 Total Execution Time

Fig. 7 depicts the *total execution time* at different CPU caps. We see *total execution time* grows with cap values before CPU cap10 for all mechanisms except **Xen Save** – because the mechanisms must engage more iterations to send the increased dirty pages when dirty frequency rises. Between CPU cap10 and 28, *total execution time* reaches the top and remains there with slight variation. The level *total execution tim*e is the result of decreasing execution time and rising *downtime* offsetting each other in this cap interval.



(a) VM memory 1G.



(b) VM memory 2G.

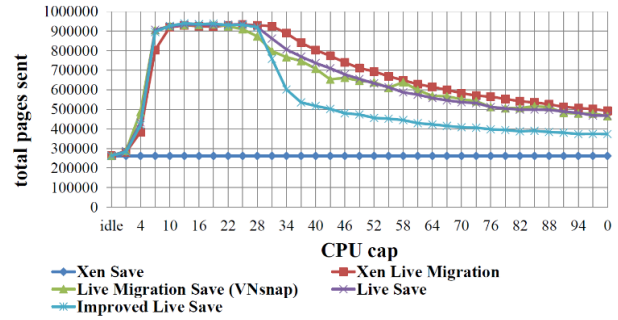
Fig. 7. Total execution time.

Total execution time decreases after CPU cap28 as *downtime* stops growing after reaching the peak and execution time continues to decrease in iterations. In all mechanisms, **Xen Save** directly shuts the VM down without iteratively sending memory pages and hence yields the least *total execution time*. **Xen Live Migration**, **Live Migration Save** and **Live Save** display a similar trend in *total execution time*, with **Xen Live Mi-**

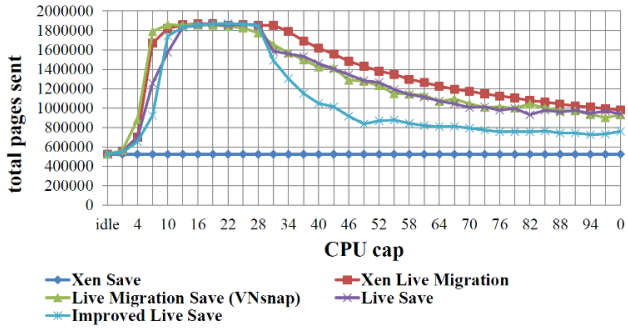
gration performs slightly better because it does not write the state data to the hard disk. In contrast to the three mechanisms, our **Improved Live Save** yields significantly reduced *total execution time* (up to 20000 ms reduction in Fig. 7 (b)), mainly because its practice effectively reduces both the number of iterations and execution time.

4.5 Total Pages Sent

The results of *total pages sent* are plotted in Fig. 8. Before CPU cap10, *total pages sent* increases with CPU caps for all mechanisms (except **Xen Save**). This is because, when dirty frequency rises, each mechanism will take more iterations to reduce dirty pages in order to meet the stop condition for the **low** dirty frequency phase. From cap10 to 28, *total pages sent* reaches the same peak at each cap. It stops growing with cap values because all have reached the iteration stop condition for the **medium** dirty frequency phase. Beyond cap28, when high dirty frequency generates more skipped pages and less transmitted pages, *total pages sent* keeps decreasing. Of the mechanisms, **Xen Save** yields the least *total pages sent* because it sends all memory pages in one transmission. **Xen Live Migration**, **Live Migration Save** and **Live Save** turn out quite similar results, with **Xen Live Migration** producing slightly more *total pages sent* (as it does not write the state data of a VM to the hard disk). Our **Improved Live Save** starts to function after cap28. By instantly adapting the dirty frequency levels, it effectively reduces both *the number of iterations* and *total pages sent* (a significant reduction of more than 100000/200000 pages is shown in Figs. 8 (a) and (b)).



(a) VM memory 1G.



(b) VM memory 2G.

Fig. 8. Total pages sent.

4.6 An Overall Recap

To facilitate illustration, we list the features of **Xen Save**, **Live Migration Save** and **Live Save** in Table 6 and recap key performance/cost comparisons for **Xen Save**, **Live Migration Save**, **Live Save** and **Improved Live Save** in Table 7.

Table 6. Basic features.

	Xen Save	Live Migration Save	Live Save
Live	Non live	Live	Live
VM downtime	From backup to the end	The stop-and-copy phase	The stop-and-copy phase
Destination of VM state data transfer	Specified file location in local host	The target host (the file not retained)	Specified file location in local host

Table 7. Key comparations.

	Xen Save	Live Migration Save	Live Save	Improved Live Save
Downtime	High	Medium	Low	Low
Total execution time	Low	High	High	Medium
Total pages sent	Low	High	High	Medium
Network banwidth consumption	No	Yes	No	No

5. CONCLUSION

As mentioned, we can adopt snapshot techniques to backup the state files of VMs and use the backup files to recover faulty VMs and maintain proper system performance. In cloud systems, it is of particular importance to attain the snapshot files by the *least* downtime in order to minimize the shutdown impact and preserve services. In this paper, we present two new snapshot mechanisms – **Live Save** and **Improved Live Save** – to enhance the performance of existing snapshot mechanisms, especially to reduce their system downtime and network bandwidth consumption. Different from existing mechanisms, our **Live Save** sends VM state data in iterations and stores the snapshot file in a local host, instead of a remote one, to obtain the backup files with *no* network bandwidth consumption. When it is necessary to send the file to a remote host, **Live Save** will directly send the generated complete file to the remote host after iterations end – to exclude the iterative transmissions in **Live Migration Save** and save network bandwidth. To avoid redundant iterations due to high dirty frequency in **Live Save**, we set a judging measure to form a modified **Improved Live Save**. The modified mechanism handles the dirty frequency problem by keeping the maximum number of iterations, as follows. When dirty frequency rises to such a high level that iterations cannot effectively reduce dirty pages, **Improved Live Save** will decrease the maximum number of iterations by 1 at the end of each iteration; it will increase the number by 1 when dirty frequency drops below the level. By adjusting the allowed maximum number of iterations, the modified mechanism is able to avoid futile iterative transmissions in **Live Save**, saving total execution time and transmitted memory data. Extended simulation runs are carried out to check the performance of different snapshot mechanisms, including ours. The results demonstrate that both **Live Save** and **Improved Live Save** take less *downtime* than related mechanisms, thanks to the distinct design: to *live* back up the VM state to a local

host without iteratively sending the memory data in the Internet. In addition to no network bandwidth consumption and less *downtime*, **Improved Live Save** is also shown to outperform others in *numbers of iterations*, *total execution time* and *total pages sent* – mainly because it handles the maximum number of iterations to avoid futile iterations.

REFERENCES

1. Virtualization, https://en.wikipedia.org/wiki/Virtualization.
2. Virtualization spectrum, http://wiki.xen.org/wiki/Virtualization_Spectrum.
3. K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization, " in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 2-13.
4. N. Jain and S. Choudhary, "Overview of virtualization in cloud computing," in *Proceedings of Symposium on Colossal Data Analysis and Networking*, 2016, pp. 1-4.
5. VMware virtualization software, http://www.vmware.com/tw.
6. Paravirtualization (PV), http://wiki.xen.org/wiki/Paravirtualization_(PV).
7. Kernel based virtual machine, http://www.linux-kvm.org.
8. QEMU, http://wiki.qemu.org/Main_Page.
9. H. Tan, C. Li, Z. He, K. Li, and K. Hwang, "VMCD: A virtual multi-channel disk I/O scheduling method for virtual machines," *IEEE Transactions on Services Computing*, Vol. 9, 2016, pp. 982-995.
10. W.-Z. Zhang, H.-C. Xie, and C.-H. Hsu, "Automatic memory control of multiple virtual machines on a consolidated server," *IEEE Transactions on Cloud Computing*, Vol. 5, 2017, pp. 2-14.
11. P.-J. Chuang and C.-Y. Chou, "SRVC: An efficient scheduler for concurrent virtual machines over the Xen hypervisor," *Journal of Applied Science and Engineering*, Vol. 20, 2017, pp. 355-365.
12. A. Kangarlou, D. Xu, P. Ruth, and P. Eugster, "Taking snapshots of virtual networked environments," in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, 2007, pp. 1-8.
13. Xen project, http://www.xenproject.org/.
14. D. Chisnall, *The Definitive Guide to the Xen Hypervisor*, Prentice Hall, NJ, 2007.
15. P. Barham, *et al.*, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 167-177.
16. Red Hat, Inc., "LVM architectural overview," https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Logical_Volume_Manager_Administration/LVM_definition.html.
17. Citrix Systems Inc., "Xen 3.0 virtualization user guide," http://bits.xensource.com/Xen/docs/user.pdf.
18. P.-J. Chuang and W.-C. Wong, "Generating snapshot backups in cloud virtual disks," in *Proceedings of IEEE 17th International Conference on Computational Science and Engineering*, 2014, pp. 1860-1863.
19. A. Kangarlou, P. Eugster, and D. Xu, "VNsnap: Taking snapshots of virtual networked environments with minimal downtime," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, 2009, pp. 524-533.

20. A. Kangarlou, P. Eugster, and D. Xu, "VNsnap: Taking snapshots of virtual networked infrastructures in the cloud," *IEEE Transactions on Services Computing*, Vol. 5, 2012, pp. 484-496.
21. Migration, http://wiki.xen.org/wiki/Migration.
22. 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 Symposiun on Networked Systems Design and Implementation*, 2005, pp. 273-286.
23. H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *Proceedings of IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1-10.
24. C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2013, pp. 41-50.
25. A. Choudhary, M. C. Govil, G. Singh, L. K. Awasthi, E. S. Pilli, and D. Kapil, "A critical survey of live virtual machine migration techniques," *Journal of Cloud Computing Advances, Systems and Applications*, Vol. 6, 2017, pp. 1-41.
26. M. Noshy, A. Ibrahim, and H. A. Ali, "Optimization of live virtual machine migration in cloud computing: A survey and future directions," *Journal of Network and Computer Applications*, Vol. 110, 2018, pp. 1-10.
27. F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: challenges, techniques, and open issues," *IEEE Communications Surveys and Tutorials*, Vol. 20, 2018, pp. 1206-1243.
28. Network file system, https://en.wikipedia.org/wiki/Network_File_System.
29. Credit scheduler, http://wiki.xensource.com/wiki/Credit_Scheduler.

Po-Jen Chuang (莊博任) received the B.S. degree from National Chiao Tung University, Taiwan, in 1978, the M.S. degree in Computer Science from the University of Missouri at Columbia, U.S.A., in 1988, and the Ph.D. degree in Computer Science from the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, U.S.A. (now the University of Louisiana at Lafayette), in 1992. Since 1992, he has been with the Department of Electrical and Computer Engineering, Tamkang University, Taiwan, where he is currently a Professor. He was the Department Chairman from 1996 to 2000. His main areas of interest include parallel and distributed processing, fault-tolerant computing, mobile computing, network security, cloud computing, software defined networking, virtualization and internet of things.

Yen-Chia Huang (黃彥嘉) received his B.S. and M.S. degrees in Electrical and Computer Engineering in 2013 and 2016 from Tamkang University, Taiwan. He is currently with X-LEGEND Entertainment Corp. in Taiwan. His research interests include parallel and distributed processing, virtualization and cloud computing.