Cuckoo Migration: Self Migration on JointCloud Using New Hardware Features

Ruifeng Liu, Zeyu Mi
Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University
Email:hustliuruifeng@gmail.com, yzmizeyu@gmail.com

***Abstract*—With the growing number of cloud providers and the increasing market of cloud computing, it's more and more necessary to realize the cooperation and aggregation of clouds. JointCloud is a framework aiming at facilitating the consociation of various clouds on the market. And the JointCloud Collaboration Environment (JCCE) is the ideal environment for global cloud providers. To realize the clouds cooperation, migration is one of the most important issues that have to be considered. Live migration has always been one of the major primitive operations of virtualization and has been discussed for long. Traditional works deal the migration mainly by using a host-driven migration method, the majority work of which is dominated by the hypervisor. However, as the age of cloud aggregation comes, traditional methods show their defects. In cloud aggregation environment, cloud providers may refuse to supply the migration service to grasp their customers, or the hypervisors are heterogeneous on the two sides of migration. Those problems raise challenges to traditional host-driven methods.**

In this paper, we propose Cuckoo Migration, a new self-migration method using Intel new hardware feature. We leverage a special processor function, VMFUNC, to create a two-EPT architecture for the guest VM, so that the guest has a mirror memory space, which can be used as the duplication of memory.

This paper mainly introduces how we build a two-EPT architecture for the guest and discusses how we leverage such architecture to do our self-migration.

I. INTRODUCTION

Cloud 1.0 realizes the cloud computing, significantly aggregating the large-scale IT resources into one single cloud provider. Previously, a company needs to prepare its own machines and hire a group of people to manage these machines so that it can start the Internet business. Nowadays, those preparation become an online service provided by various clouds providers. Hardware resources are intensively managed by clouds providers, which greatly enhance the utilization of computation infrastructures.

However, as the popularization of cloud computing and the increasing number of cloud providers, Cloud 1.0 shows some weak points, such as unbalanced workloads on different host machines, unreasonable price floating, and the dilemma of choosing clouds considering special cloud services. All those problems cause customers to desire a Cloud 2.0, which consolidate various existing clouds.

JointCloud is a conceptual prototype of Cloud 2.0. JointCloud strives for the collaboration environment of global cloud providers. The main idea of JointCloud is that clouds can share their tasks, resources or even services, which require an effective communication method among clouds. And one goal of JointCloud is to establish a sort of standards and a global environment to make communication between clouds as easy as network transmission between machines. Another goal of JointCloud is to construct a block-chain based environment, JointCloud Collaboration Environment (JCCE), capable of adjusting the workloads and resources utilization dynamically.

To realize the dynamic workloads adjustment and resources reassignment, we have to solve the problem of task sharing. One of the best choices is migration. Live migration has been discussed by predecessors in many prior works [2], [4], [8], [1], [11]. Most of them are host-driven migration. Host-driven migration methods require customer (VM owner) initially notify the hypervisor that they need to migrate. Both sides of the hypervisor (the source host and the target host) cooperate to accomplish the whole process of migration. Doing the majority of migration work by hypervisor has some drawbacks, which cause host-driven migration not suitable for the JointCloud Collaboration Environment. We conclude them as follow:

Firstly, some providers may refuse to offer migration service. Even though different cloud providers collaborate with each other in JointCloud, a cloud provider may not be willing to see customers migrate their VM to other places because that means they may lose a customer. This situation may become more severe in the early stage of JointCloud, where some clouds participate in the JointCloud and others do not. Therefore, the host-driven migration is not suitable for such scenario.

Secondly, cross-cloud migration is complicated. Delegating all migration work to the hypervisor means that the migration module is embedded in the hypervisor. Different cloud providers may have different implementations on migration because of the heterogeneous hypervisors they use. Migration between the same cloud's host machines is simple because the VM is running on the identical hypervisors on the source and target sides. But migration between different clouds' machines is much more difficult. We have to make great efforts to mitigate such diversity in the migration implementation of different clouds. But the number of the existing cloud providers is too large to do so for every two clouds is enormous and unacceptable.

Lastly, in perspective of security, host-driven migration has difficulties in verification. VMs' memory is a confidential region. But during the migration, leakage of guest data may result from vulnerabilities on either side's host. It's not hard to verify a specific cloud's migration code. But when it comes to

the scale of all clouds' migration code, things become intricate. In fact, it will be safe enough only if we verify migration procedure between every two clouds.

To address the migration issues in JCCE, we present Cuckoo Migration [1]. Cuckoo migration is a new self migration method, moving the migration work from the hypervisor space to the guest kernel space. Self migration has been introduced in previous works [2], [3]. However, traditional self-migrations either abort a significant amount of guest memory data as last state III-B, or are complex and fallible in implementing.

Cuckoo Migration is mainly based on the iterative pre-copy methods of live migration [2]. The most basic difference is that live migration [2] is a host-driven migration, instead, Cuckoo Migration is a self-migration. What's new in our design is that we construct a two-EPT architecture to mirror the guest's memory, to deal with some problem we should consider in self-migration.

We may require some basic support of hypervisor, we will describe it in III-E. And the specific migration flow will be described in III-D.

In the following pages, JCCE will be introduced as background in Section II; Our main design of Cuckoo Migration will be described in Section III; More details in our design will be described in Section IV-C; Some preliminary results are shown in Section V; Finally, the comparison between previous related works and our work is discussed in Section VI.

II. Background

JointCloud [10], a new generation of cloud computing model, as a trial solution of Cloud 2.0, aims to create a cooperative environment for global cloud providers. In the design of JointCloud, clouds share resources, and computing tasks. Any cloud in the collaboration environment is allowed to use other clouds' resources as long as it pays the price. Busy clouds can use idle cloud's resources while those resources are creating commercial profit. Meanwhile, it releases busy clouds' burden by task-sharing with the support of collaboration environment. By this way, JointCloud draws the blueprint of a harmonious ecosystem of cloud providers. To enable the cooperation of various clouds, JointCloud proposes a sort of brief software definitions. Those definitions can describe computing resources on different clouds based on different infrastructures or running different hypervisors. The core of JointCloud are two disparate parts:1) The JointCloud Collaboration Environment (JCCE), which contains several BlockChain-Based services such as resource exchanging, resource registration or supervisor. 2) the Peer Collaboration Mechanism (PCM), aka the sort of software definitions. PCM works like network protocol, coordinating the cooperation among clouds, providing information for other clouds in JCCE and providing standard open APIs.

[1] **Cuckoo** is a kind of birds, most of them never build a nest for their children. Instead, they lay eggs in other birds' nest, leaving their baby as other birds' kids. Until little cuckoo grew up, their adoptive parent will realize the fact. Our migration method names cuckoo migration because we adopt the same strategy.

III. Design

A. Overview

Hypervisor maintains guest memory by a two-dimensional paging mechanism. Guest maintains a guest page table to translate Guest Virtual Address (GVA) to Guest Physical Address (GPA). While hypervisor maintains an Extended Page Table (EPT) [2] to map GPA to Host Physical Address (HPA). Shown in Figure 1.

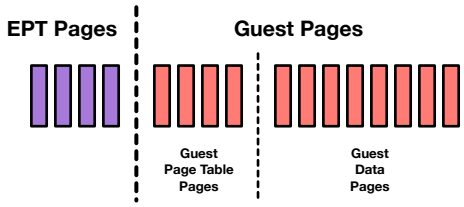

Fig. 1. Two-Dimensional Pageing Architecture

In Figure 1, each block represents a real memory page on the host machine. Guest can only access guest pages (red blocks), and is unaware of the existing of EPT pages. Guests access a guest page via GPA, and the hypervisor translates the GPA to HPA by walking EPT. Therefore, all guest data is stored in guest pages, while EPT pages store the information about guest pages' location on machine.

To migrate a VM from one machine to another, we have at least to ensure all the guest pages are copied from the source to the target host, so that to keep the VM's memory state is the same before and after migration.

Currently, the most popular way to achieve self-migration is to build a pseudo kernel on the target node at first, and then copy guest pages from the source node to the target node, finally, when the target node received all origin guest pages, it will swap its memory, using the source node's guest pages and discarding pseudo kernel guest pages. Figure 2 briefly conclude this process.

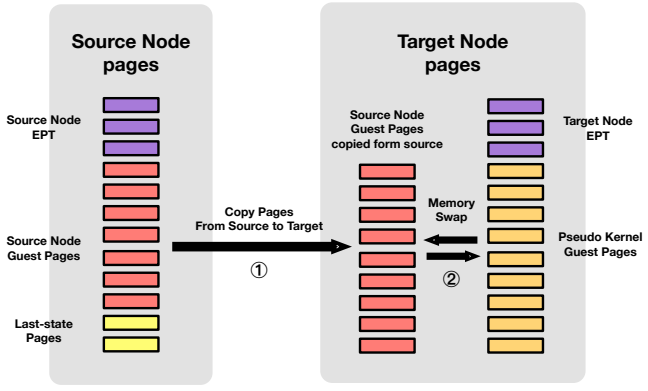

Fig. 2. Self-Migration via Swapping a Pseudo Kernel

[2] called EPT in Intel, and Nested Page Table (NPT) in AMD

B. Chanlleges for Self-Migration

In normal host-driven migration, hypervisors do the page copying jobs. Because hypervisors are isolated with guests, they can pause the guest to forbid any modification on origin guest pages, and then copy guest pages. But in self-migration, a guest can not pause itself. So it incurs challenges both on the source side and the target side.

We know that once a process is running, it modifies the heap and the stack on memory, or may write data on other memory regions. That is to say, a page belongs to a running process will keep changing. On the source side, there are some pages keep changing as if the migration is processing, we call them last-state pages (yellow blocks in Figure 2), which can not be transformed easily.

When target node received all the guest pages, it needs to remap each guest page's HPA with correct GPA, because all the GPA in guest pages do not change while the HPA of each guest pages changed. Guest can not directly manipulate the EPT, it causes trouble, too.

C. Two-EPT architecture

It's hard to accomplish self-migration by a single kernel. To deal with this problem, we create a mirror of guest's whole memory by forking another EPT, to build a two-EPT architecture for the guest kernel. In this way, the guest operate as if it has a duplication of memory. In the remain of this section, we will detail how to mirror the guest memory and build a two-EPT architecture.

VM Function. We leverage the Intel new hardware instruction VMFUNC to achieve fast switching between two EPTs. Under the Intel Virtual Machine eXtensions (VMX) architecture, a CPU runs in two modes. In the guest context, CPU runs in non-root mode, and in the hypervisor context, it runs in root mode. Some privileged operations can only be executed in the root mode, so we often need to switch CPU between these two modes. VMFUNC instruction provides VM Functions for non-root guests to directly execute certain root operations without context switching. It saves time especially where the program needs frequently switching CPU between two modes.

EPTP-switching. EPTP-switching is one of the VM Functions. Guests can invoke EPTP-switching to switch the EPT root pointer without trapping into hypervisor space. It doesn't mean that a guest is able to arbitrarily modify the EPT root, instead, hypervisors provide some options for guests, and guests switch the EPT root by choosing an index.

Virtualization Exception. When hypervisors fail to translate a GPA to HPA by walking through the EPT, or the guest does not have sufficient rights to access the physical page, it causes an EPT Violation. Generally, EPT Violation is invisible to the guest. But we can leverage the hardware technology, Virtualization Exception (VE), to let the guest know a hypervisor-level exception once it happens, and trigger the exception handler in guest kernel.

EPT Forking. In our design, we allocate two EPT slots for every guest VM, EPT-0 and EPT-1 shown in Figure 3.

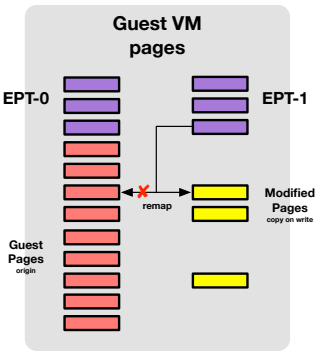

Fig. 3. Fork Guest EPT.

The main propose of forking EPT is to create an isolated guest memory space. So that we can reserve all the original guest pages and work on their duplications. To fork the guest EPT, there are following steps: 1) Allocate another EPT (mirror EPT) completely same as the original EPT (origin EPT) atomically, so that the mirror EPT has the same mapping of GPA to HPA as origin EPT at the beginning. 2) Switching to mirror EPT using VMFUNC, and set all guest pages read-only in the EPT entries (Freezing the origin EPT later described in IV-A2). 3) When we need to do modification on a frozen guest page, it causes an EPT Violation. To handle this EPT Violation, we create a duplication of this page and change the mapping in mirror EPT, then we can modify the duplication page while reserving the origin guest page. Meanwhile, we notify the guest kernel of this EPT Violation by VE, so that the guest is able to know that a page belongs to it has been duplicated. Allocating mirror EPT and creating duplicated guest pages require the support of hypervisor IV-A2, and the rest of things can be done by guest itself.

D. Cuckoo Migration Workflow

In this section we describe our migration workflow combined with Figure 4

In step one, we need to launch a VM on the target node, and operate a pseudo kernel on the EPT-1, then, we fork an EPT-0 to contain the upcoming guest pages from the source node. On the source node, we let origin OS run on the EPT-0, and fork an EPT-1 as mirror EPT.

In step two, we copy guest pages by iteration [2]. In the first iteration, We copy all the guest pages to the target node while keeping origin system running. Obviously, some guest pages changed during our first iteration. We mark these pages and in the second iteration, we copy these marked pages again , covering the outdated pages on the target node. But during our second iteration, some pages changed again, then we continue to do the third iteration. Finally, when the working set is small enough, we do the final iteration copy, copying all the rest marked pages.

During the step two, we exchange roles of the two EPTs in every iteration. In the first iteration, we let EPT-0 to be the origin EPT, keeping it stable, and we process on the EPT-1. All modification on guest pages will be copy-on-write. The

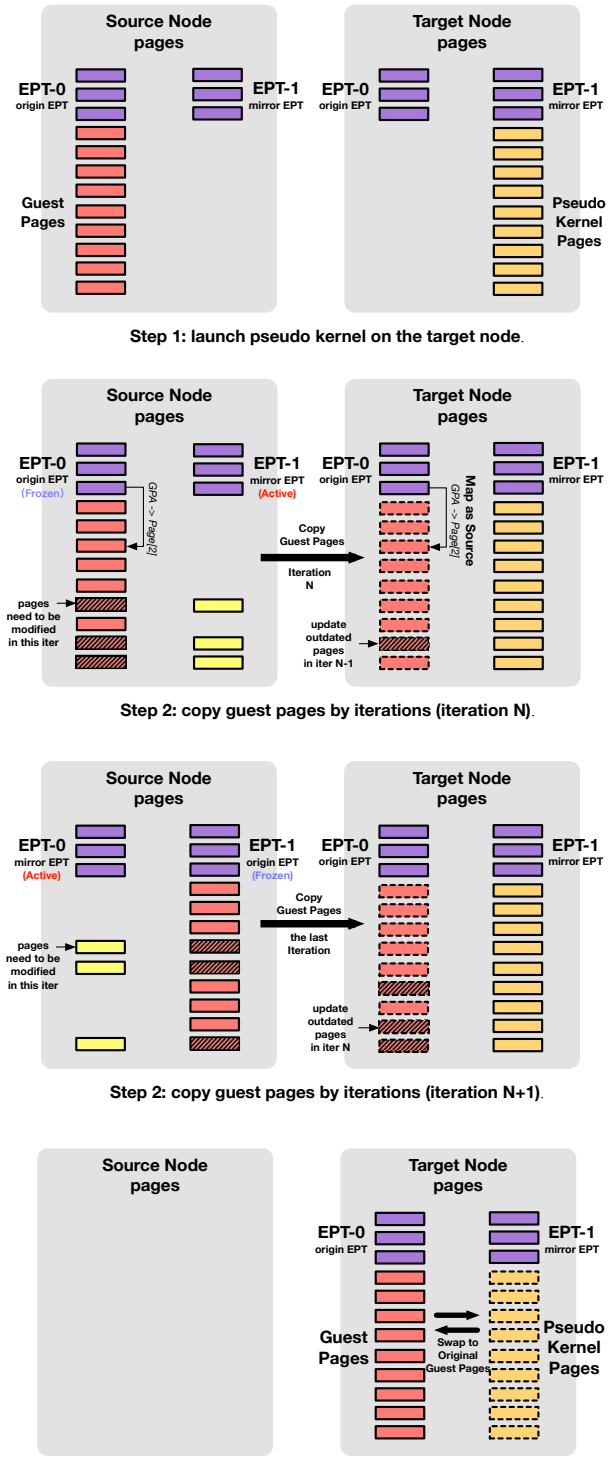

Fig. 4. Cuckoo Migration Workflow.

duplication will be created and remapped on the EPT-1. And in the second iteration, cause we don't suspend the guest, EPT-1 now stores the newest guest memory state. So we let EPT-1 to be the origin EPT, freezing it and do works on the EPT-0 which acts as the mirror EPT in this iteration. When the second iteration ends, the newest memory state will be on the EPT-0, and we exchange their roles again. The rest of iterations can be done in the same manner. The guest knows which of its page has been changed by VE, when a duplicated page is created, it will cause a VE to inform the guest, too.

In step three, the target node has already received all the guest pages, that is, all the guest data got ready on the target node. Then, we reload the running state as the source node record before the final copy iteration begins. Afterwards, we can resume the VM on the target node. Finally, we shutdown the source node and free the pseudo kernel together with EPT-1, and the migration is done.

The time since the final iteration begins, till the VM resumes on the target node, is the downtime of this migration. During downtime, the guest is paused, only remaining some duplicated components continuing the migration work.

E. Precondition

To ensure our Cuckoo Migration working, we have some preconditions. We need some hardware and software support.

- Both the source node and the target node should be equipped with enough advanced Intel Processor which provides the VM Function. And the VMFUNC is enabled on both sides, the EPT list is deployed on both sides.
- Because Cuckoo Migration is self-migration, we need a method to remote launch and shutdown a VM, to ensure the proper launching of the target VM and the proper shutdown of the source VM. Here we have to require the JCCE providing proper interfaces. We abstract our demands in the Table I.
- To support the VMFUNC and let VMFUNC work better, we require the hypervisor to provide a sort of assistant hypercalls. We will detail those hypercalls in section IV-A2.

TABLE I
JCCE MIGRATION APIS

Command and Parameters	Description
LaunchRemoteVM(cloud_id, VM_config)	launch a target VM on *cloud_id* with the configure depicted in *VM_config*
ShutdownRemoteVM(cloud_id, vm_id)	shutdown the VM *vm_id* on the *cloud_id*
CheckState(cloud_id, vm_id, &vm_state)	check the status of *vm_id* on the *cloud_id*, write vm status in *vm_state*

F. Reliability

Crash. During the migration, anyone of the two sides may crash. If crash happens on the source node, it means that we loss all the origin guest pages, the migration will fail. If crash happens on the target node, it doesn't matter because no guest data is lost, we can start migrating all over again or stop and give up the migration.

Integrity. Data integrity also need to be taken into consideration. Although we use TCP protocol to transform guest pages, and the TCP protocol guarantees the data integrity itself, data

bit flipping may happened during the vast memory writing. Bit flipping in guest pages may result in serious consequences, so it's necessary to check all guest pages' integrity before swapping the pseudo kernel. We plan to hash guest pages' data to make a brief signature, and ensure the data integrity by checking the signatures on the source and target nodes are consistent.

IV. IMPLEMENTATION

In this section, we will briefly introduce every component's duty in Cuckoo Migration. To make our idea easier to understand, we may show some pseudo-code when necessary.

A. Hypervisor Support

As we described in the Design section III, hypervisors need to provide both hardware support and software support.

1) Hardware Support.: The hardware support is that both the source node and the target node should be equipped with enough advanced Intel processor with VMFUNC instruction, besides, hypervisors need to enable VM Functions by setting certain bits in Virtual Machine Control Structure (VMCS). Every VM has its own VMCS. VMCS is used to properly configure the vcpu and execution context. To enable EPTP-switching, some fields in VMCS are involved. First, we have to set bit 13 in *SECONDARY_VM_EXEC_CONTROL* to enable VM Functions, and set bit 0 in *VM_FUNCTION_CTRL* to enable EPTP-switching. Then, hypervisor need to prepare an EPT Pointer (EPTP) list, where each item is a selectable EPT root's HPA, and fill the EPTP list's HPA in the *EPTP_LIST_ADDRESS* field. Above is what we have to do to configure EPTP-switching. Guest invoke EPTP-switching by execute VMFUNC instruction and let *EAX = 0* (EPTP-switching index of VM Functions) and let *ECX = EPTP_index*.

2) Software Support.: Hypervisors need to provide some general functions to assist our Cuckoo Migration. It doesn't obey the principle that guests doing migration itself, because these functions are very basic and little related with our main migration logic. To provide software support to assist our migration, we ask hypervisor to supply following hypercalls (listed in Table II) as preconditions:

HYPERCALL_FORK_EPT. One of the hypercalls we required is *HYPERCALL_FORK_EPT*. Guest can invoke this hypercall whenever it wants to fork a mirror EPT from an origin EPT. Two parameters are required, the first *src_ept_idx* is the index of the source EPT root in the *EPTP_LIST*, and the second *dst_ept_idx* is the index of the destination EPT. Deep copy needs to dynamically allocate memory for mirror EPT. And during EPT forking, the source EPT needs to be locked, to ensure the atomicity. If an EPT already exists on the *dst_ept_idx*, then reconstruct it.

HYPERCALL_FREEZE_EPT. Moreover, we need the hypercall *HYPERCALL_FREEZE_EPT* to freeze an EPT. An EPT is frozen means that all the guest pages being mapped by it can no longer be changed. To achieve this, we need hypervisor to assign a freeze-bit in each leaf EPT entry. Once an EPT entry's freeze-bit is set, it means the guest page it points to shouldn't be changed. This hypercall require one parameter that is *ept_idx*, indicating the index of the EPT to be frozen. *HYPERCALL_FREEZE_EPT* will travel the tree rooted at the *EPTP_LIST*[*ept_idx*] and set freeze-bit of all leaf nodes. If somebody intends to modify frozen guest pages, it will cause an EPT Violation. In the responding EPT Violation handler, a duplicated page will be created and correctly mapped as shown in Figure 3.

HYPERCALL_FETCH_PAGE.

Sometimes we may need to read data from an EPT while processing on another, but a processor can only map GPA to HPA by either of the two EPTs. To map a page belongs to another EPT, we need the help of hypervisor. Here we require the hypercall *HYPERCALL_FETCH_PAGE* to achieve cross-EPT mapping. The purpose of the hypercall is to find a guest page by going through the source EPT on the GPA *src_paddr*, and map it to the target EPT on the *dst_paddr*. In this way we can share pages between different EPT domains.

TABLE II
HYPERCALL LIST

Name	Parameters	Description
FORK_EPT	*src_ept_idx*	index of the source EPT
	dst_ept_idx	index of the mirror EPT
FREEZE_EPT	*ept_idx*	index of EPT to be frozen
FETCH_PAGE	*src_ept_idx*	index of the source EPT
	src_paddr	GPA of the page on source EPT
	dst_ept_idx	index of the mirror EPT
	dst_paddr	GPA of the page on mirror EPT

B. Source Node Migration Process

The duty of source node in Cuckoo Migration is to send guest pages together with their mapping information (responding GPA in the EPT) to the target node. We switch the roles of two EPT in every pre-copy iteration. For example, in the first iteration, EPT-0 is the stable EPT which is frozen, and we work (modify guest pages) on the mirror EPT-1. And in the second iteration, EPT-1 becomes the stable EPT to be frozen, and we work on the EPT-0. As when the first iteration ends, the latest memory state will be reserved in EPT-1. We exchange the two EPT's roles in every iteration. And in the last iteration, we only send the stable EPT's guest pages, while the mirror EPT's state will be discarded. Pseudo code listed in Listing 1 briefly conclude the flow.

```
#define ORIGIN_EPT(i) ((i)%2)
#define MIRROR_EPT(i) (((i)+1)%2)

//the ith copy iteration.
copy_iteration(int i)
{
    FORK_EPT(ORIGIN_EPT(i), MIRROR_EPT(i));
    FREEZE_EPT(ORIGIN_EPT(i));
    vmfunc(MIRROR_EPT(i));

    gpa_t page_addr;
```

```
    gpa_t buffer;

    for_each_page(&page_addr) {
        //travel every guest page

        FETCH_PAGE(ORIGIN_EPT(i), page_addr
                   MIRROR_EPT(i), buffer);
        TCP_send_data(buffer, PAGE_SIZE);
    }

}
```

Listing 1. Pseudo-code of Copy Iteration

C. Target Node Migration Process

On the target node, pseudo kernel is working on the mirror EPT-1, and we let origin guest finally run on EPT-0. We firstly received all guest pages from the source node, and map them on the EPT-1, that is to say, we can get the page we want by walking the EPT-1. Then, we begin to fake an equivalent EPT-0 on the target node. The reason why we need to fake an EPT-0 instead of directly copying EPT-0 from the source node is that the HPA of each guest page will be different between it on the source node and on the target node. So we have to make proper adjustments on the target node's EPT-0. We fake EPT-0 by invoking hypercall *HYPERCALL_FETCH_PAGE* to map address correctly. After EPT-0 and all the source guest pages are fully constructed, EPT-0 becomes master and begins operating the origin guest.

V. PRELIMINARY RESULTS

Cuckoo is in its early stage of implementation and we will report some microbenchmark result in this section. For a network virtualization exception, we have installed the VE handler into the EPT-0 in the source VM and it is responsible for emulating the source node sending a page out. To observe the overhead of copying a page, we launched two VMs on two separated physical machines connected with a local network. Both of the two machines are equipped with Intel Core i7-6700 (8 cores). Each VM owns 8 vcpus, and connects to the Internet by net-bridge. Two machines are connected with 100M-bit LAN. And the Table III gives us a review of how sending a 4K/2M page will cost.

TABLE III
AVERAGE TIME CONSUPTION OF 1000 NETWORK VIRTUALIZATION EXCEPTION

	4K (μs)	2M (μs)
network transmission	1013.52	22432.13
network transmission + hypercall	1120.74	22506.41

VI. RELATED WORK

Live migration has been one of the hot topics of virtualization for long, but most of them are host-driven migration [2], [4], [9], [8]. Self-migration has been slightly metioned in [2], [4], [3], but they throw a large amount of guest pages as last-state pages III-B, and is complex in implementing. Heterogeneous migration has been discussed in [5], but it's not practical when there are such various hypervisors. Thinking in a docker way [6], [7], we can eliminate the heterogeneous problems by adding another abstraction, but it must introduce extra overhead not just in migration.

VII. CONCLUSION

In conclusion, our main contribution is to propose a two-EPT architecture of guest VM to deal with the self-migration problem. And we abstract some basic hypervisors which may make VMFUNC and the two-EPT architecture working better. Finally, we figure out a feasible self-migration flow based on iterative pre-copy, which could be a new solution to task transforming in JCCE.

REFERENCES

[1] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.

[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-Volume 2*, pages 273–286. USENIX Association, 2005.

[3] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 23. ACM, 2004.

[4] M. R. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[5] P. Liu, Z. Yang, X. Song, Y. Zhou, H. Chen, and B. Zang. Heterogeneous live migration of virtual machines.

[6] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[7] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, 2002.

[8] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002.

[9] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. Parallelizing live migration of virtual machines. In *ACM SIGPLAN Notices*, volume 48, pages 85–96. ACM, 2013.

[10] H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 1846–1855. IEEE, 2017.

[11] T. Wood, P. J. Shenoy, A. Venkataramani, M. S. Yousif, et al. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007.