FreeBSD - Live Migration feature for bhyve

Maria-Elena Mihăilescu
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: elenamihailescu22@gmail.com

Mihai Carabaș
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: mihai.carabas@cs.pub.ro

Abstract—**When talking about servers and clouds, live migration is one of the most powerful tools that can be used to manage resources that are abstracted by virtual machines due to its small downtime. bhyve, FreeBSD's hypervisor, does not have a live migration feature implemented yet, even though it is a very useful feature for a hypervisor.**

This paper presents two approaches for implementing a live migration feature for bhyve that use the FreeBSD's virtual memory subsystem. The first one uses a Copy-on-Write mechanism that cannot be implemented due to bhyve memory layout, and the second one uses a dirty page detection mechanism.

I. Introduction

Cluster and grid solutions have become more important each day, whether we talk about web servers or data centers. The cluster and grid framework usually offers resources for the clients by providing them access to certain virtual machines that abstract the hardware resources.

The virtual machine migration is a powerful tool that is used for load balancing or as a method to avoid data loss when one of the cluster's systems may become inaccessible in the near future (e.g., partial hardware failure, the need to upgrade the infrastructure). The migration process may be automated or may be done manually by the system administrator.

One of the migration's challenges is related to the guest's downtime: the more memory a guest has assigned, the more it may take for the migration process to finish. One of the fastest ways of migrating a virtual machine is by using the live migration procedure and migrate a guest from one host to another while it is still running.

bhyve [1] is a type 2 hypervisor implemented in the FreeBSD operating system and can be used on Intel and AMD CPUs systems that have support for virtualization. Linux, FreeBSD and Windows are some of the guest operating systems that can run in a virtual machine created with bhyve.

Unlike hypervisors such as VirtualBox, Xen, Hyper-V, VMWare ESX, and KVM that have a live migration feature, bhyve does not have one, even if it is necessary. In this paper, we present a Copy-on-Write mechanism that can be used to detect memory changes between live migration rounds, but that cannot be used by bhyve due to a dual memory layout implementation. Also, we propose a live migration feature for bhyve that uses a dirty-bit mechanism to detect the memory changes, and a save and restore mechanism feature [2] developed for bhyve to migrate the guest CPU and devices state.

This paper is split in eight sections. In Section II, we will present some of the main concepts that are used to develop the live migration feature for bhyve and for finding memory differences between migration rounds. In Section III, we will present a guest state save and restore mechanism and a cold and a warm migration feature for bhyve that is based on the save-restore procedure and that will be used in the live migration development process. In Section IV, we will show a Copy-on-Write approach for live migrating a guest memory that led to the current implementation and the reason it cannot be used in bhyve. In Section VI, we will suggest an algorithm that is based on the dirty-bit mechanism to detect memory changes. In the fifth section, the current status of the project along with its results is presented. In Section VII, we will present the future work that should be done to allow this project to evolve. In the last section, we will draw some conclusions for this paper.

II. State of the Art

A. FreeBSD's Virtual Memory Subsystem

The Virtual Memory Subsystem is one of the most important parts of an operating system since it manages the relationship between the physical memory and processes. This subsystem creates an abstraction layer between the software and the hardware so that a process can see a contiguous memory allocation space. Moreover, it ensures a level of security since a process cannot access another process's memory if the access was not granted using special mechanisms such as shared memory.

In the FreeBSD operating system, the virtual memory subsystem is object oriented and has four main components that are used to abstract the physical memory:

- `struct vm_page` – is the smallest virtual memory representation entity and represents a virtual page. It is mapped one-to-one with a physical memory page.
- `struct vm_object` – is a collection of `struct vm_page` entities that have the same characteristics. A `struct vm_object` entity represents an allocated area of contiguous memory.
- `struct vm_map_entry` – is an entry into an address map (represented by a `struct vm_map` object) that place an `struct vm_object` entity or another address map between a start address and an end address into a process's address space.

- struct vmspace – is an entity that represents the process virtual address space and points to a struct vm_map that contains a list of struct vm_map_entry entities. Moreover, it contains a link to the physical page table (struct pmap) for the represented process.

A struct vmspace entity is associated with each new process that is created on the system. This entity contains both a virtual memory mapping of the virtual pages and a reference to the physical page table. For each of the process's contiguous memory regions with the same characteristics (i.e., same permissions and flags) a struct vm_map_entry entity and a struct vm_object entity are created.

The Copy-on-Write (CoW) mechanism is used to optimize the system's memory usage and to rapidly create a new process when the fork() function is called. When fork() is called, the parent process' memory is marked as copy-on-write which means that the parent process and the child process share the same memory pages until one of them tries to modify one of the pages. Then, a page fault is triggered and the page is duplicated such that each of the two processes have an individual copy.

In FreeBSD, the Copy-on-Write mechanism is implemented using shadow objects [3]. A shadow object is a struct vm_object entity that is backed by another struct vm_object entity. It may happen that the backing objects can be another shadow object creating a list of shadow objects.

When a page from the shadow object is accessed, the page is first searched in the shadow object and if it is not found there, the page is searched in the backing object list. If the page resides in a backing object and is accessed for a write operation, then a copy of that page is added to the shadow object. If the page is accessed for a read operation, the object's layout is not modified. In the case of a fork call, an object is shadowed by two new objects: one for the parent and another one for the child.

B. Guest Memory Management

In a bhyve, the guest address space is split into three main components [4]:

- lowmem segment – this is the memory segment that is mapped between 0GB and lowmem limit size which is set at 3GB [5] at the time this paper was written. If the guest assigned memory size is smaller than the lowmem limit value, then the segment size is equal to the guest memory size. Otherwise, it is equal to the lowmem limit value.
- PCI hole – this is a non-mapped memory region between lowmem limit and 4GB (currently between 3GB and 4GB) that is used to access the devices through Memory-Mapped I/O (MMIO).
- highmem segment – this is the memory segment that is mapped starting from 4GB. This segment is equal in size to the difference between the guest assigned memory size and lowmem limit value, and it may not exist if the guest memory size is smaller than the lowmem limit value.

In bhyve, the guest memory is allocated during initial setup [5], where the user-space utility, bhyve, maps a contiguous area that is then divided between the lowmem segment and the highmem segment. Each of the two segments will have a new object assigned in the bhyve user-space process address space (a new struct vm_map_entry entry in the process's struct vmspace that will indicate the user-land bhyve process address range in which the segment was mapped). Then, in the kernel-space, the bhyve hypervisor, using the architecture dependent implementation for the vmspace_alloc function from the struct vmm_ops entity (a wrapper of architecture dependent functions), creates a new struct vmspace entity that will eventually point to the same memory objects that are allocated for the lowmem and highmem segments. However, the two struct vmspace entities (the one for the bhyve user-land utility, and the one that is allocated in the kernel) have different virtual and physical mappings, the latter corresponding to the guest address space layout that was previously discussed.

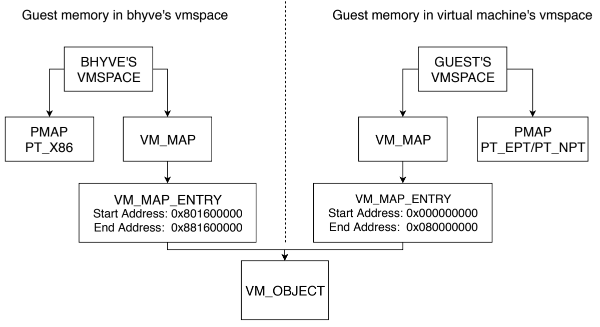
Guest memory in bhyve's vmspace
Guest memory in virtual machine's vmspace
BHYVE'S VMSPACE
GUEST'S VMSPACE
PMAP PT_X86
VM_MAP
VM_MAP
PMAP PT_EPT/PT_NPT
VM_MAP_ENTRY Start Address: 0x801600000 End Address: 0x881600000
VM_MAP_ENTRY Start Address: 0x000000000 End Address: 0x080000000
VM_OBJECT

Fig. 1. Dual Guest Memory View - lowmem segment

In Fig. 1 the dual guest memory view previously described for a bhyve virtual machine that has assigned 2GB of memory is represented. The left side represents the guest memory as it is mapped by the bhyve user-space tool and the right side is the guest memory representation as it is seen by the virtual machine itself. The same object that contains the guest memory pages is referred by two struct vm_map_entry entities. Each of the two struct vmspace entities have a link to a struct pmap. Since the nesting paging feature was introduced in bhyve [4], each physical mapping for the amd64 architecture has a mapping of type PT_x86 for normal mapping, or PT_EPT (for Intel Extended Page Table feature), and PT_NPT (for AMD Nested Page Table feature) for guest memory mappings.

The dual guest memory view represents a communication mechanism between the host and a virtual machine. User-space emulated devices (e.g., virtio, ahci, e1000) are running in different threads and receive and fulfill requests from a guest (e.g., reading or writing data on disk, receiving and sending network packages).

The left side view from Fig. 1 displays what happens when a request to read from disk is received by the host from the guest: the thread that emulates the disk interface from host user-space (e.g, virtio-block, ahci) is reading the data from disk and updates the guest memory by writing directly to it. The right side view from Fig. 1 shows how the enities are used when the guest accesses its memory and the information is directly read from the guest's `struct vmspace` entity.

C. Virtual Machine Migration

The virtual machine migration mechanism allows a user to move a guest from one host to another. From literature [6] [7] [8] [9] [10], the migration techniques can be divided into two main categories:

- Non-live Migration - the guest is powered off or suspended at migration time.
- Live Migration - the guest is running during the migration process.

The **Non-Live Migration** technique is divided in two main categories as well: cold migration and warm migration [7] [6]. The cold migration implies that the guest is powered off and all its data (disk and auxiliary files) are moved to another system. The warm migration procedure implies that a guest is suspended, its state copied onto the destination, and then the guest is resumed from the saved state.

Whereas in the cold migration case there are no restrictions regarding the guest's disk (because it is copied from one system to another), for the warm migration procedure, the disk image must be shared between the source and destination hosts. In terms of performance, the warm migration technique is faster than cold migration because of the fact that the disk is shared among systems.

The **Live Migration** technique has the best results in terms of migrated guest's downtime because the virtual machine is migrated while the guest is still running. The live migration procedure [8] [9] [10] has two phases: a phase in which the guest memory is migrated in rounds while the virtual machine is still running, and a phase in which the guest is stopped and the CPU's and devices' state are migrated to the destination.

Based on the method that is used in order to live migrate the guest's memory, there are two types of live migration [8] [9]:

- Pre-Copy Live Migration [8] – The memory migration is done in rounds. In the first round, all the guest pages are copied to the destination. For each of the following rounds, only the pages that were written between two rounds are copied to the destination. After a number of memory migration rounds or when a threshold number of dirty pages is reached, the virtual machine is stopped and the remaining dirty pages, together with the CPU's and devices' state is transferred to the destination host and the guest is started.
- Post-Copy Live Migration [9] – The memory is migrated using a page-fault approach. In the first phase, the source guest is stopped, the CPU's and devices' state is migrated to the destination and the guest is started on the destination host. When a memory access occurs, a page fault is generated on the destination, and then, the destination requires the page that caused the page fault from the source. To optimize the process, other pages will be delivered with the required page as well.

While the Post-Copy Live Migration has the advantage that the memory is transmitted a single time through the network, a fall-back mechanism is hard to be implemented, as opposed to the Pre-Copy Live Migration where if the migration process fails, the guest will continue running on the source host.

III. Related Work

A. Suspend and Resume a bhyve guest's state

As presented in Section II-C, during the migration process, a state save and restore procedure is needed: the guest's state is saved on the source host and restored on the destination host.

A project for bhyve state save and restore is also developed at University POLITEHNICA of Bucharest [2]. The project [2] introduces a suspend/resume feature for bhyve. When the suspend command is received, the bhyve process stops the virtual machine, saves the guest's state and its memory to disk files and destroys the guest. When resuming a virtual machine, the bhyve process restores the guest state based on the saved information.

```
# Suspend bhyve guest
root@host# bhyvectl --suspend=file.ckp \
                     --vm=vmname


# Restore a guest from checkpoint
root@host# bhyve <bhyve_options> \
                 -r file.ckp vmname
```

Listing 1. Suspend and Resume a bhyve guest

The Suspend request is sent to a bhyve guest by using the `bhyvectl` tool with the `--suspend` option and a file name for saving the data, as seen in Listing 1. Considering the code snippet in Listing 1, during the suspend process, three new files are created:

- **filename.ckp** - contains guest's memory.
- **filename.ckp.kern** - contains guest's devices and CPU state.
- **filename.ckp.meta** - contains metadata related to the saved devices and their offset in the **filename.ckp.kern** file.

Aside from the guest memory, there are other three main components whose state is saved during the suspend process:

- CPU state and related structures,
- Kernel devices such as VHPET (Virtual High Precision Timer), VRTC (Virtual Real Time Clock), VLAPIC (Virtual Local APIC), TSC (Time Stamp Counter).
- Userspace emulated devices (e.g., virtio-net, virtio-block, uart, ahci, lpc, frame buffer, xhci).

The Resume request, as seen in Listing 1, uses the `bhyve` tool with the `-r` parameter followed by the file name used when suspending the guest state. The restore process creates a fresh virtual machine based on the given disk image and updates its state and memory before the virtual CPUs are started.

B. Warm Migration in bhyve

Based on the save and restore feature for bhyve presented in Section III-A, a warm migration feature was added to bhyve [12]. Using the same API to retrieve a guest's state and memory as the save and restore project, the migration feature opens a connection between the source host and the destination host and sends the guest's state and memory through a socket.

As presented in Section III-A, the suspend/resume feature does not provide a disk checkpoint mechanism, and therefore, in order to warm migrate a bhyve guest, the same disk image must be shared between the two hosts using a storage sharing mechanism such as NFS (Network File System).

```
# Start source guest
root@src# bhyve <bhyve_options> vmsrc

# Start destination guest
# and wait for migration
root@dst# bhyve <bhyve_options> \
                  -R src_ip,port vmdst

# Migrate guest
root@src# bhyvectl --migrate=dst_ip,port \
                  --vm=vmsrc
```

Listing 2. Warm migrating a bhyve guest

In Listing 2 is presented an example of warm migration usage. In order to warm migrate a virtual machine, a fresh guest is started on the destination host using the `bhyve` tool with the `-R` parameter followed by the host's IP and the listening port. The destination host waits to receive source guest's state. To send the guest's state, on the source host, the `bhvyectl` tool is used with the `--migrate` parameter followed by the destination host and a port. After the communication between the two hosts is established, the source host is stopped, its state and memory is sent to the destination host and if the migration is successfully completed, the source guest is destroyed (otherwise, if an error occurs, the guest will continue running on the source host). The destination host receives the guest's state and memory and based on the resume state API [2], restores the guest and starts the virtual machine's CPUs.

IV. Live Memory Migration using a Copy-on-Write approach

The memory migration is the core of a live migration feature, and in the same time, is the most difficult part to implement. As stated in Section II-C, the memory can be migrated before [8] or after [9] starting the guest on the destination host.

Considering the pre-copy live migration feature [8], the same memory page can be migrated more than once, whereas in the post-copy live migration approach [9], each page is migrated only once. However, when the migration procedure fails (e.g., network connection become unavailable), the post-copy live migration approach needs to implement a fall-back mechanism [9]. In the pre-copy live migration, when an error occurs, the fall-back mechanism is ensured by default because the guest continues running on the source host. Considering this, we choose to implement a pre-copy live migration feature for bhyve.

As presented in Section II-C, in a pre-copy live migration approach, the memory is migrated in rounds and, each round, the algorithm has to determine the pages that were dirtied since the last round started. In other words, the procedure needs to determine the changes that occured in the same memory area between two moments in time.

The Copy-on-Write (CoW) mechanism can be used to determine the memory differences that were made in a time interval. As presented in Listing 3, in FreeBSD, to determine the memory differences, the memory object can be marked as Copy-On-Write. Then, the object and its shadow object can be compared to determine what pages were modified. As presented in Section II-A, when a write operation occurs in a page that was not been dirtied since the initial object was marked as CoW, a copy of that page is added into the shadow object. Considering this, the shadow object contains only the pages that were dirtied since the initial object was marked as CoW.

```
VM_OBJ = get_current_mem_obj();
SHADOW_OBJ = set_obj_cow(VM_OBJ);
wait_for_round_to_finish();
pg = get_pg(SHADOW_OBJ);
```

Listing 3. Determining memory differences using the Copy-on-Write

The Copy-on-Write approach presented in Listing 3 can be used to determine the memory that can be migrated in each round. The algorithm needs guest memory objects to be marked as CoW before starting a new memory migration round. In the next round, the pages that should be migrated are represented by the shadow object pages.

However, due to the dual memory view presented in Section II-B, the Copy-on-Write approach previously presented cannot be used to migrate de virtual machine's memory. In order to mark a virtual memory object (the `struct vm_object` entity) as Copy-on-Write, some flags (`MAP_ENTRY_COW` and `MAP_ENTRY_NEEDS_COPY`) have to be set in the `struct vm_map_entry` entity. However, as seen in Fig. 1, the guest and the host have different `struct vm_map_entry` entities to refer the same object.

If one of the `struct vm_map_entry` entities (from guest side or from host side) is marked as Copy-on-Write, the other `struct vm_map_entry` entity points to other `struct vm_object` so the two sides will no longer have the same view of the memory. If one of them writes in the guest's memory (such as virtio devices) the page will be copied into the shadow object and the other side will not see the

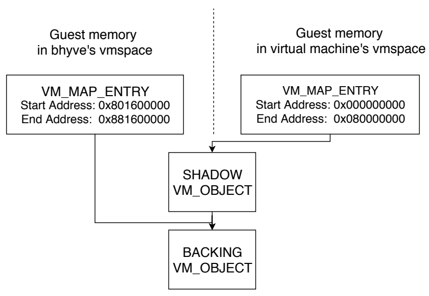

Fig. 2. Dual Guest Memory View - lowmem segment - bhyve's VM_MAP_ENTRY is marked as COW

changes. This leads to communication errors between the host and the guest that will eventually crash the virtual machine.

In Fig. 2, it is shown the virtual memory layout after the guest's side `struct vm_map_entry` that contains the memory object was marked as copy-on-write.

This approach is the one of the easiest methods of determining the memory that should be migrated each round because of the FreeBSD virtual memory subsystem implementation. Using the CoW mechanism, the only pages that should be migrated are the ones that reside in the shadow object. Even if this method cannot be used in bhyve due to its dual memory view, it gave us a clearer picture of the virtual memory subsystem functionality and we developed another memory modification detection mechanism to implement the live migration feature for bhyve.

V. Live Memory Migration using a dirty-bit approach

As stated in Section IV, the memory migration is one of the most challenging parts in a live migration feature. In Section IV we presented a memory modification detection mechanism that uses with virtual memory objects.

Even though the virtual memory objects that represent the guest memory are the highest entities in the virtual machine memory hierarchy (as presented in Fig. 1) that are common for both the guest (i.e., as seen by the virtual machine's `vmspace` entity) and the host (i.e., as seen by the bhyve user space utility tool's `vmspace` entity), there are other constraints that do not allow the use of virtual memory objects to determine memory differences. For instance, one of the constraints is the fact that a `vm_object` entity is marked as CoW by setting some flags in a memory layout enitity that is not shared between host and guest because of their separate memory views. Thus, we will present a solution that uses another entity shared between host and guest: the `struct vm_page` entities.

Each `struct vm_page` contains a dirty flag field that indicates if the page has been modified and if so, the changes should be written on the disk. This flag is updated from time to time based on the modified bit of the physical page by inspecting the A/D - access/dirty - bits. Even though this flag could be used for determining the pages that should be migrated, we cannot interfere with this flag since the virtual memory system relies on it to perform some virtual memory subsystem actions such as the memory laundering process [11], and modifying its behavior would imply some undesired operating system behavior.

Instead of using the dirty flag, the proposed approach uses a custom dirty bit used only by the bhyve hypervisor. The custom dirty bit, named **virtual-machine-dirty bit**, is set each time the dirty flag is set, but unset only by the bhyve hypervisor.

```
VM_OBJ = get_current_mem_obj();
clear_vmm_dirty_bit_for_pg_in(VM_OBJ);
wait_for_round_to_finish();
pg = get_pg_with_vmm_dirty_bit(VM_OBJ);
```

Listing 4. Finding a guest's dirty pages using dirty bit

The pseudo-code snippet in Listing 4 presents a mechanism that can be used to determine memory pages changed between two moments in time. Firstly, each page from the memory object should have the virtual-machine-dirty bit clean (to eliminate false-positive cases). The memory differences can be determined by inspecting the virtual memory object pages that have the virtual-machine-dirty bit set.

VI. Algorithm

In order to implement a live migration feature for bhyve, we use the approach presented in Section V for migrating the memory and the state save and restore mechanism implemented for bhyve [2] that was presented in the Section III-A. The connection mechanism between the source host and the destination host is the socket solution also used by the warm migration feature for bhyve [12] shown in Section III-B. The live migration algorithm is similar to the warm migration algorithm, the major differences are related only to the memory migration.

```
connect(src, dst);
check_compatibility(src, dst);
live_migration_send_memory_to(dst);
snapshot_and_send_state();
destroy_vm()
```

Listing 5. Live Migration Algorithm - Source host method

```
connect(src, dst);
check_compatibility(src, dst);
live_migration_recv_memory_from(src);
recv_and_resume_state();
spinof_vcpus();
```

Listing 6. Live Migration Algorithm - Destination host method

The pseudo-code snippets from Listing 5 and from Listing 6 present the functions that run on the source host and on the destination host in order to migrate a guest. After the connection between the source and the destination hosts is done, there is a check to determine whether the two are compatible for migration (e.g., same CPU vendor and model, same guest memory size, same virtual memory page size). After that, the memory is migrated in rounds. In the last step, the guest's remaining dirty memory is sent to the destination, and the guest's CPU's and devices's state is being snapshot using the state save and restore feature. The virtual machine's state is restored at the destination. The guest is stopped on the source host before the last step.

```
live_migration_send:
  for i=1:N
    if i == 0
      // First Round
      mark_all_memory_dirty();
     endif
     if i == N
       // Last Round
       stop(vm);
     endif
     pages = get_dirty_pages();
     send(pages);
  end for

send(pages):
  for each page : pages
        get_from_memory(page);
        clear_dirty_bit(page);
        send_to_dest(page);
  end for
```

Listing 7. Live Memory Migration Algorithm - Send Memory

```
live_migration_recv:
  while recv_from_src(page)
    update(page);
  end while
```

Listing 8. Live Memory Migration Algorithm - Receive Memory

In Listing 7 the algorithm used for sending the guest memory in rounds to the destination is presented. In the first migration round, all guest pages should be migrated so each page is artificially set as dirty by setting the virtual-machine-dirty bit. In the next rounds, the memory differences are determined by iterating through all of the guest pages and sending them one by one to the destination. When copying a page from the guest memory, we clear the virtual-machine-dirty bit. By clearing the virtual-machine-dirty bit for each migrated page, we prepare the guest for the next round. In the last round, the virtual machine should be stopped and the remaining memory migrated. Listing 8 shows the algorithm used by the destination host. It receives pages one by one and updates the guest memory that will be started after the migration process completes.

VII. Current Status in bhyve and Future Work

The algorithm presented in Section VI is implemented in bhyve [13] and the project is still under development. To start a migration procedure, the process is similar to the warm migration algorithm. As seen in Listing 9, in terms of usage, the only difference between warm and live migration is related to the `bhyvectl` command that starts migrating the guest: instead of `migrate=dst_ip,port` the `migrate-live` option is used.

```
# Start source guest
root@src# bhyve <bhyve_options> vmsrc

# Start destination guest
# and wait for migration
root@dst# bhyve <bhyve_options> \
            -R src_ip ,port vmdst

# Migrate guest
root@src# bhyvectl \
            --migrate-live=dst_ip ,port \
            --vm=vmsrc
```

Listing 9. Live migrating a bhyve guest

In order to have the same connection framework for both warm and live migration, we modified the algorithm so among the initial messages related to specification checks, the type of migration (warm or live) is sent to the destination. Based the migration type, the destination determines the functions to be called for memory migration.

The framework for live migration is implemented, the live migration feature is not yet stable and there are currently some limitations that should be considered next:

- the guest memory should be wired - since we added a mechanism for retrieving pages from memory and the first migration round is supposed to migrate all the pages, all memory should be allocated, and the pages should not be swapped out. Thus, we choose to live migrate only wired guests.
- the guest memory size should be less than the lowmem segment - we implemented the framework to work for guests that have assigned a virtual memory object only for lowmem segment.
- the migrated guest crashes in some of the test scenarios. The debugging process is still ongoing and this behavior may be caused by the emulated devices that run in user-space threads, that will continue running even when the guest's virtual CPUs are locked, affecting or even corrupting the guest's state and disk.

VIII. Conclusion

In this paper, we presented two mechanism for determining the memory differences between two memory rounds.

The first approach is based on the FreeBSD Copy-on-Write mechanism. To identify the pages that should be migrated using shadow virtual memory objects. Even if the algorithm can determine the modified pages between two memory migration rounds, it cannot be implemented in bhyve due to the dual memory view of the guest memory.

The second mechanism for detecting the modified pages is based on a dirty bit approach. We use a bit, named virtual-machine-dirty bit, that is managed only by the hypervisor.

The dirty-bit approach is currently used for live migrating the guest memory. Even if the framework is implemented, the live migration feature is not yet stable and there are some challenges and improvements that should be considered in the future.

Acknowledgments

The authors would like to thank to John Baldwin, Mark Johnston and Alan Cox for their help and advice in regards to bhyve and FreeBSD's virtual memory subsystem implementation and functionality. Also, the authors would like to thank to Matthew Grooms for his technical and financial support in form of scholarship for Elena Mihailescu.

References

[1] FreeBSD Handbook, Chapter 21. Virtualization, Section 21.7 FreeBSD as Host with bhyve [Online]. Available: https://www.freebsd.org/doc/handbook/virtualization-host-bhyve.html, [Accessed Dec, 21st, 2018].

[2] University POLITEHNICA of Bucharest, Save & Restore Project for bhyve (amd64) [Online]. Available: https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyve_snapshot, [Accessed Jan, 27th, 2019].

[3] Matthew Dillon, "Design elements of the FreeBSD VM system" [Online]. Available: https://www.freebsd.org/doc/en/articles/vm-design/, [Accessed Dec, 21st, 2018].

[4] N. Natu, P. Grehan, "Nested Paging in bhyve", in *AsiaBSDCon, Tokio, Japan, March 2014*

[5] The FreeBSD Project, FreeBSD Source Code [Online]. Available: https://github.com/freebsd/freebsd, [Accessed Dec, 21st, 2018].

[6] VMware, "VMware virtual machine migration types vSphere 6.0" [Online]. Available: https://communities.vmware.com/docs/DOC-31922 [Accessed Dec, 21st, 2018].

[7] Network Startup Resource Center, "Virtual Machine Migration" in *Cloud / Virtualization workshop*, Thimphu, Bhutan, 17-21 January 2014 [Online]. Available: https://nsrc.org/workshops/2014/sanog23-virtualization/raw-attachment/wiki/Agenda/migration-storage.pdf [Accessed Dec, 21st, 2018].

[8] C. Clark, and K. Fraser and S. Hand, and J.G. Hansen, and E. Jul, and C. Limpach, and 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*, 2005, pp. 273-286

[9] M.R. Hines and U. Deshpande and K. Gopalan, "Post-copy live migration of virtual machines", *ACM SIGOPS operating systems review*, vol. 43, pp. 14-26, 2009.

[10] A. Kivity and Y. Kamay and D. Laor and U. Lublin and A. Liguori, "kvm: the Linux virtual machine monitor", in *Proceedings of the Linux symposium*, vol. 1, pp. 225-230, 2007

[11] The FreeBSD Documentation Project, "FreeBSD Architecture Handbook", Chapter 7. Virtual Memory System [Online]. Available: https://www.freebsd.org/doc/en/books/arch-handbook/vm.html [Accessed Jan, 27th, 2019]

[12] University POLITEHNICA of Bucharest, Warm Migration for bhyve (amd64) [Online]. Available: https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyve_warm_migration, [Accessed Jan, 27th, 2019].

[13] University POLITEHNICA of Bucharest, Live Migration for bhyve (amd64) [Online]. Available: https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyve_migration_dev, [Accessed Jan, 27th, 2019]