VPC: Scalable, Low Downtime Checkpointing for Virtual Clusters

Peng Lu
ECE Department
Virginia Tech
Blacksburg, VA, USA
Email: lvpeng@vt.edu

Binoy Ravindran
ECE Department
Virginia Tech
Blacksburg, VA, USA
Email: binoy@vt.edu

Changsoo Kim
ETRI
Daejeon, South Korea
Email: cskim7@etri.re.kr

***Abstract*—A virtual cluster (VC) consists of multiple virtual machines (VMs) running on different physical hosts, interconnected by a virtual network. A fault-tolerant protocol and mechanism are essential to the VC's availability and usability. We present Virtual Predict Checkpointing (or VPC), a lightweight, globally consistent checkpointing mechanism, which checkpoints the VC for immediate restoration after VM failures. By predicting the checkpoint-caused page faults during each checkpointing interval, VPC further reduces the solo VM downtime than traditional incremental checkpointing approaches. Besides, VPC uses a globally consistent checkpointing algorithm, which preserves the global consistency of the VMs' execution and communication states, and only saves the updated memory pages during each checkpointing interval to reduce the entire VC downtime. Our implementation reveals that, compared with past VC checkpointing/migration solutions including VNsnap, VPC reduces the solo VM downtime by as much as 45%, under the NPB benchmark, and reduces the entire VC downtime by as much as 50%, under the NPB distributed program. Additionally, VPC incurs a memory overhead of no more than 9%. In all cases, VPC's performance overhead is less than 16%.**

I. Introduction

High availability (HA) is increasingly becoming important in today's clusters and data centers. HA refers to an entire cluster or data center and its associated implementation being continuously operational for a long period of time. Whole-server replication is a conventional method to increase HA — once a primary server fails, the running applications are migrated and resumed on the backup server. However, there are several limitations that make this method unattractive for deployment: it needs specialized hardware and software, which are usually expensive. Additionally, such a system may also require complex customized configurations, which are difficult to manage and maintain.

One possible way to overcome these limitations is virtualization: all applications now run on a virtual machine (VM), and one or several VMs run on a physical server. Thus, whole-server replication can be easily and efficiently implemented by replicating the running VMs — a copy of the VM on a physical server is continuously check-pointed, transferred, and saved on a backup server. As VMs are totally hardware-independent, the cost is significantly lower compared to the hardware expenses in traditional HA solutions. Moreover, virtualization technology also provides numerous benefits to clusters and data centers such as greater security within user-customized environments, improved load balancing, and server consolidation. Perhaps, the biggest advantage is the flexibility to map physical resources to a virtual server (or a VM) and its applications so as to handle workloads dynamically.

A virtual cluster (VC) generalizes the VM concept for distributed applications and systems. A VC is a set of multiple VMs deployed on different physical servers, but managed as a single entity. A VC can be created to provide computation, software, data access, or storage services to individual users, who do not require knowledge of the physical location or configuration of the system. End-users typically submit their applications, often distributed, to a VC, and the environment transparently hosts those applications on the underlying set of VMs. VCs are gaining increasing traction in the "Platform as a Service" (PaaS; e.g., Google App Engine [2]) and "Infrastructure as a Service" (IaaS; e.g., Amazon's AWS/EC2 [1]) paradigms in the cloud computing domain.

To provide benefits such as dynamic resource allocation and fault tolerance, a useful feature of virtualization is the possibility of saving, restoring, and migrating an entire VM through transparent checkpointing. Most state-of-the-art/practice virtualization systems (e.g., Xen [7], VMware [26], KVM [3]) provide mechanisms to checkpoint/restart VMs in an application-transparent manner. Unlike application-level checkpointing [15, 22], VM-level checkpointing usually involves recording the virtual CPU's state, the current state of all emulated hardware devices, and the contents of the running VM's memory. VM-level checkpointing is typically a time consuming process due to potentially large VM memory sizes (sometimes, it is impractical as the memory size may be up to several gigabytes). Therefore, for solo VM checkpointing, often a lightweight methodology is adopted [27, 17], which doesn't generate a large checkpoint file.

The checkpointing size also affects the scalability of providing fault tolerance for an entire VC (through checkpointing). In addition, in a VC, since multiple VMs are distributed as computing nodes across different servers, the failure of one VM can affect the states of other related VMs, and may sometimes cause them to also fail. For example, assume that we have two VMs, VM_a and VM_b, running in a VC. Say, VM_b sends some messages to VM_a and then fails. These

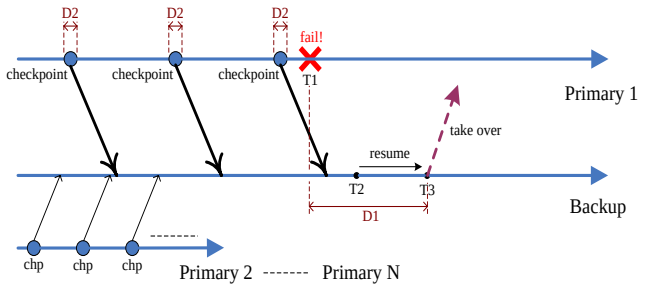

Fig. 1. Primary-Backup model and the downtime problem. T_1: primary VM fails; T_2: the failure is observed by the VC; T_3: VM resumes on backup server; D_1 ($T_3 - T_1$): VC downtime; D_2: VM downtime.

messages may be correctly received by VM_a and may change the state of VM_a. Thus, when VM_b is rolled-back to its latest, correct check-pointed state, VM_a must also be rolled-back to a check-pointed state before the messages were received from VM_b. In other words, all the VMs (i.e., the entire VC) must be check-pointed at globally consistent states.

Figure 1 illustrates a basic fault-tolerant protocol design for VC, using the classical "primary-backup" model. All the VMs in the VC are represented by "primary" in the figure. The state of each primary VM is check-pointed at a globally consistent state, and all the checkpoints are saved on the "backup", which can be a VM or a physical machine. When anyone VM fails, which causes a VC failure, the backup VM/server will take over and roll-back each VM to its previous check-pointed state, and thereby resume the entire VC from a globally consistent checkpoint.

Figure 1 also illustrates the primary metric of HA systems: *downtime*. Two downtimes are of interest: First is VC downtime, which is the time from when the failure was detected in the VC to when the VC resumes from the last check-pointed state on the backup VM/server and starts to handle client requests. Second is VM downtime, which is the time from when the VM pauses to save for the checkpoint to when the VM resumes. Obviously, saving a smaller checkpoint will cost less time than saving a larger checkpoint. Thus, a lightweight checkpoint methodology directly reduces the VM downtime.

We present Virtual Predict Checkpointing (or VPC), a lightweight, globally consistent checkpointing mechanism for VCs. VPC checkpoints only the updated memory pages instead of the whole image during each checkpointing interval. We show that a small checkpoint size leads to minimal downtime with acceptable memory overhead. By predicting the checkpoint-caused page faults during each checkpointing interval, VPC further reduces the solo VM downtime than traditional incremental checkpointing approaches [24, 17]. We develop a variant of Mattern's distributed snapshot algorithm [16] for capturing global snapshots of the VC that preserve the consistency of the VMs' execution and related communication states.

We construct a Xen-based implementation of VPC and conduct experimental studies. Our studies reveal that, compared with past VC checkpointing/migration solutions including VNsnap [13], VPC reduces the solo VM downtime by as much as 45%, under the NPB benchmark [4], and reduces the entire VC downtime by as much as 50%, when running the distributed programs in NPB benchmark. VPC only incurs a memory overhead of no more than 9%. In all cases, VPC's performance overhead is less than 16%.

Our VPC implementation is open-sourced [25].

The rest of the paper is organized as follows. Section II discusses past and related work. Section III presents the design and implementation of VPC. Section IV presents our globally consistent checkpointing algorithm. Section V reports our experimental environment, benchmarks, and evaluation results. We conclude in Section VI.

II. Related Work

Checkpointing is a commonly used approach for achieving fault-tolerance. Checkpoints can be taken at different levels of abstraction. Application-level checkpointing is one of the most widely used methods. For example, Lyubashevskiy *et al.* [15] develop a file operation wrapper layer with which a copy-on-write file replica is generated while keeping the old data unmodified. Pei *et al.* [22] wrap standard file I/O operations to buffer file changes between checkpoints. However, these checkpointing tools require modifying the application code, and thus, they are not transparent to applications.

OS-level checkpointing solutions have also been widely studied. For example, Libckpt [23] is an open-source, portable checkpointing tool for UNIX. It mainly focuses on performance optimization, and only supports single-threaded processes. Osman *et al.* [20] present Zap, which decouples protected processes from dependencies to the host operating system. A thin virtualization layer is inserted above the OS to support checkpointing without any application modification. However, these solutions are highly context-specific and often require access to the source code of the OS kernel, which increases OS-dependence. In contrast, VPC doesn't require any modification to the guest OS kernel or the application running on the VM.

Our work focuses on checkpointing at the VM-level. VM-level checkpointing can be broadly classified into two categories: stop-and-save checkpointing and checkpointing through live migration. In the first category, a VM is completely stopped, its state is saved in persistent storage, and then the VM is resumed [18]. This technique is easy to implement, but incurs a large system downtime during checkpointing. Live VM migration is designed to avoid such large downtimes — e.g., VMWare's VMotion [19], Xen Live Migration [8]. During migration, physical memory pages are sent from the source (primary) host to the destination (backup) host, while the VM continues to run on the source host. Pages modified during replication must be re-sent to ensure consistency. After a bounded iterative transferring phase, a very short stop-and-copy phase is executed, during which the VM is halted, the remaining memory pages are sent, and the destination hypervisor is signaled to resume the execution of the VM. Remus [9]

uses high frequency checkpointing to handle hardware fail-stop failures on a single host with whole-system migration. It does this by maintaining a completely up-to-date copy of a running VM on the backup server, which automatically activates if the primary server fails. However, Remus incurs large overhead (overhead reported in [9] is approximately 50% for a checkpointing interval of $50ms$).

Multi-VM checkpointing mechanisms — our target problem space — have also been studied. Based on a nonblocking distributed snapshot algorithm, VNsnap [13] takes global snapshots of virtual networked environments and does not require reliable FIFO data transmission. VNsnap runs outside a virtual networked environment, and thus does not require any modifications to software running inside the VMs. The work presents two checkpointing daemons, called VNsnap-disk and VNsnap-memory. These solutions generate a large checkpoint size, which is at least the guest memory size. Also, VNsnap-memory stores the checkpoints in memory, which duplicates the memory, resulting in roughly 100% memory overhead. Additionally, their distributed snapshot algorithm (which is a variant of [16]) uses the "receive-but-drop" strategy, which would cause temporary backoff of active TCP connections inside the virtual network after checkpointing. The TCP backoff time is non-negligible and seriously affects the downtime. In Section V-C, we experimentally compare VPC with VNsnap, and show VC downtime improvements by as much as 60%.

III. Design and Implementation of VPC

A. Lightweight Checkpointing Implementation

Since a VC may have hundreds of VMs, to implement a scalable lightweight checkpointing mechanism for the VC, we need to checkpoint/resume each VM with minimum possible overhead. To completely record the state of an individual VM, checkpointing typically involves recording the virtual CPU's state, the current state of all emulated hardware devices, and the contents of the guest VM's memory. Compared with other preserved states, the amount of the guest VM memory which needs to be checkpointed dominates the size of the checkpoint. However, with the rapid growth of memory in VMs (several gigabytes are not uncommon), the size of the checkpoint easily becomes a bottleneck. One solution to alleviate this problem is incremental checkpointing [24, 17], which minimizes the checkpointing overhead by only synchronizing the dirty pages during the latest checkpoint. A page fault-based mechanism is typically used to determine the dirty pages [11]. We first use incremental checkpointing in VPC.

We deploy a VPC agent that encapsulates our checkpointing mechanism on every server. For each VM on a server, in addition to the memory space assigned to its guest OS, we assign a small amount of additional memory for the agent to use. During system initialization, we save the complete image of each VM's memory on the disk. To differentiate this state from "checkpoint," we call this state, "non-volatile copy". After the VMs start execution, the VPC agents begin saving the correct state for the VMs. For each VM, at the beginning of a checkpointing interval, all memory pages are set as read-only. Thus, if there is any write to a page, it will trigger a page fault. Since we leverage the shadow-paging feature of Xen, we are able to control whether a page is read-only and to trace whether a page is dirty. When there is a write to a read-only page, a page fault is triggered and reported to the Xen hypervisor, and we save the current state of this page.

When a page fault occurs, this memory page is set as write-able, but VPC doesn't save the modified page immediately, because there may be another new write to the same page in the same interval. Instead, VPC adds the address of the faulting page to the list of changed pages and removes the write protection from the page so that the application can proceed with the write. At the end of each checkpointing interval, the list of changed pages contains all the pages that were modified in the current checkpointing interval. VPC copies the final state of all modified pages to the agent's memory, and resets all pages to read-only again. A VM can then be paused momentarily to save the contents of the changed pages (which also contributes to the VM's downtime). In addition, we apply a high frequency checkpointing mechanism (Section III-B), which means that each checkpointing interval is set to be very small, and therefore, the number of updated pages in an interval is small as well. Thus, it is unnecessary to assign large memory to each VPC agent. (We discuss VPC's memory overhead in Section V-D.)

Note that, this approach incurs a page fault whenever a read-only page is modified. When running memory-intensive workloads on the guest VM, handling so many page faults affects scalability. On the other hand, according to the principle of locality on memory accesses, recently updated pages tend to be updated again (i.e., spatial locality) in the near future (i.e., temporal locality). In VPC, we set the checkpointing interval to be small (tens to hundreds of milliseconds). So similarly, the dirty pages also follow this principle. Therefore, we use the updated pages in the previous checkpointing interval to predict the pages which will be updated in the upcoming checkpointing interval – i.e., by pre-marking the predicted pages as writable at the beginning of the next checkpointing interval. By this improved incremental checkpointing methodology, we reduce the number of page faults.

The page table entry (PTE) mechanism is supported by most current generation processors. For predicting the dirty pages, we leverage one control bit in the PTE: accessed (A) bit. The accessed bit is set to enable or disable write access for a page. Similar to our incremental checkpointing approach, for each VM, at the beginning of a checkpointing interval, all memory pages are set as read-only (accessed bit is cleared as "0"). Thus, if there is any write to a page, it will trigger a page fault, and the accessed bit is set to "1." However, unlike our incremental checkpointing approach, after the dirty pages in a checkpointing interval are saved in the checkpoint, we do not clear the accessed bits of these newly updated pages at the end of a checkpointing interval. Instead, the accessed bits of these pages are kept as writeable to allow write during the next interval. At the end of the next interval,

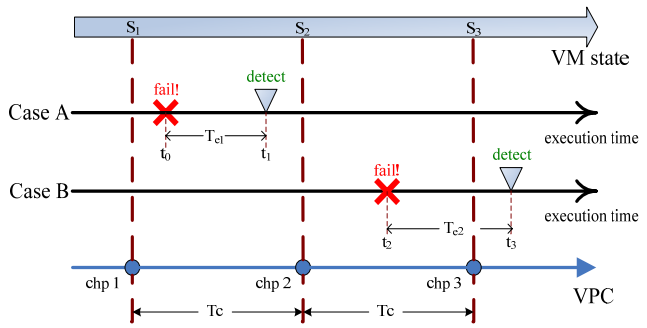

Fig. 2. Two execution cases under VPC.

we track whether these pages were actually updated or not. If they were not updated, their accessed bits are cleared, which means that the corresponding pages are set as read-only again. Our experimental evaluation shows that, this approach further reduces the (solo) VM downtime (Section V-B).

B. High Frequency Checkpointing Mechanism

VPC uses a high frequency checkpointing mechanism. Our motivation for this methodology is that, several previous fault injection experiments [14, 21] have shown that most system crashes occur due to transient failures. For example, in the Linux kernel [10], after an error happens, around 95% of crashes occur within 100 million CPU cycles, which means that, for a 2 GHz processor, the error latency is very small (within $50ms$).

Suppose the error latency is T_e and the checkpointing interval is T_c. Thus, as long as $T_e \leq T_c$, the probability of an undetected error affecting the checkpoint is small. For example, if more than 95% of the error latency is less than T_e, the possibility of a system failure caused by an undetected error is less than 5%. Therefore, as long as T_c (application-defined) is no less than T_e (in this example, it is $50ms$), the checkpoint is rarely affected by an unnoticed error. Thus, this solution is nearly error-free by itself. On the other hand, if the error latency is small, so is the checkpointing interval that we choose. A smaller checkpointing interval means a high frequency methodology.

In VPC, for each VM, the state of its non-volatile copy is always one checkpointing interval behind the current VM's state except the initial state. This means that, when a new checkpoint is generated, it is not copied to the non-volatile copy immediately. Instead, the last checkpoint will be copied to the non-volatile copy. The reason is that, there is a latency between when an error occurs and when the failure caused by that error is detected.

For example, in Figure 2, an error happens at time t_0 and causes the system to fail at time t_1. Since most error latencies are small, in most cases, $t_1 - t_0 < T_e$. In case A, the latest checkpoint is $chp1$, and the system needs to roll-back to the state S_1 by resuming from the checkpoint $chp1$. However, in case B, an error happens at time t_2, and then a new checkpoint $chp3$ is saved. After the system moves to the state S_3, this error causes a failure at time t_3. Here, we assume that $t_3 - t_2 < T_e$. But, if we choose $chp3$ as the latest correct checkpoint and roll the system back to the state S_3, after resuming, the system will fail again. We can see that, in this case, the latest checkpoint should be $chp2$, and when the system crashes, we should roll it back to the state S_2, by resuming from the checkpoint $chp2$.

VPC is a lightweight checkpointing mechanism, because, for each protected VM, the VPC agent stores only a small fraction of, rather than the entire VM image. For a guest OS occupying hundreds of megabytes of memory, the VPC checkpoint is no more than 20MB. In contrast, past efforts such as VNsnap [13] duplicates the guest VM memory and uses the entire additional memory as the checkpoint size. In VPC, with small amount of memory, we can store multiple checkpoints for different VMs running on the same server. Meanwhile, as discussed in Section I, the size of the checkpoint directly influences the VM downtime. This lightweight checkpointing methodology reduces VPC's downtime during the checkpointing interval. (We evaluate VPC's downtime in Section V-B.)

IV. Distributed Checkpoint Algorithm in VPC

A. Communication Consistency

To compose a globally consistent state of all the VMs in the VC, the checkpoint of each VM must be coordinated. Besides checkpointing each VM's correct state, it is also essential to guarantee the consistency of all communication states within the virtual network. Recording the global state in a distributed system is non-trivial because there is no global memory or clock in a traditional distributed computing environment. So the coordination work must be done in the presence of non-synchronized clocks for a scalable design.

We illustrate message communication with an example in Figure 3. The messages exchanged among the VMs are marked by arrows going from the sender to the receiver. The execution line of the VMs is separated by their corresponding checkpoints. The upper part of each checkpoint corresponds to the state before the checkpoint and the lower part of each checkpoint corresponds to the state after the checkpoint. A global checkpoint (consistent or not) is marked as the "cut" line, which separates each VM's timeline into two parts.

We can label the messages exchanged in the virtual network into three categories:

1) The state of the message's source and the destination are on the same side of the cut line. For example, in Figure 3, both the source state and the destination state of message m_1 are above the cut line. Similarly, both the source state and the destination state of message m_2 are under the cut line.

2) The message's source state is above the cut line while the destination state is under the cut line, like message m_3.

3) The message's source state is under the cut line while the destination state is above the cut line, like message m_4.

For these three types of messages, we can see that a globally consistent cut must ensure the delivery of type (1) and type (2) messages, but must avoid type (3) messages. For example, consider the message m_4 in Figure 3. In VM3's checkpoint

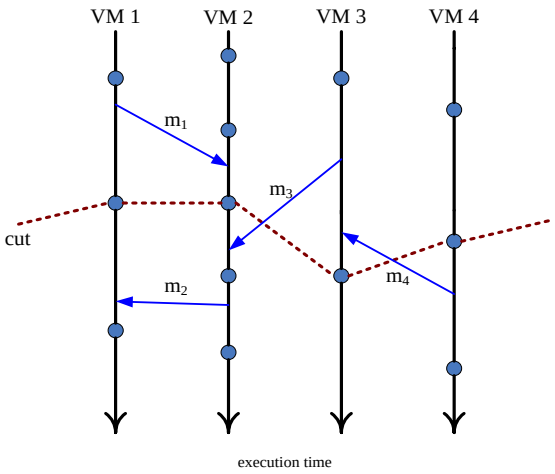

Fig. 3. The definition of global checkpoint.

saved on the cut line, m_4 is already recorded as being received. However, in VM4's checkpoint saved on the same cut line, it has no record that m_4 has been sent out. Therefore, the state saved on VM4's global cut is inconsistent, because in VM4's view, VM3 receives a message m_4, which is sent by no one.

B. Globally Consistent Checkpointing Design

In VPC design, we develop a variant of the distributed checkpointing algorithm in [16] as the basis of our lightweight checkpointing mechanism. For completeness, we summarize the original algorithm here: This algorithm relies on vector clocks, and uses a single initiator process. At the beginning, a global snapshot is planned to be recorded at a future vector time s. The initiator broadcasts this time s and waits for acknowledgements from all the recipients. When a process receives the broadcast, it remembers the value s and acknowledges the initiator. After receiving all acknowledgements, the initiator increases its vector clock to s and broadcasts a dummy message. On the receiver's side, it takes a local snapshot, sends it to the initiator, and increases its clock to a value larger than s. Finally, the algorithm uses a termination detection scheme to decide whether to terminate the algorithm.

As illustrated before, type (3) messages are unwanted, because they are not recorded in any source VM's checkpoints, but they are already recorded in some checkpoints of a destination VM. In VPC, there is always a correct state for a VM, recorded as the non-volatile copy in the disk. As explained in Section III-B, the state of the non-volatile copy is one checkpointing interval behind the current VM's state, because we copy the last checkpoint to the non-volatile copy only when we get a new checkpoint. Therefore, before a checkpoint is committed by saving to non-volatile copy, we buffer all the outgoing messages in the VM during the corresponding checkpointing interval. Thus, type (3) messages are never generated, because the buffered messages are unblocked only after saving their information by copying the checkpoint to the non-volatile copy. Our algorithm works under the assumption that the buffering messages will not be lost or duplicated. (This assumption can be overcome by leveraging classical ideas, e.g., as in TCP.)

In VPC, there are multiple VMs running on different servers connected within the network. One of the servers is chosen to deploy the VPC Initiator, while the protected VMs run on the primary servers. The Initiator can be running on a VM which is dedicated to the checkpointing service. It doesn't need to be deployed on the privileged guest system like the Domain 0 in Xen. When VPC starts to record the globally consistent checkpoint, the Initiator broadcasts the checkpointing request and waits for acknowledgements from all the recipients. Upon receiving a checkpointing request, each VM checks the latest recorded non-volatile copy (not the in-memory checkpoint), marks this non-volatile copy as part of the global checkpoint, and sends a "success" acknowledgement back to the Initiator. The algorithm terminates when the Initiator receives the acknowledgements from all the VMs. For example, if the Initiator sends a request (marked as rn) to checkpoint the entire VC, a VM named VM_1 in the VC will record a non-volatile copy named "vm1_global_rn". All of the non-volatile copies from every VM compose a globally consistent checkpoint for the entire VC. Besides, if the VPC Initiator sends the checkpointing request at a user-specified frequency, the correct state of the entire VC is recorded periodically.

V. Evaluation and Results

A. Experimental Environment

Our experimental testbed includes three multicore machines as the primary and backup servers. Each server has 24 AMD Opteron 6168 processors (1.86GHz), and each processor has 12 cores. The total assigned RAM for each server is 11GB. We set up a 1Gbps network connection between the servers for experimental studies. We used two machines as the primary servers, and used the third as the backup server. To evaluate the overhead of VPC (Sections V-E), we set up the VC environment by creating 16 guest VMs (allocated 512MB RAM for each guest VM) on one primary server, and 24 guest VMs (allocated 256MB RAM for each guest VM) on the other primary server. We built Xen 3.4.0 on all servers and let all the guest VMs run PV guests with Linux 2.6.31. Each Domain 0 on the three servers has a 2GB memory allocation, and the remaining memory was left for the guest VMs to use. All the physical machines and the VMs were connected with each other based on the bridging mechanism of Xen.

We refer to our proposed checkpointing design with page fault prediction mechanism as VPC. In Sections V-B, V-D, and V-E, to evaluate the benefits of VPC, we compare VPC with our initial incremental checkpointing design without prediction, which we refer to as VPC-np.

Our competitors include different checkpointing mechanisms including Remus [9], LLM [12], and VNsnap [13]. Remus uses checkpointing to handle hardware fail-stop failures on a single host with whole-system migration. The Lightweight Live Migration or LLM technique improves Remus's overhead while providing comparable availability. While

Application	VPC	VPC-np	VNsnap
idle	$55ms$	$57ms$	$53ms$
Apache	$187ms$	$224ms$	$267ms$
NPB-EP	$179ms$	$254ms$	$324ms$

TABLE I
SOLO VM DOWNTIME COMPARISON.

Remus and LLM implementations are publicly available, VNsnap is not. Thus, we implemented a prototype by using the distributed checkpointing algorithm in VNsnap and used that implementation in our experimental studies.

B. VM Downtime Evaluation

Recall that there are two types of downtime in the VC: VM downtime and the VC downtime. We first consider the case of solo VM to measure the VM downtime. The solo VM case is considered, as it is a special case of the virtual cluster case, and therefore gives us a baseline understanding of how our proposed techniques perform.

As defined in Section I, the VM downtime is the time from when the VM pauses to save for the checkpoint to when the VM resumes. Table I shows the downtime results under VPC, VPC-np, and VNsnap daemon for three cases: i) when the VM is idle, ii) when the VM runs the NPB-EP benchmark program [4], and iii) when the VM runs the Apache web server workload [6]. The downtimes were measured for the same checkpointing interval, with the same VM (with 512MB of RAM) for all three mechanisms.

Several observations are in order regarding the downtime measurements. First, the downtime results of all three mechanisms are short and very similar for the idle case. This is not surprising, as memory updates are rare during idle runs, so the downtime of all mechanisms is short and similar.

Second, when running the NPB-EP program, VPC has much less downtime than the VNsnap daemon (reduction is roughly 45%). This is because, NPB-EP is a computationally intensive workload. Thus, the guest VM memory is updated at high frequency. When saving the checkpoint, compared with other high-frequency checkpointing solutions, the VNsnap daemon takes more time to save larger dirty data due to its low memory transfer frequency.

Third, when running the Apache application, the memory update is not so much as that when running NPB. But the memory update is significantly more than that under the idle run. The results show that VPC has lower downtime than VNsnap daemon (downtime is reduced by roughly 30%).

Finally, compared with VPC-np, VPC also has less downtime when running NPB-EP and Apache (reduction is roughly 30% and 17%, respectively). As for both VPC-np and VPC, the downtime depends on the amount of checkpoint-induced page faults during the checkpointing interval. Since VPC-np uses an incremental checkpointing methodology, and VPC tries to reduce the checkpoint-induced page faults, VPC incurs smaller downtime than VPC-np.

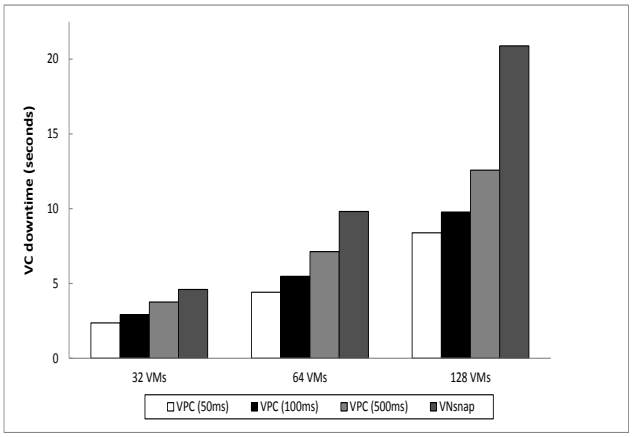

Fig. 4. VC downtime under NPB-EP framework.

C. VC Downtime Evaluation

As defined in Section I, the VC downtime is the time from when the failure was detected in the VC to when the entire VC resumes from the last globally consistent checkpoint. We conducted experiments to measure the VC downtime under 32-node (VM), 64-node, and 128-node environments. We used the NPB-EP benchmark program [4] as the distributed workload on the protected VMs. NPB-EP is a compute-bound MPI benchmark with a few network communications.

To induce failures in the VC, we developed an application program that causes a segmentation failure after executing for a while. This program is launched on several VMs to generate a failure, while the distributed application workload is running in the VC. The protected VC is then rolled back to the last globally consistent checkpoint. A suite of experiments were conducted with VPC deployed at 3 different checkpointing intervals ($500ms$, $100ms$, and $50ms$). We ran the same workloads on VNsnap daemon. We note that in a VC with hundreds of VMs, the total time for resuming all the VMs from a checkpoint may take up to several minutes (under both VPC and VNsnap). In the presentation, we only show the downtime results from when the failure was detected in the VC to when the globally consistent checkpoint is found and is ready to resume the entire VC.

Figure 4 shows the results. From the figure, we observe that, in the 32-node environment, the measured VC downtime under VPC ranges from 2.31 seconds to 3.88 seconds, with an average of 3.13 seconds; in the 64-node environment, the measured VC downtime under VPC ranges from 4.37 seconds to 7.22 seconds, with an average of 5.46 seconds; and in the 128-node environment, the measured VC downtime under VPC ranges from 8.12 seconds to 13.14 seconds, with an average of 11.58 seconds. The corresponding results from VNsnap are 4.7, 10.26, and 22.78 seconds, respectively. Thus, compared with VNsnap, VPC reduces the VC downtime by as much as 50%.

Another observation is that the VC downtime under VPC slightly increases as the checkpointing interval grows. Since

T_c	perlbench	bzip2	gcc	xalancbmk
$VPC-np:50ms$	684	593	991	1780
$VPC-np:100ms$	1389	1231	2090	2824
$VPC-np:500ms$	5345	5523	5769	5428
$VPC:50ms$	826	737	1041	2227
$VPC:100ms$	1574	1411	2349	3572
$VPC:500ms$	6274	5955	6813	7348

TABLE II
VPC CHECKPOINT SIZE MEASUREMENT (IN NUMBER OF MEMORY PAGES) UNDER SPEC CPU2006 BENCHMARK.

we didn't consider the resumption time from the checkpoint, when VPC is deployed with different checkpointing intervals, the VC downtime is determined by the time to transfer all the solo checkpoints from primary servers to the backup server. Therefore, a smaller checkpoint size incurs less transfer time, and thus less VC downtime. The checkpoint size depends on the number of memory pages restored. Therefore, as the checkpointing interval grows, the checkpoint size also grows, so does the number of restored pages during transfer.

D. Memory Overhead

In VPC, each checkpoint consists of only the pages which are updated within a checkpointing interval. We conducted a number of experiments to study the memory overhead of VPC's checkpointing algorithm at different checkpointing intervals. In each of these experiments, we ran four workloads from the SPEC CPU2006 benchmark [5], including: (1) perlbench, which is a scripting language (stripped-down version of Perl v5.8.7); (2) bzip2, which is a compression program (modified to do most work in memory, rather than doing I/O); (3) gcc, which is a compiler program (based on gcc version 3.2); and (4) xalancbmk, which is an XML processing program (a modified version of Xalan-C++, which transforms XML documents to other document types).

For both VPC-np and VPC designs, we measured the number of checkpoint-induced page faults (in terms of the number of memory pages) in every checkpointing interval (e.g., $T_c = 50ms$) of each experiment duration.

Table II shows the results. We observe that for both designs, the average checkpoint sizes are very small: around 2.00% of the size of the entire system state when the checkpointing interval is $50ms$. For example, when VPC-np is deployed with a checkpointing interval of $50ms$, the average checkpoint size is 1012 memory pages or 3.9MB, while the size of the entire system state during the experiment is up to 65,536 memory pages (256MB). The maximum checkpoint size observed is less than 7MB (1780 pages when running the xalancbmk program), which is less than 3% of the entire system state size. When the checkpointing interval is increased to $100ms$, all checkpoints are less than 3,000 pages, the average size is 1883.5 pages (7.36MB, or 2.9% of the entire memory size), and the maximum checkpoint size is about 11MB (2824 pages when running the xalancbmk program). When the checkpointing interval is increased to $500ms$ (i.e., two checkpoints in a second), we observe that all the checkpoint sizes are around 5500 pages, the average size is 5516.25 pages (21.55MB, or 8.4% of the entire memory size), and the maximum checkpoint size is about 22.5MB (5769 pages when running the gcc program). Thus, we observe that, the memory overhead increases as the checkpointing interval grows. This is because, when the interval increases, more updated pages must be recorded during an interval, requiring more memory.

Another observation is that, the checkpoint size under VPC is larger than that under VPC-np. This is because, under VPC, as we improve VPC-np with page faults prediction, it generates more "fake" dirty pages in each checkpointing interval. For example, we pre-make the updated pages in the first checkpointing interval as writable at the beginning of the second checkpointing interval. Thus, at the end of the second checkpointing interval, there are some fake dirty pages, which are actually not updated in the second checkpointing interval. Therefore, besides the dirty pages which are actually updated in the second interval, the checkpoint recorded after the second checkpointing interval also includes these pages which are not updated in the second interval but still set as dirty because of the prediction mechanism. Note that although VPC generates a slightly larger checkpoint, it incurs smaller downtime than VPC-np (Section V-B).

E. Performance Overhead

We measured the performance overhead introduced into the VC by deploying VPC. We chose two distributed programs in the NPB benchmark [4], and ran them on 40 guest VMs with VPC deployed. NPB contains a combination of computational kernels. For our experimental study, we chose to use the EP and IS programs. EP is a compute-bound MPI benchmark with a few network communications, while IS is an I/O-bound MPI benchmark with significant network communications.

A suite of experiments were conducted under the following cases: (1) a baseline case (no checkpoint), (2) VPC deployed with 3 different checkpointing intervals ($500ms$, $100ms$, and $50ms$), (3) Remus [9] deployed with one checkpointing interval ($50ms$), (4) LLM [12] deployed with one checkpointing interval ($50ms$), and (4) VPC-np deployed with one checkpointing interval ($50ms$).

A given program executes with the same input across all experiments. To test the performance accurately, we repeated the experiment with each benchmark five times. The execution times were measured and normalized, and are shown in Figure 5. The normalized execution time is computed by dividing the program execution time with the execution time for the corresponding baseline case.

We first measured the benchmarks' runtime when they were executed on the VMs without any checkpointing functionality. After this, we started the VPC agents for each protected VM, and measured the runtime with different checkpointing intervals. We chose EP Class B program in NPB benchmark and recorded its runtime under different situations. Besides, we also chose EP Class C in NPB benchmark (the Class C problems are several times larger than the Class B problems)

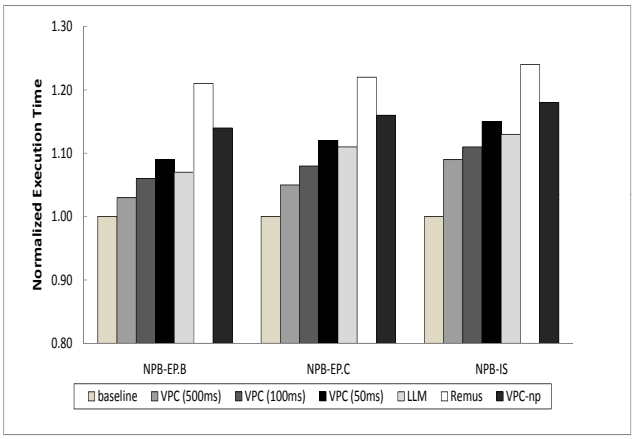

Fig. 5. Performance overhead under NPB benchmark.

to see how VPC performs if we enlarge the problem size. Finally, we tested some extreme cases with I/O intensive applications. We chose the IS program in NPB benchmark, a NPB benchmark with significant network communications, and excluding floating point computations. We ran the same benchmarks on Remus, LLM, and VPC-np, following the same manner as on VPC.

Figure 5 shows the results. We observe that, for all programs running under VPC, the impact of the checkpoint on the program execution time is no more than 16% (the normalized execution times are no more than 1.16), and the average overhead is 12% (the average of the normalized execution times is 1.12), when VPC is deployed with $50ms$ checkpointing interval. When we increase the checkpointing interval to $100ms$, the average overhead becomes 8.8%, and when we increase the checkpointing interval to $500ms$, the average overhead becomes 5.4%. Thus, we observe that the performance overhead decreases as the checkpointing interval grows. Therefore, there exists a trade-off when choosing the checkpointing interval. In VPC, checkpointing with a larger interval incurs smaller overhead, while causing a longer output delay and a larger checkpoint size. (This also means larger memory overhead, confirming our observation in Section V-D.)

Another observation is that VPC incurs lower performance overhead compared with other high-frequency checkpointing mechanisms (Remus and VPC-np). The reason is that memory access locality plays a significant role when the checkpointing interval is short, and VPC precisely reduces the number of page faults in this case. LLM also reduces the performance overhead, but it is a checkpointing mechanism only for solo VM. In our experiment, we deployed 40 VMs in the VC, and from Figure 5, we observe that the overhead of VPC is still comparable to that in the solo VM case under LLM.

VI. Conclusions

We presented VPC, a lightweight, globally consistent checkpointing mechanism that records the correct state of an entire VC, which consists of multiple VMs connected by a virtual network. The design, implementation, and experimental evaluation of VPC shows that by applying a high frequency checkpointing mechanism and recording only the updated memory pages during each small checkpointing interval, the downtime can be reduced with acceptable memory overhead. Additional reduction in the downtime can be obtained by predicting the checkpoint-caused page faults during each checkpointing interval. Based on this lightweight design under VPC, the global consistency in the entire VC is ensured by modifying and implementing a classic distributed algorithm.

References

[1] Amazon EC2. http://aws.amazon.com/ec2/.
[2] Google App Engine. http://code.google.com/appengine/.
[3] KVM: Kernel based virtual machine. www.redhat.com/f/pdf/rhev/DOC-KVM.pdf.
[4] NAS PARALLEL BENCHMARKS. http://www.nas.nasa.gov/Resources/Software/npb.html.
[5] SPEC CPU2006. http://www.spec.org/cpu2006/.
[6] The Apache HTTP Server Project. http://httpd.apache.org/.
[7] P. Barham, B. Dragovic, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, USA, 2003.
[8] C. Clark, K. Fraser, and A. Warfield. Live migration of virtual machines. In *NSDI*, pages 273–286, USA, 2005.
[9] B. Cully et al. Remus: high availability via asynchronous virtual machine replication. In *NSDI*, pages 161–174, USA, 2008.
[10] W. Gu et al. Error sensitivity of the linux kernel executing on PowerPC G4 and Pentium 4 processors. In *DSN*, pages 887–896, 2004.
[11] J. Heo, S. Yi, and S. Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC*, pages 1558–1562, USA, 2005.
[12] B. Jiang, B. Ravindran, and C. Kim. Lightweight live migration for high availability cluster service. In *SSS*, USA, 2010.
[13] A. Kangarlou et al. VNsnap: Taking snapshots of virtual networked environments with minimal downtime. In *DSN*, pages 524 –533, 2009.
[14] M.-L. Li et al. Understanding the propagation of hard errors to software and implications for resilient system design. In *ASPLOS*, pages 265–276, USA, 2008.
[15] I. Lyubashevskiy and V. Strumpen. Fault-tolerant file-I/O for portable checkpointing systems. *J. Supercomput.*, 16:69–92, May 2000.
[16] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *JPDC*, 18:423–434, 1993.
[17] D. T. Meyer, G. Aggarwal, and A. Warfield. Parallax: virtual disks for virtual machines. In *EuroSys*, USA, 2008.
[18] D. S. Milojicic, F. Douglis, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32:241–299, 2000.
[19] M. Nelson, B. H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX ATC*, pages 25–25, USA, 2005.
[20] S. Osman, D. Subhraveti, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36:361–376, 2002.
[21] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. Automated derivation of application-aware error detectors using static analysis. In *13th IEEE International On-Line Testing Symposium*, pages 211 –216, 2007.
[22] D. Pei. Modification operation buffering: A low-overhead approach to checkpoint user files. In *IEEE 29th Symposium on Fault-Tolerant Computing*, pages 36–38, 1999.
[23] J. Plank, J. S. Plank, and K. Li. Libckpt: Transparent checkpointing under Unix. In *USENIX Winter Technical Conference*, pages 213–223, 1995.
[24] J. S. Plank et al. Memory exclusion: optimizing the performance of checkpointing systems. *Softw. Pract. Exper.*, 29:125–142, February 1999.
[25] Virginia Tech Systems Software Research Group. Project Resilire: High availability virtualization. http://107.20.237.171/resilire, 2010.
[26] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.
[27] W. Zhao, Z. Wang, and Y. Luo. Dynamic memory balancing for virtual machines. *SIGOPS Oper. Syst. Rev.*, 43:37–47, July 2009.