2012 IEEE 14th International Conference on High Performance Computing and Communications

Radiata: Enabling Whole System Hot-mirroring via Continual State Replication

Yang Chen, Chunming Hu, Tianyu Wo
School of Computer Science and Engineering
Beihang University
Beijing 100191, China
{chenyang, hucm, woty}@act.buaa.edu.cn

***Abstract*—Checkpoint-recovery based on system virtualization is an attractive approach for providing the transparent and economic fault tolerance service in virtualized environments. The previous approaches introduce either great performance degradation or complex implementation issues. In this work, we propose a whole system hot-mirroring platform, namely Radiata, to provide fault-tolerance for any type of service by encapsulating the service instance into a virtual machine, and hot-mirroring the state changes of the virtual machine via the continual state replication. Our approach exploits three key optimizations for further reduction of the performance overhead: the asynchronous state replication, the COW-based memory checkpoint and the dirty page prediction. Based on the KVM platform, we have implemented the prototype system. The comprehensive evaluations under a variety of workloads demonstrate that Radiata is able to effectively support rapid and transparent fail-over in case of unexpected hardware failure, and outperforms the existing mechanisms in terms of the performance degradation in failure-free condition.**

Keywords-fault tolerance; hot-mirroring; virtual machine; continual replication; COW-base checkpoint;

I. INTRODUCTION

With the prevalence of the virtualization technology in the contemporary IT infrastructures, more and more services are consolidated into virtualized environments based on commodity servers [22]. By leveraging the encapsulation, isolation and flexibility of virtual machine (VM), the service providers are able to achieve greater utilization of physical resources. However, the hardware failure for commodity servers is a common case rather than an exception [24] in IT infrastractures. Due to server consolidation, a hardware failure over one physical server consequently lead to multiple service outages, the violation of SLA (Service Level Agreement)with customers, and economic penalty and revenue loss.

Whole system hot-mirroring can be used as a common approach for providing fault tolerance services, especially for mission-critical applications. It works in primary-backup mode, in which the state of the backup is kept almost synchronous with the primary at all time, so that a fail-over can be immediately performed to the backup in the event of failures over the primary. However, the commercial fault tolerance services regularly require specialized hardware configurations or software instrumentations. On the other hand, the approaches of rebooting the VMs based on the shared VM disk or restoring the states of VMs from the checkpoint files are not capable of hiding the service failures to the external clients. Furthermore, the rebooting or restoring approaches can be space and time consuming, which impose inevitable waiting time before the VM loading correct state.

System state replication at the Virtual Machine Monitor (VMM) layer is a promising solution to provide fault tolerance services. VMM takes advantage of the privileged control over VMs, to achieve transparent and cost-efficient replication of the system state regardless of the application or the operating system. Unfortunately, existing approaches either potentially entail great performance degradation or encounter practical and complex implementation issues.

Deterministic replay based approach [3, 9] records the external inputs and non-deterministic events of the primary VM on the backup VM in case of hardware failure. While this approach deeply depends on the specific architecture implementation, and also sustains unacceptable performance degradation when it be applied to multi-processor VMs, as tracking the shared memory accesses by the multi-core CPU can be non-deterministic events [23]. State replication based approaches, such as Remus [4] and Kemari [5], evade the multi-core problem by capturing and replicating the state changes of the primary VM. The incremental checkpoints are repeatedly taken and sent to the backup at a high frequency. However, these works introduce substantial performance degradation to the protected service. For instance, Remus may sustain a 103% performance degradation under some workload, and the primary cause for the degradation is its deficient memory checkpoint scheme [7, 19]. Kemari is reported that it entails steeper performance overhead due to its transactional checkpoint mode [6].

In this paper, we propose a cost-efficient whole system hot-mirroring platform, namely *Radiata*, to provide transparent fault tolerance for any service over unexpected physical hardware failures. Our primary design objective is to support seamless failover with negligible performance overhead, only based on commodity hardware and software. In essence, our approach takes advantage of the inherent capabilities of virtualization to encapsulate the protected services into VMs (the replicated VMs), and hot-mirror the system state changes by capturing and replicating the incremental checkpoints to the backup VMs. A checkpoint data contains of the states of a VM's CPU, memory, and other devices at a given time point. In the event of hardware failures on the primary, the backup would rapidly and seamlessly take over the affected services.

For greater physical separation between the replicate and backup VMs, we eliminate the dependency of the shared storage configuration. We also extend the key points of the live migration mechanism, to take checkpoint as frequent as

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.70

tens per second. In order to mitigate the service interruptions, the state replications are performed asynchronously with execution of the replicated VM. The outward network packets issued by the replicated VM are transparently intercepted and delayed releasing according to the output buffering protocol, as we should ensure that the backup could be restored from a state where the outward packets are issued despite any failures. For keeping the VM downtimes small and constant, we also exploit the COW-based memory checkpoint scheme, in which the state changes of memory are captured immediately on a memory write fault handling, rather than performing memory pages copying while the VM is suspended. In order to further improve the memory checkpoint performance, we exploit dirty page prediction scheme to reduce the overhead of write page faults handling. Historical memory access patterns are used as heuristic information to predict the pages that about to be modified within the current execution interval. We have implemented the proposed schemes on the Linux Kernel Virtual Machine (KVM) platform, and performed the evaluations under a variety of workloads. The results show that Radiata could achieve a reasonable performance comparing to the native VM.

The remainder of this paper is organized as follows. The next section reviews existing research on fault tolerance based on system virtualization. Section 3 elaborates the design rationale of Radiata. Section 4 details implementation of Radiata system. The comprehensive evaluation results are presented and analyzed in Section 5. Finally, Section 6 presents the paper's conclusions and our future work.

II. RELATED WORK

Great efforts have been taken towards providing fault tolerance service based on system virtualization technology. These works can be generally categorized into *deterministic replay* based and *checkpoint-recovery* based approaches. Bressoud and Schneider [3] firstly proposed the idea of providing fault tolerance based on deterministic replay by keeping the backup VM executing in sync with the primary VM in *lock-step* mode. They recorded all the inputs and non-deterministic events on the primary VM, and correctly replayed on the backup VM in case of unexpected hardware failure. Recent work introduced by the VMware also has implemented the similar mechanism on vSphere 4.0 platform for the fully virtualized x86-VMs [9]. Comparing to the checkpoint-recovery solution, deterministic replay based approach requires less network bandwidth consumption for state synchronization. However, it highly depends on the architecture-specific implementation. Moreover, it suffers unacceptable performance issues when it's applied to multi-processor VMs, because the shared memory accesses by the multi-core CPU can be non-deterministic events. It is fundamentally difficult to efficiently track and record the order of shared memory accesses. Recently, Respec [11] implemented the deterministic relay of multithreaded applications with improved performance by lazily increasing the level of synchronization it enforces depending on whether it observes divergence during replay, whereas it restricts the execution replay to be performed on the same physical system, making it inappropriate for separate primary and backup configuration of fault tolerance service.

Checkpoint-recovery based solutions such as Kemari [5] and Remus [4] achieve state mirroring by frequently replicating the checkpoints of the primary VM to the backup. Remus firstly demonstrated the feasibility of keeping the state of the backup in sync by high-frequency system state replication. It utilizes the asynchronous replication and speculative execution processing to alleviate the performance overhead, but as a primitive demonstration it yet sustains serious performance degradation of the primary VM [6, 7, 19]. Although the authors recently identified two sources of the performance overhead in RemusDB [19], and tried to reduce both the amount of state transferred and the network latency due to the output buffering mechanism, their improvements are mainly against the database workload, and require source instrumentations to the replicated services (i.e., database management system). Similar with previous work [13, 14], Remus delays the outward network packets releasing until the corresponding checkpoint acknowledged by the backup, in order to ensure the consistency of the external state. Rather than taking checkpoints in a static interval, Kemari makes checkpoints whenever the primary VM is about to handle an external I/O event, such as network, disk and mouse et al.. Meanwhile, Kemari removes the VM disks into networked storage, and configures the primary and backup VMs with a sharing disk. By this means, Kemari eliminates both the network packets buffering, and disk state replication. However, it suffers from a steeper performance cost, due to troublesome interruptions caused by intensive checkpoints driven by the external I/O events.

Some works recently have been taken for in-depth studying and improving the performance cost of checkpoint-recovery based solutions. Zhu et al. [7] improved the performance of the log dirty mode by reducing read and write page faults handling. Lu et al. [6] proposed fine-grained dirty region tracking and speculative state transferring to reduce the amount of replication traffic. Both two works mainly focus on the reducing performance cost of memory checkpoint. However, the performance results are attained under the condition with other primary features disabled, such as network packets buffering, and non-sharing disk, it is not clear whether their approach is feasible for the high-frequency checkpoint performance under the integrated system configuration similar with Remus and Radiata.

The major virtualization platforms, such as Xen [2], KVM [1], and VMware [17], also provide a primitive disk-based checkpoint mechanism, by which the instant system state is captured and written into a disk file. In this way, the transient memory and internal system state can be kept in the persistent storages. Whereas, restoring VM state by loading the checkpoint state from a disk file can be a time consuming process, and also loses the connection states with service clients which were active while the checkpoint was taken. Hou et al. [22] and Sun et al. [23] have proposed their prototype systems which leverages disk-based checkpoint to provide high availability facility. They drew on the COW-based checkpoint and lazily restoring approaches to shorten the durations of checkpoint and restoring, respectively.

Nevertheless, as the consistent disk state is required to be captured along with the system state, the checkpoint frequency is lower than that of Radiata (tens per second). Furthermore, taking disk checkpoints currently relies on the snapshot capability of the physical storage system, such as LVM [8] logical volumes.

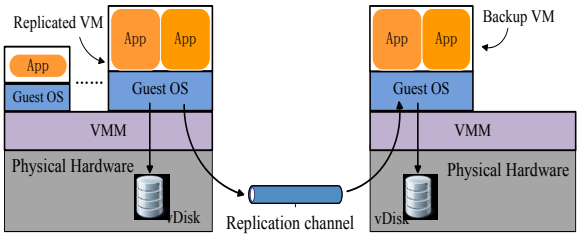

Fig. 1 The configuration of Radiata

III. DESIGN RATIONALE AND RADIATA OVERVIEW

A. Basic Design

Based on commodity hardware, Radiata is designed to mirror the state of the entire system. Essentially, the basic concept of Radiata is to efficiently track and instantly propagate the state changes to the backup VM. In this paper, we exploit the continual state replication, which extends the abilities of disk-based checkpoint and live migration of VM, to capture the incremental checkpoints of the replicated VM at a high frequency, and continually propagates these checkpoint data to the backup via the physical network. We introduce the notion of *epoch* [3, 4] as a static time interval, similar with previous work. The normal execution of the replicated VM is divided into epochs, within which the system state changes are captured. The length of epochs ranges from tens of milliseconds to one second.

In general, Radiata mainly takes into consideration the following three primary objectives.

a) **Transparency and generality.** The hot-mirroring should be taken as an OS and application agnostic service, and does not require any specialized hardware or software.

b) **Rapid and consistent recovery.** In the event of hardware failures on the primary, the backup should rapidly take over both the service and the external connections.

c) **Lightweight overhead.** The hot-mirroring service should not introduce substantial performance degradation to the protected services.

Figure 1 illustrates the basic configuration of Radaita system. The protected service is encapsulated into the replicated and backup VMs, which are respectively hosted in independent physical servers. The servers are connected by the replication channel, through which the checkpoints are continually transferred from the replicated VM to the backup. At the very beginning, the two VMs arrive at the identical state by the whole system live migration from the primary to the backup. Different from the live migration, the replicated VM continues its execution after migration, and the backup enters checkpoint-loading mode. The replication process on the replicated site, continually takes checkpoints at a high frequency, and propagates the checkpoint data to the backup through the channel. In order to mitigate service interruption, the replication of checkpoint data should be performed asynchronously with the normal execution of the replicated VM. On the other hand, the backup VM is kept suspended, and periodically loaded the most recent state disseminated from the replicated site. Once the backup detects any failure happening on the replicated VM, it immediately stops the passive checkpoint-loading and takes over execution with the most recent system state.

In order to detect the hardware failure on the replicated or backup VM, we exploit implicit connection state monitoring on the replication channel. If the backup detects the deviant timeout of the channel, it assumes that the replicated has failed and goes live directly; while the replicated would stop the replication process if it was conscious of the abnormal timeout of the backup reply to the checkpoint data stream. Since the replicated and backup VMs keep communication at a high frequency (i.e., tens per second), a timeout on the channel is detected, the corresponding site can take the appropriate measures to response the failure just within several epoch intervals.

Different of previous work [5, 7], in this paper, we consider the state of VM disk as the internal state. We do not rely on any networked storage for eliminating the state replication of disk blocks. The both VMs are configured with an independent virtual disk respectively, and the backup exactly mirror the virtual disk of the replicated VM. By this approach, Radiata allows us for the possibility of greater physical separation between the replicated and backup, and provides more reliability than the single shared storage. Furthermore, in the shared disk case, since the disk should be treated as an external device, the write operations on the replicated should be held until the checkpoint acknowledgement has been received from the backup end [7, 10]. This buffering approach inevitably causes the significant performance degradations of disk throughput. Conversely, Radiata keeps tacking of disk blocks on the replicated VM, and periodically synchronizes its incremental changes. In this way, the disk write operations can be directly released to the virtual disk on the replicated site. A direct benefit of this method is that it ensures an appreciable disk performance perceived by the internal service of the replicated VM.

B. Replication Protocol

For the internal devices, such as memory, CPU and disk blocks, et al., we can exploit the continual replication to capture their incremental checkpoints, and asynchronously replicate the checkpoints to the backup. However, the external state, i.e., the network connections, introduces a rather different issue, as the backup is not aware of how many and when network packets have been released by the replicated VM. Nevertheless, it requires that no externally perceived state or data is lost whenever the backup takes over the service. It requires that Radiata should keep the internal state loaded into the backup is consistent with the external state perceived by the service clients. One method fulfilling this requirement is to delay outward network packets releasing until the backup has received the complete

checkpoint data which generated those packets. In this way, the backup could be resumed with the most recent internal state, which is consistent with the external output delivered by the replicated before the failure occurs. In this work, we adopt *the output buffering protocol*, in some way similar with previous works [13, 14].

Restricted by the protocol, the outward network packets issued by the replicated VM, would be held until the complete checkpoint has been received and acknowledged by the backup. Consequently, Radiata should intercept and buffer all the outward network packets on the replicated site. Nevertheless packet interception and buffering should not interrupt the normal execution of the replicated VM. We leverage the VMM's privileged controls to transparently intercept the packets issued by the replicated VM. In this way, the replicated VM would be agnostic to the packet interception, and not immediately affected by delayed release of packets, as long as the epoch is kept short enough.

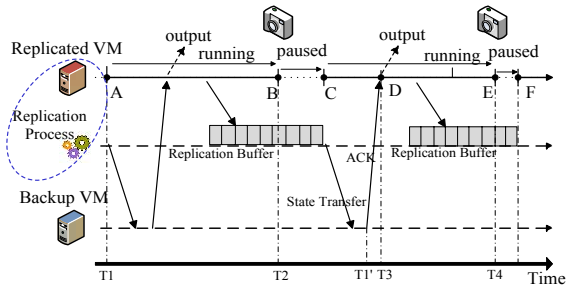

Fig. 2 Continual state replication process of Radiata

We depicts the continual replication process in Figure 2. At the beginning of each epoch (e.g., A and C), Radiata resets the dirty tracking mode on the replicated VM. Within an epoch (e.g., between A and B), Radiata tracks the modifications to the memory pages and disk blocks. At the same time, all the outwards packets released are captured and appended into the queuing buffer. At the end of each epoch (e.g., B and E), the replicated VM would be paused for taking a checkpoint of instant state for its virtual devices. It is worth noting that the replicated would be immediately resumed after the checkpoint data have been flushed into the replication buffer, instead of waiting for the checkpoint being propagated to the backup. Whenever receiving the acknowledgement for a given checkpoint (e.g., D), Radiata commits the buffered outward packets generated by that checkpoint. Suppose that a failure would occur immediately after packets releasing, the backup should drop the incomplete checkpoint data and load the most recent state, that consistent with the replicated VM's last outward packets commitment. However, outwards packets buffered after the last releasing would be lost along with the crash. Nevertheless, the QoS-aware network protocols, such as TCP/IP, can recover the connection state even after packets loss or duplication. Therefore, those lost packets would be naturally treated as transient network errors, from which the network connections and applications could easily self-recover.

C. *Capture Memory Changes*

In order to track changes of the memory state, we extend the dirty memory-tracking scheme, which is originally designed to equip the VM for disk-based checkpoint and live migration. For live migration based on pre-copy mechanism [12, 15], the dirty memory pages are captured and transferred to the destination hosts in iterated phases during the process of migration. The memory-tracking mode is reset at the beginning of each iteration phase, by enabling all of a VM's memory pages write-protection. Within each iteration phase, any first write to a write-protected page causes a page fault, and is trapped into the VMM, wherein the fault page is marked as dirty, and then restored with write permission. However, the existing dirty memory-tracking scheme does introduce nontrivial overhead if it is directly applied in high-frequency memory checkpoint. This is because that the scheme introduces substantial page faults handling and context switching, and leads to inevitable execution interruptions to the VM [12, 16]. Furthermore, comparing to that of live migration (e.g., tens of seconds), the duration of failure-free in hot-mirroring scenario is normally a long-term condition, wherein the substantial performance degradation is not acceptable.

On the other hand, the time and spatial locality of memory access [18] could provide the probability for alleviating intensive page faults handling caused by the dirty memory-tracking. To mitigate the service interruption caused by the page fault handling, we exploit the dirty page prediction approach to reduce the number of write page faults. When a write page fault that triggered by the dirty tracking mode is intercepted by the VMM, we would predict the pages that about to be modified within the current epoch (in Section IV.B). Once a page is predicted to be write accessed, it would be marked as dirty, and restored write permission in advance. However, the false positive of the prediction is inevitable. It results in additional pages added into the memory checkpoint, and more network bandwidth consumption. In fact, it is fundamentally challenging to accurately predict the memory access, we use historical memory write access pattern as heuristic information to control the size of predicted pages. The overall performance showed by evaluations, validates the efficiency of our approach for attaining a good balance between influence from false positive prediction and dirty memory-tracking.

D. *COW-based Memory Checkpoint*

In Radiata, the continual replication process has to interrupt the normal execution of the replicated VM at the end of each epoch, for capturing the instant states of CPU, and other virtual devices. In previous work [4, 5], the replicated VM have to be paused until the overall dirty memory pages along with the checkpoints of other devices has been flushed into the memory buffer. Flushing great volume of checkpoint data into the buffer still takes a substantial amount of time, which inevitably interrupt the normal execution of the protected service (e.g., tens of milliseconds). Furthermore, the pause-and-copy approach results in that the pause time is potentially proportional to the volume of checkpoint data generated in the each epoch. The

majority of an incremental checkpoint data comes from dirty memory pages and disk blocks. Especially, under memory intensive workloads, a considerable volume of memory checkpoint data is generated within each epoch, which is needed to be timely copied into the buffer. Subsequently, this approach results in that the pause time tightly depends on the update rate of VM memory within each epoch.

In order to keep the pause time constant and small, even under the intensive memory write workload, we exploit the COW (Copy-on-Write) based memory checkpoint to capture the memory state changes within each epoch, rather than to copy dirty pages at the end of each epoch. With the aid of dirty memory-tracking scheme, we could intercept any first write access to a memory page, and copy the original content of page into the replication buffer before recovering write permission to the page. In this way, we could eliminate the memory page copying during the VM pause time, and keep the service interruptions to a relatively constant value.

One direct result due to the COW-based memory checkpoint is that the state of memory lightly lags behind other devices in a given checkpoint. This is because an memory checkpoint consists of original contents of pages that are modified within an epoch. While for other devices, we capture their instant states at the end of epoch. This inconsistency between the memory and other devices requires that the backup should accurately load states of each virtual devices corresponding to a given checkpoint. On the backup, once receiving a completed checkpoint data, Radiata firstly appends it into the memory buffer instead of loading it into the backup VM. Therefore, when the fail-over occurs to the primary VM, Radiata should abandon the inconsistent checkpoint data remaining in the buffer, and immediately resume the backup with the state that consists of the most recent memory state. It is worth noting that the implementation of COW-based memory checkpoint is taken on the VMM layer, mainly concentrated on the memory virtualization module. In this way, we could ensure the memory checkpoint is completely transparent to the replicated VM and services, without any assistance from or instrumentation to the Guest OS or service.

IV. IMPLEMENTATION DETAILS

In this work, we have implemented the Radiata prototype based on the Linux Kernel Virtual Machine (KVM [1]) platform. KVM exploits the advantage of the hardware virtualization extensions to add the VMM (or hypervisor) capability to Linux system. KVM platform consists of two components, i.e., the kernel module driver *kvm-kmod* and the user space tool *qemu-kvm*.

We mainly modified the *qemu-kvm* for state replication functionality, in which the *replication controller* and *checkpoint loader* components reside, respectively on the primary and backup sites. The other key component added into *qemu-kvm* of the primary is the *checkpoint controller*, which is responsible for periodically taking checkpoint of the VM, and capturing the memory modifications within each epoch. We also extended the types of *QEMUFile* object, by associating it with the memory buffer. In the kernel space, we mainly modified the MMU virtualization component *kvm_mmu* of the *kvm-kmod* module, in order to perform the COW-based memory checkpoint and dirty pages prediction. All the above instrumentations are taken on the VMM layer, which are transparent to the VMs and their internal services.

A. Asynchronous replication

At the very beginning of the continual state replication, Radiata initializes the memory buffers *crep_buffer* on both VMs sites, and associate them with the *QEMUFile* objects. Subsequently, the checkpoint controller on the primary site starts periodical checkpoints according to the pre-set interval. Whenever a checkpoint has been taken, the replication controller is awakened to write the checkpoint data into the replication channel in parallel with the VM execution.

On the backup, the checkpoint loader reads checkpoint data with the help of the *QEMUFile* object referring to the socket connection. Whenever it extracts a checkpoint data, it appends the data into the *crep_buffer*, from which the system state is loaded. As the state of the memory lags behind other devices in a given checkpoint, the loader firstly loads the instant states of other devices that already residing in the *crep_buffer*, and then it applies the latest memory checkpoint. If a hardware failure occurs to the replicated VM during the checkpoint data reading, the loader would directly destroy the *crep_buffer*, and resume the backup VM with a consistent state. On the other hand, if a failure occurs within the state loading process, the backup is still able to load a consistent state from the *crep_buffer*, as the most recent system state has already resided in the local memory buffer.

B. Memory checkpoint

For predicting the write accesses to memory pages within a given epoch, we implemented a prediction scheme based on the historical write access pattern. On the replicated site, Radiata calculates the average length of consecutive dirty pages *avg_slice_len* at the end of each epoch. To describe the varying trends of *avg_slice_len* between the consecutive epochs, Radiata maintains the value *delta_slice_len* for each epoch, which is defined as:

$$delta\_slice\_len_n = avg\_slice\_len_n - avg\_slice\_len_{n-1} \quad (1)$$

Given that, we also define the predicated dirty pages size for the n^{th} epoch:

$$prelice\_len_n = \alpha * avg\_slice\_len_{n-1} + (1-\alpha) * delta\_slice\_len_{n-1} \quad (2)$$

where the value α denotes the weight of write access behavior in the previous epoch. In this paper, we set $\alpha = 0.6$ by default. Our scheme is in some way analogous to that discussed in [7] with difference that we use the dirty pages instead of shadow page entries for eliminating duplicated SPT scanning process, and we add the varying trends of write access behavior for refining the predicted size.

We modified the page fault handler in *kvm_mmu* to capture the first write accesse to the memory pages within each epoch. Specifically, whenever the handler intercepts a write page fault triggered by a valid page, it firstly locates the faulting page according to the guest physical address. Subsequently, the handler predicts the pages that within

prelice_len forwards as dirty. After that the *kvm-kmod* surrenders the execution to the *qemu-kvm*, in which the checkpoint controller copies the contents of faulting and predicted pages into the buffer, and finally resumes the replicated VM. Note that the above historical information is counted at the end of each epoch, and only based on the statistic information of dirty pages from the *kvm-kmod* (i.e., via *ioctl* system calls). The calculation process does incur trivial computation overhead. We enable the checkpoint controller in the user space to pass historical properties of dirty memory pages by extending the existing *ioctls* of KVM.

C. Network buffering and disk blocks replication

For buffering the outward network packets, we modified *network queuing* in *qemu-kvm*. Specifically, we disabled automatical packets delivering of the *NetQueue* object. All the outward packets issued by the replicated VM would be intercepted and appended into a modified *NetQueue* object. In order to distinguish packets generated within different epochs, we tagged each packets with its checkpoint IDs. Whenever receiving the checkpoint acknowledgement from the backup, the replication controller would flush the packets corresponding to the checkpoint ID into the physical network.

Currently, we capture the updates of disk blocks by extending the existing block tracking scheme of live block migration. At the beginning of each epoch, we empty the dirty block lists, and dump the changes of disk blocks into the *crep_buffer* at the end of each epoch. As we keep the execution epoch short enough, the amount of changes to disk blocks within an epoch is minor to the memory checkpoint. Although we apply pause-and-copy approach for dirty blocks checkpoint, it does not make a great impact on the service interruption. Similar with the checkpoint of other devices, the backup buffers the changes of disk blocks corresponding to a given incremental checkpoint.

V. EVALUATION

In this section, we present evaluation results and related analysis of our Radiata prototype under various types of workloads. Our objective is to evaluate the effectiveness and efficiency of our Radiata, in terms of the performance degradation of internal services in failure-free condition.

A. Evaluation Setup

We deployed Radiata prototype on the DELL T1500 workstations, consisting of one 2.66 GHz Intel Core i5 750 CPU, 8G of RAM, 320G STAT hard driver, and 1 Broadcom BCM57780 Gigabit network interface. The host machines are installed Ubuntu server 10.04 TLS on Linux kernel 2.6.32-24, and we used qemu-kvm 0.12.3 and kvm-kmod 2.6.32.27 modified as described in Section IV. We also configured the virtual machines with one virtual CPU, 1G of RAM, and 20G virtual disk. The VMs were bridged to the physical network, and enabled with the *virtio* drivers.

We performed evaluations under a variety of workloads, and focused on the VM performance degradations of disk I/O, memory and CPU. The evaluation workloads consisted of Linux kernel compilation, iozone benchmark, customized memory stress and static web test. Wherein, the kernel compilation is a system-call intensive workload, and also a balanced workload for CPU, memory and disk, the iozone benchmark is a disk I/O intensive workload, and the memory stress workload is an intensive memory updating program, which updates a working set of memory with a specific rate.

B. Functionality verification

To verify the intended functionality of our system, we firstly validated the correctness of the system under different workloads described above. In each verification test, we connected to the replicated VM via SSH from an external machine. Simultaneously, a daemon program inside the replicated VM continually *ping* an external server. The epoch of Radiata was set to 25ms, and we injected artificial failure by mandatorily killing the replicated VM process. For a given workload, each test was repeated four times.

At every failure point, the backup VM went live instantly (within one second), and both its internal *ping* program and workload application were kept active without noticeable interruption. The SSH connection from the external machine was also kept connected. Furthermore, after the fail-over to the backup, we used the *fsck* tool to check and optionally repair the ext3 file system on the backup VM disk. For each test, the *fsck* tool returned no error.

C. Performance

1) *Memory stress workload:* Firstly, we focused on the impact of memory updating rate on the performance, we developed a memory stress workload that updated the memory pages at the specific rate. It firstly allocats a changeable working set size, and then repeatedly touches all the memory pages by writing a random value into the first byte of each page. Figure 3 illustrates the performance variation with that of the native VM when the execution epoch was set to be 25ms. It shows that the performance placidly decreases from 64% to 54% as the working set size increasing from 1MB/s to 512MB/s. The results indicate that Radiata is insensitive to the memory updating rate. We attribute it to the optimization from the COW-based memory checkpoint and dirty page prediction. On the one hand dirty page prediction scheme skips the potential write fault handling; on the other hand the COW-based checkpoint relieves the suspending interruption at the end of each epoch.

Figure 4 illustrates the performance of the memory workload with different epoch configurations under the fixed working set size 16MB/s. It shows that the workload performance gradually increases from 58% to 61.2% as the execution epoch grows from 25ms to 100ms. Although the amount of the checkoint data increase as the execution epoch grows, the data is transferred asynchronously, overlapping the VM's execution in the subsequent epoch. Furthermore, we observed that the dirty page prediction does relieve overhead of the page fault handling caused by the dirty-memory tracking. This set of workload also reflects from another perspective that the performance overhead caused by the state replication could stable, even under memory intensive workload. We believe that the performance in this set of workload is reasonable for a general-purpose service,

as the realistic service workloads do not usually achieve these extreme memory updating rates.

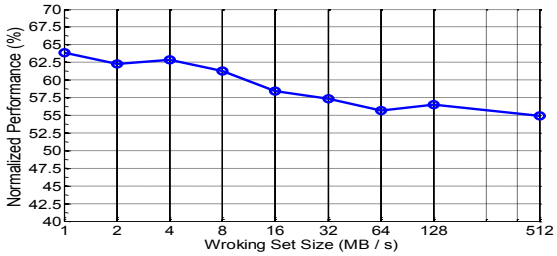

Fig. 3 Memory stress workload (Epoch = 25ms)

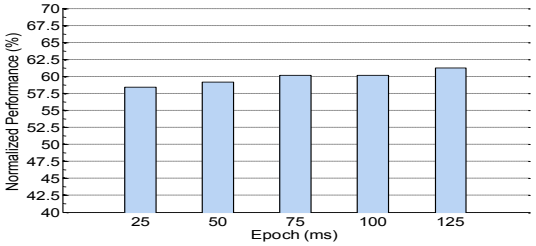

Fig. 4 Performance of memory stress (WSS = 16M/s)

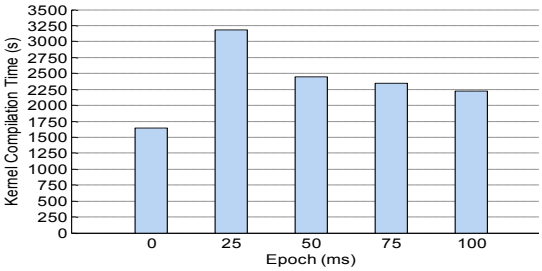

Fig. 5 Kernel compilation time

2) *Kernel compilation:* We also measured the average wall-clock time required to compile the Linux kernel version 2.6.32 with the default configuration. For each epoch setting, we repeated compilation three times.

Figure 5 shows the quantitative comparison for each epoch configuration. Comparing to the native VM, the performance overhead was 92.8%, 48%, 42% and 34.9% respectively, as epoch increased from 25ms to 100ms (e.g., checkpoint frequency decrease from 40 to 10 times per second). As the execution epcoch grows, the compile workload is assigned with the reduced resouce in terms of CPU and I/O operations, due to that the replication process requires more capabilities to transfer checkpoint data. Although the replication process does impose a certain level of performance degradation, however, Radiata achieves significant performance improvements comparing to over 103% degradation from the previsou work Remus [4], under the identical evaluation.

3) *Iozone:* To understand the performance degradation of disk I/O, we performed the evaluations with iozone benchmark, which is sensitive to both disk throughput and response. In order to factually and completely reflect the effects of the state replication on the disk I/O, we enabled memory and network protections during these evalutations, which was different from the previous work.

In Figure 6, we compared the WRITE and READ performance with different record sizes settings. Similar with kernel compilation evaluation, the iozone performance increases as the checkpoint frequency decreases. As the execution epoch grows, the iozone could be assigned more resource for intensive disk I/O operations during each execution epoch. When the epoch was set to be 25ms, the both read and write performance dropped to the worst case 35% (record size 512KB) and 40% (record size 64KB) respectively. The iozone in this case gains the lowest CPU and I/O resource to satisfy the intensive disk I/O operations.

4) *Network latency:* The last workload we investigated was the static web test. In these evaluations, we configured both the primary and backup VM as a static Apache web server for answering a 256KB file downloading requests from 100 simultaneous connections.

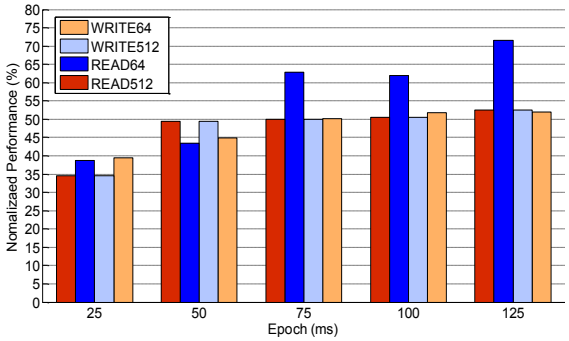

Fig. 6 Performance of iozone

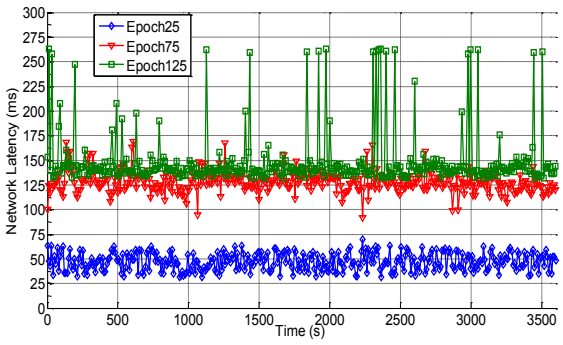

Fig. 7 Network latency

Figure 7 plots the network latency at a high frequency for different epoch settings. As seen, Radiata imposes a certain level of network latency increament. As the outward network packets must be buffered until the corresponding checkpoint has been acknowledged from the backup site, the measured latency can be slightly higher than the configured epoch. For an example, the average latency is about 140ms when the epoch is set to be 125ms. In general, the larger the epoch results in the lower memory checkpoint overhead, the higher the network latency and the lower throughput as perceived

by the external world. Therefore, how to keep the epoch short while minimizing the replication overhead is a key challenge for Radiata alike system.

VI. CONCLUSTOIN

Checkpoint-recovery based system state replication is a promising approach for providing fault tolerance service for the entire system stack residing in the VM. In this paper, we have proposed a hot-mirroring platform to provide the transparent and economic fault tolerance capability for any type service by encapsulating the service instance into a virtual machine. We exploited the continual state replication to mirror the system state changes in the backup so that the latter can take over in case of hardware failures over the primary site. The system state replication is performed asynchronously with the service execution in order to mitigate the service interruptions. We also adopt the COW-based memory checkpoint and the dirty page prediction schemes to further alleviate the state replication overhead.

The evaluation results demonstrate the effectiveness and efficiency of our system. In case of a fail-over, the backup can rapidly take over within one second. Evaluation results also have showed that our approaches do not introduce excessive overhead. Future work will focus on reduction of the replication data, such as the lightweight compression method and redundant exclusion for further performance improvements

ACKNOWLEDGMENT

This work is partially supported by grants from China 863 High-tech Program (No. 2011AA010500), China 973 Fundamental R&D Program (No. 2011CB302602) and National Natural Science Funds for Distinguished Young Scholar (No. 91018004, 90818028).

REFERENCES

[1] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux virtual machine monitor,” in Ottawa Linux Symposium, July 2007, pp. 225–230.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles. NewYork, NY, USA: ACM, 2003, pp. 164–177.

[3] T. Bressoud and F. B. Schneider, “Hypervisor-based fault tolerance,” in Proceedings of the fifteenth ACM symposium on Operating systems principles, ser. SOSP ’95. New York, NY, USA: ACM, 1995, pp. 1–11.

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, “Remus: high availability via asynchronous virtual machine replication,” in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, ser. NSDI’08, 2008, pp. 161–174.

[5] Y. Tamura, “Kemari: Virtual Machine Synchronization for Fault Tolerance using DomT,” NTT Cyber Space Labs, Tech. Rep., 2008.

[6] M. Lu and T. cker Chiueh, “Fast memory state synchronization for virtualization-based fault tolerance,” in Dependable Systems Networks, 2009. DSN ’09. IEEE/IFIP International Conference on, 2009, pp. 534–543.

[7] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, “Improving the Performance of Hypervisor-based Fault Tolerance,” in Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 2010, pp. 1 –10.

[8] “LVM2 resource page.” http://sourceware.org/lvm2/.

[9] D. J. Scales, M. Nelson, and G. Venkitachalam, “The design of a practical system for fault-tolerant virtual machines,” SIGOPS Oper. Syst. Rev., vol. 44, pp. 30–39, December 2010.

[10] W. Cui, D. Ma, T. Wo, and Q. Li, “Enhancing reliability for virtual machines via continual migration,” in ICPADS ’09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems. Washington, DC, USA: IEEE Computer Society, 2009, pp. 937–942.

[11] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, “Respec: Efficient online multiprocessor replayvia speculation and external determinism,” In Proc. Int. Conf. on Arch. Support for Prog. Lang. and Operating Systems (ASPLOS), 2010, pp. 77–90.

[12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live Migration of Virtual Machines,” in NSDI’05:Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[13] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the Sync. In Proceedings of the 2006 Symposium on Operating Systems Design and Implementation (Nov. 2006).

[14] Strom, R.E. and Bacon, D.F. and Yemini, S.A., “Volatile logging in n-fault-tolerant distributed systems,” in Fault-Tolerant Computing, Eighteenth International Symposium on, Jun 1988, pp. 44 –49.

[15] M. Nelson, B. H. Lim, and G. Hutchins, “Fast transparent migration for virtual machines,” in ATEC ’05: Proceedings of the annual conference on USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX Association, 2005, p. 25.

[16] X. Zhang, Z. Huo, J. Ma, and D. Meng, “Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration,” in Cluster Computing (CLUSTER), 2010 IEEE International Conference on, 2010, pp. 88 –96.

[17] "VMware Inc. Vmware fault tolerance (FT)," http://www.vmware.com/products/vsphere/, 2011.

[18] A. S.Tanenbaum, Modern Operating Systems. Prentice Hall, 2007.

[19] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, A. Wrfield, "RemusDB: Transparent High Availability for Database Systems," in 37th International Conference on Very Large Data Bases, 2011, pp. 738-748.

[20] Kai-Yuan Hou, Mustafa Uysal, Arif Merchant, Kang G. Shin and Sharad Singhal, "HydraVM: Low-Cost, Transparent High Availability for Virtual Machines," HP Laboratories, Tech. Rep. HPL-2011-24, 2011.

[21] M. H. Sun, D. M. Blough, “Fast, lightweight virtual machine checkpointing,” Georgia Institute of Technology, Tech. Rep. GIT-CERCS-10-05, 2010.

[22] Vishwanath, K.V., and Nagappan, N., “Characterizing cloud computing hardware reliability,” in Proc. Proceedings of the 1st ACM symposium on Cloud computing, Indianapolis, Indiana, USA, 2010, pp. 193-204.

[23] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, “Execution replay of multiprocessor virtual machines,” in Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ser. VEE ’08, 2008, pp. 121–130.

[24] L. A. Barroso and U. Hölzle, “The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines,” In Synthesis Lectures on Computer Architecture, 2009.