# Phantasy: Low-Latency Virtualization-Based Fault Tolerance via Asynchronous Prefetching

Shiru Ren, Yunqi Zhang, *Member, IEEE*, Lichen Pan, and Zhen Xiao, *Senior Member, IEEE*

**Abstract**—Fault tolerance has become increasingly critical for virtualized systems as growing amount of mission-critical applications are now deployed on virtual machines rather than directly on physical machines. However, prior hardware-based fault-tolerant systems require extensive modification to existing hardware, which makes them infeasible for industry practitioners. Although software-based techniques realize fault tolerance without any hardware modification, they suffer from significant latency overhead that is often orders of magnitude higher than acceptable. To realize practical low-latency fault tolerance in the virtualized environment, we first identify two bottlenecks in prior approaches, namely the overhead for tracking dirty pages in software and the long sequential dependency in checkpointing system states. To address these bottlenecks, we design a novel mechanism to asynchronously prefetch the dirty pages without disrupting the primary VM execution to shorten the sequential dependency. We then develop Phantasy, a system that leverages page-modification logging (PML) technology available on commodity processors to reduce the dirty page tracking overhead and asynchronously prefetches dirty pages through direct remote memory access via RDMA. Evaluated on 25 real-world applications, we demonstrate Phantasy can significantly reduce the performance overhead by 38 percent on average, and further reduce the latency by 85 percent compared to a state-of-the-art virtualization-based fault-tolerant system.

**Index Terms**—Fault tolerance, virtualization, checkpoint, recovery

## 1 Introduction

In recent years, deploying on virtual machines (VMs) rather than directly on physical machines has become more and more common for the increasingly wide range of application domains for the ease of use and high portability. While some of these application domains may have low availability requirements, others like financial services, database management systems, and network functions virtualization services are mission-critical and therefore demand extremely high availability and reliability. For instance, financial services and network functions virtualization services typically demand at least five nines availability (i.e., 99.999 percent) [1], which is orders of magnitude higher than what cloud providers like Amazon EC2 can promise in the face of hardware failures [2].

To close down this gap, recent work has sought to provide fault tolerance (FT) by seamless failover in the event of hardware failures for the virtualized environment using both hardware and software techniques. Hardware-based fault tolerance survives failures by duplicating hardware components to provide the necessary redundancy in case one should fail [3], [4]. However, such techniques require extensive modifications to the existing hardware or additional hardware components that are not available in commodity systems, which make them infeasible for industry practitioners.

In contrast, software-based techniques provide fault tolerance without any modification to existing hardware by periodically backing up the entire system states of the primary VM to a secondary VM hosted on a different physical machine, which will continue execution on behalf of the primary VM in the event of hardware failures on the primary VM [5], [6], [7], [8], [9], [10], [11], [12], [13]. Specifically, these systems often take incremental checkpoints of the system states including CPU, memory and other devices, and transmit these checkpoints to the secondary VM periodically (i.e., every epoch) to enable seamless failover. Because such systems can run on commodity hardware without any modification, they are considered much more feasible than hardware-based techniques.

However, these systems introduce significant latency and overhead (i.e., two to three orders of magnitude higher latency as we show later in the paper) because all the checkpoints are managed in software as opposed to hardware. This greatly restricts their application and prevents such systems from being widely deployed in production, as many mission-critical applications are real-time and inherently sensitive to latency. For example, stock exchange systems have stringent latency requirements to be able to react in time to the frequent trades in real-time, and high latency in online services like web search and social networks directly translates to poor user experience [14], [15]. Therefore, realizing efficient and low-latency virtualization-based fault-tolerant systems that can be deployed in production still largely remains an open research question.

- *S. Ren, L. Pan, and Z. Xiao are with the Department of Computer Science, Peking University, Beijing 100871, China. E-mail: {rsr, plc, xiaozhen}@net.pku.edu.cn.*
- *Y. Zhang is with the Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI 48109. E-mail: yunqi@umich.edu.*

*Manuscript received 22 Nov. 2017; revised 8 July 2018; accepted 3 Aug. 2018. Date of publication 16 Aug. 2018; date of current version 22 Jan. 2019. (Corresponding author: Zhen Xiao).*
*Recommended for acceptance by M. Caccamo.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TC.2018.2865943*

0018-9340 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

To answer this question, we first conduct an in-depth investigation to understand the bottlenecks of prior software-based techniques, in which we find the software-based dirty page tracking and long sequential dependency in checkpoint management being the largest contributors to the significant latency overhead. Specifically, to replicate the system states of the primary VM, all the dirty pages need to be recorded and transmitted to the secondary VM in the checkpoints. Given how fast modern processors operate and thereby the large amount memory accesses issued, translating to large numbers of dirty pages, keeping track of all the dirty pages in software is simply intractable because each new dirty page needs invoke an expensive page fault VM exit. In addition, all the output system states generated within an epoch (e.g., a response network packet to respond a query) cannot be sent out until the corresponding checkpoint has been sent successfully to the secondary VM after the end of the epoch, resulting in a long sequential dependency which consists of executing each epoch, generating and transmitting the checkpoint. This dependency is again magnified by the large number of dirty pages generated in each epoch as larger checkpoints need to be managed, which leads to significant degradation in response latency.

To overcome these bottlenecks, we design and develop Phantasy, a system that keeps track of dirty pages using page-modification logging (PML) [16] available in commodity Intel processors to reduce page tracking overhead, and asynchronously prefetches the dirty pages to shorten the sequential dependency of generating and transmitting checkpoints. Specifically, PML provides hardware-assisted monitoring of dirty pages generated during the VM execution, which significantly reduces the overhead compared to the software-based techniques that invoke expensive page fault VM exits. To shorten the sequential dependency, we design a fundamentally novel technique that speculatively prefetches the dirty pages identified by PML from the primary VM without interrupting its execution instead of waiting for the end of each epoch. Combined with the direct access to the remote memory provided by RDMA, the prefetch is designed to be driven entirely by the secondary VMM (Virtual Machine Manager) and completely transparent to the execution of the primary VM. Consequently, only the dirty pages that have not been prefetched need to be checkpointed at the end of the epoch, while the majority of the dirty pages have already been prefetched to the secondary VM.

We evaluate Phantasy on 25 real-world applications spanning both conventional memory-intensive and compute-intensive batch processing applications and mission-critical latency-sensitive applications that benefit significantly from fault tolerance. By asynchronously prefetching the dirty pages, our system reduces the number of dirty pages that need to be checkpointed each epoch by more than 50 percent and up to 84 percent compared to the state-of-the-art virtualization-based fault-tolerant system. Experimental results show that Phantasy significantly reduces the overhead of batch processing applications by 38 percent on average, and further improves the latency of latency-sensitive applications by more than 85 percent.

To the best our knowledge, this is the first paper that realizes a virtualization-based fault-tolerant system with low enough overhead that is practical to be deployed in production. Specifically, this paper makes the following contributions.

- We present an asynchronous prefetching mechanism that speculatively prefetches the dirty pages from the primary VM without disrupting its execution to shorten the sequential dependency of generating and transmitting VM checkpoints.
- We develop Phantasy leveraging PML technology available on commodity Intel processors and direct remote memory access offered by RDMA to reduce the overhead of dirty page tracking and prefetch the dirty pages to the secondary VM.
- We evaluate our system using 25 real-world applications spanning a wide range of application characteristics, and demonstrate the effectiveness of Phantasy in reducing performance overhead and query latency.

The rest of this paper is organized as follows. Section 2 gives a brief summary of prior techniques and illustrates their limitations. Section 3 presents design principles and the system architecture of Phantasy. Section 4 describes how Phantasy implements the asynchronous prefetching mechanism via RDMA and PML to overcome the limitations of prior work. Section 5 discusses several performance optimizations. We evaluate Phantasy in Section 6. Section 7 discusses related work and Section 8 concludes the paper.

## 2 Background

In this section, we first present a brief summary of prior work in virtualization-based fault-tolerant systems leveraging periodic checkpoints. We then discuss the limitations of prior works preventing these techniques from being widely adopted in the production environment.

### 2.1 Virtualization-Based Fault Tolerance

Prior work introduces the technique of periodically checkpointing the entire VM states to a secondary VM running on a different physical machine to provide fault tolerance in case of machine failures. Specifically, the primary VMM takes incremental checkpoints of the CPU, memory, and device states periodically at a fixed frequency, which is very similar to the pre-copy phase of live VM migration [17]. To ensure the secondary VM can transparently continue the execution in the event of machine failure, all the I/O events (i.e., network communication and disk operations) need to be held in a buffer before they can be committed after the corresponding VM checkpoint has been successfully transmitted to the secondary VM.

In such systems, the primary VM is typically paused every tens of milliseconds, which is often referred as an epoch, to checkpoint system states as illustrated in Fig. 1. During each epoch, all dirty pages are tracked while handling the page fault VM exits in the VMM. Meanwhile, all the output system states are buffered temporarily. At the end of each epoch, the primary VM execution will be paused, so that the primary VMM can generate an incremental checkpoint by invoking the pre-copy phase of live migration [17] to copy all the dirty pages and system states to a local staging area in the VMM. A new buffer is then

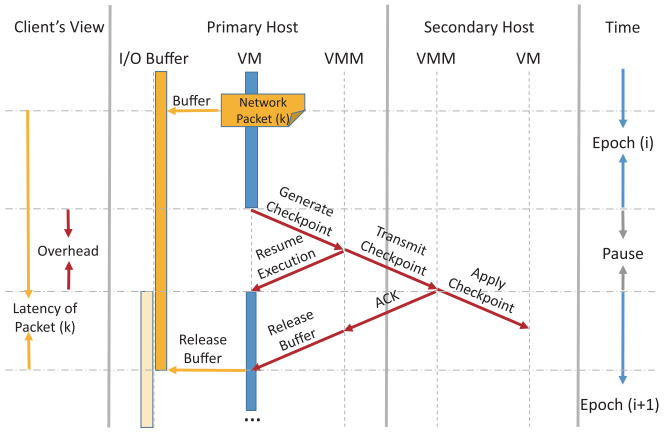


Fig. 1. The execution flow of the virtualization-based checkpoint-recovery fault tolerance.

inserted to keep track of all the output states of the next upcoming epoch. Once the new buffer has been inserted, the primary VM can resume execution while the VMM starting to transmit the checkpoint to the secondary VMM. The primary VM can resume execution immediately because the speculative states in the next epoch are not visible to the outside world. Upon receiving the acknowledgement of successfully transmitting the last checkpoint, the primary VMM can free up the state buffer used for the previous epoch.

In the event of hardware failure on the primary host, the primary VMM will stop heartbeating, which triggers a failover to the secondary VM instantly. The secondary VM starts execution on behalf of the primary VM from the most recent successfully transmitted checkpoint, so the transitioning can be transparent to the users. The uncommitted states on the primary VM that haven't been transmitted to the secondary VM and the buffered outgoing network packets will be lost, but it will only appear to be a temporary packet loss to the outside world as the secondary VM takes over the execution.

## 2.2 Limitations of Prior Work

While the simplicity of such design is appealing, it introduces significant latency and performance overhead, which greatly restricts its application.

### 2.2.1 Batch Processing Applications

In particular, there are primarily two sources of performance overhead as illustrated in Fig. 1 for batch applications (e.g., SPEC, PARSEC, and kernel-build) that are not latency-sensitive.

- The overhead of tracking all the dirty pages in the VMM.
- The overhead of generating checkpoints, where the primary VM execution needs to be paused.

Given how fast modern multi-core processors operate, applications often issue a huge amount of memory accesses even in an extremely short period of time, translating to a large number of dirty pages. Therefore, tracking all these dirty pages in the VMM incurs significant performance overhead in the software stack. In addition, the more dirty pages there are, the longer it takes to generate the checkpoints. In aggregate, prior work often incurs over 40 percent performance overhead running these batch applications [6].

### 2.2.2 Latency-Sensitive Applications

As opposed to these batch applications, many mission-critical applications are latency-sensitive (e.g., database management systems, network functions virtualization services, and data caching services) that heavily interact with I/O devices. Due to their criticality, these applications can benefit a great deal from fault tolerance. However, they experience even more degradation, typically in the form of latency increase, running on systems presented in prior work. This is because they suffer two more sources of degradation in addition to the two we discussed above as demonstrated in Fig. 1.

- The latency to finish executing an epoch, as all outgoing I/O events will be buffered within each epoch.
- The latency to transmit the generated checkpoint and receive the acknowledgement, because the buffer can only be released after receiving the acknowledgement.

These four sources of latency present an interesting tradeoff between the length of an epoch and the overhead of generating and transmitting the checkpoint. The longer each epoch is, the less overhead generating and transmitting the checkpoint contribute overall because they are invoked less often. However, longer epoch also translates to longer latency to finish executing each epoch, which also contributes to the overall latency increase. To amortize the overhead of generating and transmitting the checkpoint, prior work often sets the length of each epoch to at least tens of milliseconds. As a result, any mission-critical or latency-sensitive application with millisecond or sub-millisecond level latency suffers significant performance degradation.

Moreover, when the application yields a large amount of dirty pages such that the checkpoint cannot be generated and transmitted within the duration of the next epoch, the VMM will not receive the acknowledgement for the current checkpoint in time to start to generate the next checkpoint. This results in a prolonged next epoch, which further aggravates the overhead in both the latency to finish executing an epoch (i.e., the next epoch will be lengthened due to the delayed acknowledgement) and the latency to generate and transmit the next checkpoint (i.e., longer epoch often yields more dirty pages thereby larger checkpoint). Consequently, the application will experience increasing amount of latency degradation due to the queueing effect (i.e., overutilized queueing system) as the length of each epoch and the size of each checkpoint keep growing.

### 2.2.3 Summary

Based on the above analysis, we summarize the limitations of prior work as following.

- Software-Based Page Tracking: Keeping tracking of dirty pages in the software stack introduces significant overhead.
- Long Sequential Dependency: Large portion of the overhead and latency degradation can be attributed to the long series of events that can only be executed sequentially, namely executing each epoch, generating the checkpoint and transmitting the checkpoint.

Both of these limitations are further magnified by the large number of dirty pages, which increases the overhead of tracking dirty pages and prolongs the duration of the sequential events. In practice, systems like Remus [6] incur two to three orders of magnitude latency degradation, which greatly limits the practicality of such systems. Therefore, designing fault-tolerant virtualization-based systems without incurring significant latency and overhead still remains an open research question, and a novel approach is needed before such systems can be widely adopted in the production environment.

## 3 System Overview

Given the limitations of prior work, we first illustrate the design principles of a system that can overcome these limitations in providing virtualization-based fault tolerance. We then present the system architecture of Phantasy, a system we design and develop to realize virtualization-based fault tolerance without incurring significant overhead in latency.

### 3.1 Design Principles

As we discussed in Section 2, the overhead of tracking dirty pages and the long sequential dependency are the two limiting factors for prior work. To realize a virtualization-based fault-tolerant system that can be widely deployed in the production environment, we need to meet the following design goals:

- Lower the dirty page tracking overhead.
- Shorten the sequential dependency in checkpointing execution.

To lower the high overhead of tracking dirty pages in the software stack, we explore other efficient options and find the Page-Modification Logging technology available on the latest commodity Intel processors to be the perfect candidate. PML is an enhancement for the VMM to provide hardware-assisted monitoring of memory pages modified during the VM execution. Because the monitoring logic is implemented in the hardware, the overhead for tracking dirty pages is much lower than doing it in the software stack.

To shorten the sequential dependency in checkpoint VM execution, we investigate a fundamentally different approach by asynchronously prefetching dirty pages using a pulling model. Instead of waiting for all the dirty pages to checkpoint at the end of each epoch, we design a system to speculatively prefetch the dirty pages right after they have been modified by proactively pulling them to the secondary VM without interrupting the execution of the primary VM. Consequently, only the dirty pages that have not been properly prefetched by the secondary VMM need to be checkpointed, while the majority of the dirty pages have already been prefetched to the secondary VM as we will demonstrate later in the paper. This significantly reduces the number of dirty pages that need to be checkpointed and transmitted, which directly shortens the sequential portion of the checkpoint process. Moreover, the shortened latency of generating and transmitting checkpoints in turn allows the duration of each epoch to be shortened (i.e., no longer need to amortize the overhead of generating and transmitting the checkpoint as much), which further reduces the network latency for latency-sensitive applications as the time the network packets need to be buffered is reduced.

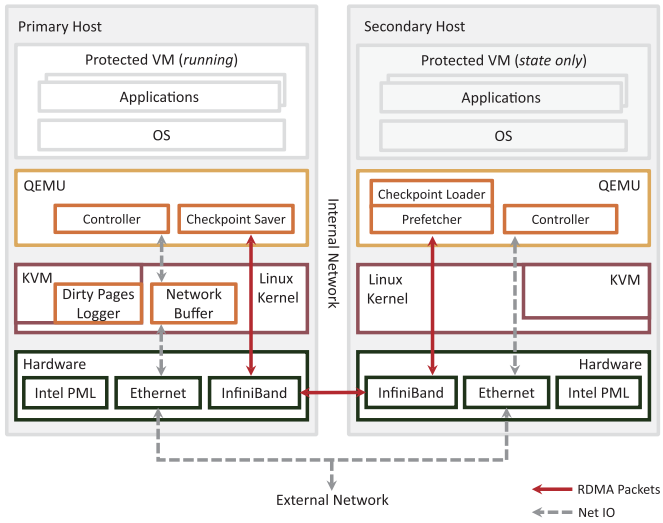


Fig. 2. Architecture overview.

### 3.2 System Architecture

To realize these design principles with strict correctness requirements, we design Phantasy as illustrated in Fig. 2. Phantasy currently focuses on providing FT in a LAN environment by seamless failover in the event of hardware fail-stop failures [7] (e.g., power supply, memory, processor, or PCIe link failure). The system is composed of five components as highlighted in orange boxes in Fig. 2: the Controller, the Checkpoint Saver/Loader, the Prefetcher, the Dirty Pages Logger and the Network Buffer. The *Controller* periodically sends keep-alive messages to the secondary VMM to detect system failures, and engages the recovery procedure by redirecting execution to the secondary VM when failures are detected. During each epoch of VM execution:

- The *Dirty Pages Logger* tracks the dirty pages leveraging the hardware-assisted PML technology at real-time.
- The *Prefetcher* running on the secondary VMM speculatively prefetches the dirty pages the *Dirty Pages Logger* has already recorded by asynchronously pulling them without interrupting the primary VM execution. The reason why these prefetches are "speculative" is because these pages might be modified again after the prefetch, in which case they will be prefetched again or transmitted in the checkpoint.
- The *Network Buffer* buffers all the network packets within the duration of each epoch.

At the end of each epoch:

- The *Checkpoint Saver* generates a checkpoint based on which pages remain "dirty" (i.e., has not been properly prefetched by the *Prefetcher*), which will then be transmitted to the secondary VM.
- The *Controller* resumes primary VM execution after the *Checkpoint Saver* finishes generating the checkpoint.
- Once the checkpoint has been successfully transmitted to the secondary VMM, the *Checkpoint Loader* will apply the received checkpoint to the secondary VM.

- The network packets stored in the *Network Buffer* will be released when the *Controller* receives the acknowledgement for successfully transmitting the previous checkpoint.

## 4 Asynchronous Dirty Page Prefetching

As described in Section 3, the key to overcome the limitations of prior work is the capability of asynchronously prefetching dirty pages in parallel with the primary VM execution to shorten the sequential dependency of constructing and transmitting the checkpoints. To realize this, we need a mechanism to asynchronously pull memory pages, as well as an algorithm to determine the "right" pages to prefetch (i.e., the dirty pages that no longer need to be checkpointed if prefetched). Therefore, we break this section down into the following two topics.

- The mechanism to allow the secondary VMM to pull memory pages asynchronously without interrupting the primary VM execution (Section 4.1).
- The mechanism to determine which memory pages to pull to maximize the prefetch precision (i.e., the ratio between pages no longer need to be checkpointed once prefetched and total number of prefetched pages), efficiently shortening the sequential dependency (Section 4.2).

We then put these mechanisms together and present the detailed workflow of Phantasy during normal execution (Section 4.3) and how it provides fault tolerance in the event of hardware failures (Section 4.4).

### 4.1 Asynchronous Memory Page Pulling

To shorten the sequential dependency of constructing and transmitting the checkpoints, we first need a mechanism to pull memory pages from the primary VM in an asynchronous fashion without interrupting its execution. In this section, we present the mechanism we use in Phantasy, and describe the specific prefetching protocol, which is designed to be generic and VM-agnostic.

To facilitate asynchronous memory page pulling, we leverage the direct remote memory access provided by RDMA, which completely bypasses the CPU and OS kernel of the host system and thereby can be done entirely in parallel with the primary VM execution. In addition, RDMA also offers zero-copy, low latency, and high throughput communication [18], [19], which makes it a great candidate as the underlying foundation of the asynchronous memory page pulling mechanism.

Specifically in Phantasy, the secondary VMM pulls memory pages from the primary VM by issuing one-sided READ verb offered by RDMA. As opposed to SEND and RECV verbs, which are two-sided, READ and WRITE verbs are one-sided, which allows the secondary VMM to drive the remote memory access entirely without any involvement of the primary VM. What this means is the primary VM can continue its execution uninterrupted while the secondary VMM pulls memory pages in parallel. Moreover, one-sided verbs typically provide lower communication latency and higher throughput compared to two-sided ones, as demonstrated by prior works in leveraging RDMA for VM migrations and replication [12], [20], [21].

### 4.2 Dirty Page Prediction

With the asynchronous memory page pulling mechanism, the secondary VMM can prefetch memory pages without interrupting the primary VM. However, it still needs to determine which memory pages to prefetch without consulting with the primary VM, because prefetching pages that are not "dirty" is not going to shorten the sequential dependency as the number of dirty pages that need to be checkpointed stays the same. In addition, when to prefetch is also critical for constructing an efficient prefetching mechanism, because prefetching dirty pages that will be written again within the same epoch cannot reduce the size of the checkpoints. This section discusses the mechanism we leverage in Phantasy to maximize the prefetching efficiency by precisely predicting the dirty pages.

Initially, we try to tackle this challenge the same way as cache prefetcher, which actively predicts memory pages are likely to be dirty based on past patterns. We experiment with several state-of-the-art memory access prediction models including multi-level dirty page caches and a Markov model-based machine learning algorithm [22]. However, we find the prediction accuracy is too low even with the best model (i.e., about 12 percent) to provide a non-negligible performance improvement.

We then look into the techniques proposed by prior work in obtaining page modification information, where the majority of the systems [6], [9] tracks dirty pages by write-protecting all memory pages. During each epoch, a VM exit is triggered to trap into the VMM to mark the page dirty whenever a memory write operation occurs, which is extremely time-consuming and introduces significant performance overhead. Recent work has also sought to leverage extended page table (EPT), an emerging technology available on Intel processors that contains dirty flags of the page table [23], [24], [25]. However, it is still required to traverse the entire EPT to gather all the leaf entries where the dirty flag is set, which is still too expensive for predicting dirty pages to prefetch.

To address this challenge, we then construct our dirty page prediction mechanism using the PML technology [16] on commodity Intel processors, which is a hardware-assisted enhancement to allow the VMM directly monitoring the modified memory pages during VM execution. When PML is enabled, each memory write will automatically generate an entry in a pre-allocated in-memory buffer, which contains the guest-physical address (GPA) of the write. The pre-allocated buffer is composed of 512 64-bit entries, where each entry references the GPA of a modified page.

Based on PML, Phantasy implements the Dirty Page Logger in KVM to obtain the page modification information by periodically checking the GPAs stored in the PML buffer. It then stores the information in a dirty page bitmap mapped between user and kernel space by `mmap()`. To avoid locking, a new bitmap is generated each time and indexed by a timestamp so both primary VMM and secondary VMM are able to identify which bitmap it is and reference all the bitmaps in the same order. All these bitmaps are stored in a specified area of primary host's memory which has already been mapped into user space and registered to enable remote read access. As a result, the secondary VMM can now directly read all the dirty page bitmaps entirely

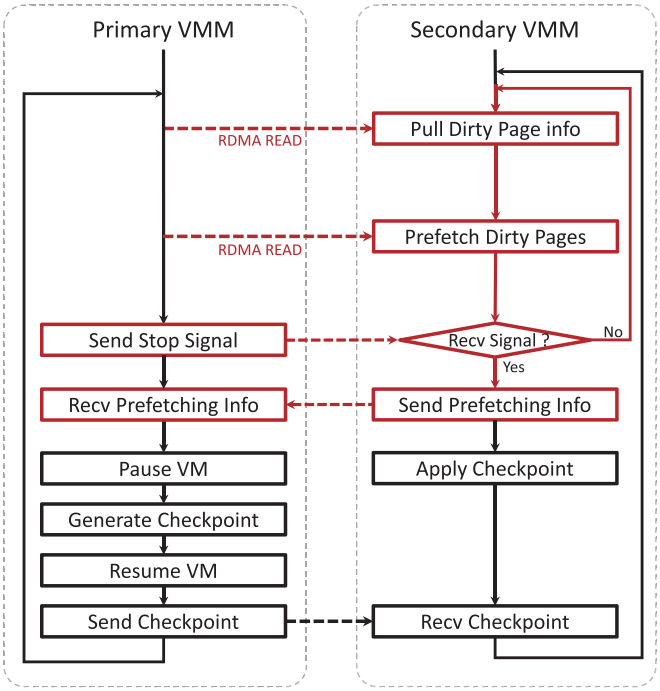


Fig. 3. The workflow of Phantasy for asynchronously prefetching dirty pages to shorten the sequential dependency of constructing and transmitting checkpoints.

asynchronously from the user space of the primary host via RDMA READ operations. With these bitmaps, Phantasy can simply try to prefetch all the memory pages that are already dirty instead of predicting which ones are likely to be dirty, which significantly improves the precision and efficiency of the prefetches. Note that the secondary VMM may not have enough time before the end of the epoch to prefetch all the dirty pages and some already prefetched pages may be written and become dirty again even with multi-round prefetching, so the remaining dirty pages still need to be transmitted in the checkpoints.

### 4.3 Putting It Together

With both the mechanisms for pulling memory pages asynchronously and predicting dirty pages, we are able to design and develop Phantasy, a system that asynchronously prefetches dirty pages without interrupting the primary VM to shorten the sequential dependency of constructing and transmitting the checkpoints. In this section, we present the detailed workflow of our system.

Fig. 3 illustrates the workflow of Phantasy, in which the novel steps we introduce in this paper are highlighted in red. Specifically, the primary and secondary VMMs go through the following steps.

1) The secondary VMM periodically checks if there are any new dirty pages generated on the primary VM by checking the dirty page bitmaps through RDMA READ operations.
2) If there are any new dirty pages, the secondary VMM will pull the dirty pages, whose GPAs can be obtained by taking the union of all the dirty page bitmaps, again through RDMA READ accesses.
3) Once all the dirty pages identified by the previous check on the dirty page bitmaps have already been prefetched, Phantasy repeats the same process from Step 1 to prefetch the newly written dirty pages. Note the secondary VMM might be able to finish multiple rounds of prefetches within the duration of each epoch.
4) At the end of each epoch, the primary VMM will first send a stop signal to the secondary VMM to stop prefetching and also pause its own execution.
5) Once the secondary VMM receives the stop signal, it stops prefetching and sends the primary VMM the information about which pages have already been prefetched. This is needed because the last prefetch might not have been completed by the time the secondary VMM receives the signal, so the remaining pages still need to be transmitted in the checkpoint.
6) The primary VMM generates a new checkpoint based on the received prefetching information and transmits it to the secondary VMM.
7) Once the new checkpoint has been sent out to the secondary VMM, the primary VM can continue execution, and repeats from Step 1.
8) Once the secondary VMM has successfully received the latest checkpoint, it will send back an acknowledgement to the primary VMM.
9) After the primary VMM receives the acknowledgement for successfully transmitting the checkpoint, it will release the buffered outgoing packets.

### 4.4 Failure Recovery

Phantasy periodically sends keep-alive messages from the primary VMM to the secondary VMM, and detects failures when the secondary VMM does not receive five successive keep-alive messages. Once the failure has been detected, the secondary VM immediately takes over the execution using the contexts of the most recent checkpointed state.

To resume execution from exactly the most recent checkpoint, the secondary VMM does not prefetch the dirty pages directly to the VM memory at real-time, otherwise, the secondary VM might end up being states that are slightly ahead of the last checkpoint since some dirty pages have already been prefetched. Instead, a pre-allocated buffer is used to store these dirty pages temporarily before they can be applied when the next checkpoint has been successfully received.

## 5 Optimizations

On top of the asynchronous prefetching dirty page prefetching methodology, we also implement three optimizations to further improve the system performance. In this section, we first present the double buffering and undo logging we leverage to reduce the overhead of applying dirty pages on the secondary VM (Section 5.1). We then describe our priority-based prefetching algorithm that prioritizes the "write-cold pages" over "write-hot pages" to improve the prefetching efficiency especially for memory-intensive applications (Section 5.2). Last but not the least, we compress the work completion messages of RDMA READ verbs to further reduce the prefetching latency (Section 5.3).

### 5.1 Double Buffering and Undo Logging

As discussed in Section 4.4, the prefetched dirty pages cannot be directly applied to the secondary VM's memory in

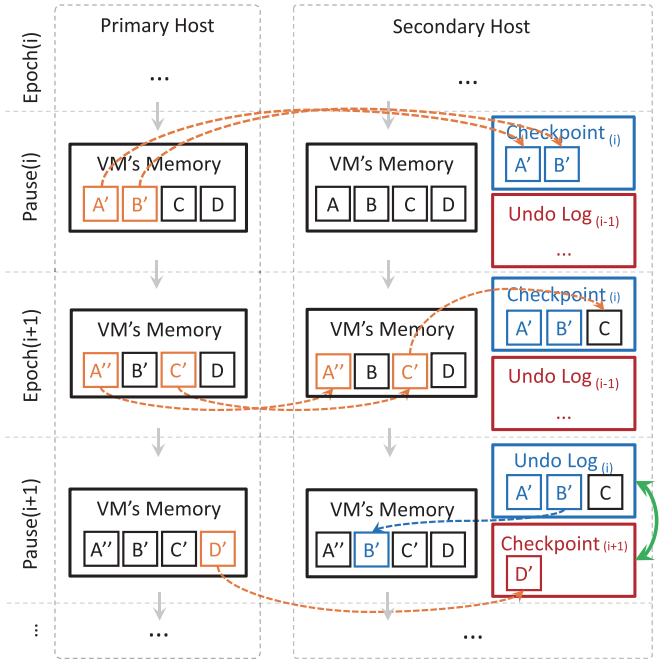


Fig. 4. Double buffering and undo logging in Phantasy.

real-time, otherwise we will not be able to start execution exactly at the last checkpoint in the event of hardware failure. However, naively managing the buffer can make applying checkpoints quite costly in terms of performance, because all the dirty pages need to be copied at least twice (i.e., once to the buffer and once to the secondary VM memory). To address this issue, we leverage double buffering to swap two buffers (i.e., shown as the blue and red boxes on the right-hand side in the figure) between undo log and checkpoint as illustrated in Fig. 4.

Phantasy manages two buffers on the secondary host: one is used to help store the latest checkpoint, and the other buffers the most recent undo state that can be restored in the event of failure. During each epoch, the dirty pages prefetched by the secondary VMM will be stored directly into the VM memory. If the corresponding page is already buffered in the checkpoint buffer, nothing else needs to be done since we can restore its previous state using the checkpoint buffer. Otherwise, the current state of the page needs to be written to the checkpoint buffer. After each epoch, the two buffer will be swapped (i.e., the previous checkpoint buffer will become the new undo log buffer and vice versa). The new dirty pages transmitted in the checkpoint will be stored in the new checkpoint buffer, and the undo log will contain the states of all the necessary pages to resume execution at the state right after applying the previous checkpoint. For instance, Fig. 4 demonstrates an example of this procedure.

- During the pause($i$) after the end of epoch($i$), the checkpoint containing dirty pages (denoted in orange boxes) $A'$ and $B'$ is stored in the blue buffer, which is serving as the checkpoint buffer at this point.
- During the next epoch($i+1$), the secondary VMM asynchronously prefetches the new dirty pages $A''$ and $C'$. $A''$ can be directly written to memory because its previous state $A'$ is already in checkpoint ($i$) stored in the blue buffer. However, the current state of page $C$ will need to be written to checkpoint ($i$) before we can apply $C'$ to the memory since it has not been buffered previously.
- At the next pause($i+1$) after epoch($i+1$), the red buffer and blue buffer will be swapped, where the blue buffer becomes the new undo log($i$) and the red buffer becomes the new checkpoint($i+1$). Now the undo log($i$) in the blue buffer contains the states of the relevant pages after applying checkpoint($i$), which can be restored if a failure occurs in pause ($i+1$). Meanwhile, the new dirty page $D'$ transmitted in the checkpoint will be stored in the new checkpoint buffer checkpoint($i+1$) first, and the checkpoint($i+1$) will be applied to the secondary VM's memory in pause($i+2$).

### 5.2 Priority-Based Prefetching

During each epoch, Phantasy tries to prefetch as many dirty pages as possible whenever the primary VM marks them in the dirty page bitmaps. However, it is not always the case that Phantasy will have enough time to prefetch all the dirty pages, especially for write-heavy memory-intensive applications that generate large quantities of dirty pages. When not all the dirty pages can be prefetched in time, some of the dirty pages might never get pulled by the secondary while other frequently over-written pages might have already been prefetched multiple times within the same epoch even though only the last prefetch is actually useful. This can result in inefficient prefetches as we waste a large amount of resource fetching the same pages over and over again and leave the rest of the pages untouched.

To quantify this observation, we measure the modification frequency of each memory page within each epoch, which is the number of rounds the same page is marked as dirty per epoch. We find that at least 21.51 percent pages are marked as dirty throughout the entire epoch, while 44.47 percent pages are only marked once per epoch. We refer these frequently marked dirty pages as "write-hot pages" and the others "write-cold pages". Prefetching the "write-hot pages" multiple times within the same epoch wastes a large amount of resource, which we could and should have used to prefetch those "write-cold pages". Based on this insight, we develop a priority-based prefetching algorithm that prioritizes "write-cold pages" over "write-hot pages" to further improve the prefetching efficiency.

Specifically, Phantasy prioritizes the pages which are the least recently modified. This optimization requires keeping track of when pages were modified. Therefore, Phantasy maintains statistics about the memory page's last-modified timestamp on the secondary VMM. In this revised design, the secondary VMM performs each round of prefetching at a fixed frequency. At the beginning of each round of prefetching, the secondary VMM sorts the dirty pages in ascending order according to the last-modified timestamp if there is not enough time to prefetch all the dirty pages based on estimation, and prefetches dirty pages in the same order to prioritize the pages that are the least recently modified (i.e., "write-cold pages"). If the secondary VMM estimates all the dirty pages can be prefetched in time, it skips the sorting to save resources. The time interval of each round of

TABLE 1
Specification of the Experimental Platform

| | Specification |
|---|---|
| Server | Dell OptiPlex 7470 Workstation |
| Processor | Intel Core i5-6500 @ 3.2 GHz (4-core, 6 MB LLC) |
| DRAM | 8 GB @ 2133 MHz |
| InfiniBand NIC | Mellanox ConnectX 40 Gbps InfiniBand (via PCIe 3.0 x8) |
| InfiniBand Switch | Mellanox SX6005 56 Gbps InfiniBand Switch |
| Ethernet NIC | Intel I219-LM 1 Gbps NIC |
| Ethernet Switch | TP-Link 8-Port Gigabit Ethernet Switch |
| Kernel Version | 4.4.62 |
| QEMU Version | 2.3.50 |

TABLE 2
Benchmarks Used in the Evaluation

| Category | Benchmarks |
|---|---|
| Compute-intensive batch | blackscholes, bodytrack, canneal, ferret, fluidanimate, freqmine, streamcluster, swaptions, vips, x264 from PARSEC [28] barnes, cholesky, fft, fmm, radix, volrend from SPLASH-2 [29] |
| Memory-intensive batch | kernel-build, pbzip, pfscan |
| Latency-sensitive | TPC-C, Twitter, Voter, SmallBank, TATP, YCSB from OLTP-Bench [30] |

prefetching is set to 1 ms by default and can be configured to any value through the control interface. Phantasy performs each round of prefetching at a fixed frequency since the impact of prefetching frequency on performance is negligible. Particularly, if there is time left after prefetching all dirty pages of a round, the secondary VMM will continue the unfinished prefetching of previous rounds.

### 5.3 Using RDMA READ with Unsignaled Completion

By default, each work request (WR) generates a work completion (WC) signal to notify completion when using RDMA verbs. However, handling large numbers of WCs can be resource consuming and introduce additional latency. Specifically, polling and checking the WCs from the completion queue consumes additional CPU cycles, and generating WCs also consumes precious resource on RDMA devices.

To reduce this overhead, Phantasy issues RDMA READ verbs with unsignaled completion when performing prefetching. With the unsignaled completion mechanism, the READ operations will no longer generate WCs for completion [26]. Meanwhile, transmission failures can still be detected as any error will yield a WC to provide the error message [27].

The unsignaled completion mechanism does not affect the correctness of our prefetching protocol because the prefetching operations within a round update different memory addresses without overlapping, and the memory regions that contain the prefetched pages will not be reused or destroyed. Particularly, to control the speed of prefetching and to avoid depleting completion queue resources, we generate a WC and wait for its acknowledgement after every fifty RDMA READs.

## 6 Evaluation

In this section, we first describe our evaluation methodology (Section 6.1) and profile Phantasy running in action (Section 6.2). We then evaluate the effectiveness of Phantasy in providing virtualization-based fault tolerance by comparing against prior work in the following aspects.

- Reducing performance overhead for batch processing applications (Section 6.3).
- Reducing latency for latency-sensitive applications (Section 6.4).

### 6.1 Experimental Setup

All the experiments in this section are conducted on the experimental platform described in Table. 1. The VMs are configured with 2 virtual CPUs and 2 GB memory running Ubuntu 14.04 with a kernel version at 3.13.0. We run one VM on the primary host to minimize the impact of running multiple VMs on the same host. However, Phantasy can protect the specified VM running concurrently with other VMs, as it can distinguish the dirty pages generated by the protected VM from the dirty pages generated by other VMs.

To cover a wide range of applications in our evaluation, we use 25 benchmarks from PARSEC [28], SPLASH-2 [29], and OLTP-Bench [30] representing three categories of applications, namely compute-intensive batch applications, memory-intensive batch applications, and latency-sensitive applications, as listed in Table 2. Specifically, we use simlarge or native input for all benchmarks from PARSEC [28] and SPLASH-2 [29]. To represent memory-intensive batch applications, we use kernel-build that compiles Linux kernel 3.13.1 with the default configuration, pbzip2 that compresses 111 MB of Linux source code, and pfscan that searches the word "error" in Linux source code. For latency-sensitive applications, we use six benchmarks from OLTP-Bench [30], because they are sensitive to latency degradation and can also benefit significantly from fault tolerance due to the criticality of online transaction processing.

For all experiments in this section, we compare our system against Micro-Checkpointing (MC) [31], [32], which is the state-of-the-art virtualization-based fault-tolerant system implemented on QEMU based on Remus [6]. We chose MC for comparison because, first, MC is designed and optimized for RDMA-based systems [31], therefore, it has shown relatively good performance in an RDMA environment; second, MC is implemented based on the same platform as Phantasy, which facilitates an accurate and unbiased direct comparison in terms of performance. In addition, we also present the overhead compared to the native VM execution without any fault tolerance capability.

### 6.2 Phantasy in Action

To understand the implication of the asynchronous prefetching mechanism we develop, we first characterize the RDMA bandwidth usage of Phantasy and MC over time. As shown in Fig. 5, we profile the bandwidth usage of Phantasy and MC for a 900 ms segment of execution of x264 from PARSEC [28], where the epoch size is configured at 100 ms.

MC incurs a bandwidth spike every 100 ms because all the dirty pages generated in the previous epoch can only be checkpointed and transmitted to the secondary VM sequentially at the end of the epoch. On the other hand, the bandwidth usage of Phantasy stays relatively constant. During

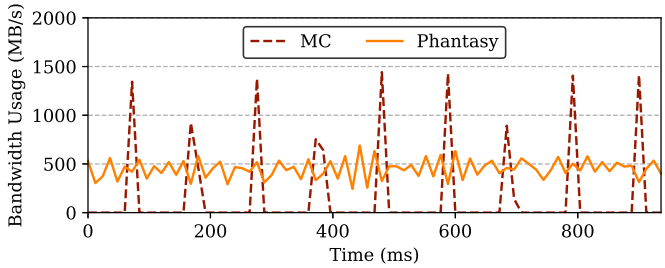


Fig. 5. The RDMA bandwidth usage of Phantasy and MC as a function of time for a 900 ms segment of execution of x264 from PARSEC [28] at 100 ms epoch size (with a sampling interval of 12 ms).

each epoch, the dirty pages are asynchronously prefetched by the secondary VM, resulting in a bandwidth usage at about 500 MB/s. At the end of each epoch, because the majority of the dirty pages have already been prefetched, the number of dirty pages that need to be checkpointed and transmitted is much lower, drastically reducing the probability of incurring large spikes in the bandwidth usage like MC. More importantly, it significantly reduces the length of the sequential dependency for generating, transmitting, receiving acknowledgement for the checkpoints and releasing the buffered network packets, which can directly reduce the performance overhead as well as the query latency for latency-sensitive applications.

### 6.3 Overhead for Batch Processing Applications

In this section, we first evaluate the end-to-end overhead of Phantasy compared to MC (Section 6.3.1). To further understand the source of the overhead reduction, we analyze the reduction in VM exits (Section 6.3.2) and in dirty pages (Section 6.3.3).

#### 6.3.1 Reduction in End-to-End Overhead

To evaluate the end-to-end overhead introduced by MC and Phantasy compared to the baseline native VM execution, we measure the execution time of the compute-intensive and memory-intensive batch applications in Table 2 for 50 times for all three configurations. We also vary the size of each epoch for MC and Phantasy to measure its impact.

The results are shown in Fig. 6, in which the *y*-axis denotes the overhead of MC and Phantasy at different epoch sizes compared to the native VM execution (i.e., higher bars represent more overhead). As shown in the figure, Phantasy significantly reduces the end-to-end overhead compared to MC, which is the state-of-the-art prior work. Across all benchmarks and three different epoch sizes, Phantasy can reduce the overhead drastically by 38.88 percent on average.

We also observe that Phantasy achieves larger improvement on I/O intensive applications (i.e., kernel-build, pbzip, pfscan) than CPU intensive applications. On average, Phantasy incurs 35.24 percent less overhead than MC for CPU intensive applications, and 58.33 percent less overhead for I/O intensive applications. This is because I/O intensive applications tend to have much more frequent memory writes, translating to more dirty pages, thereby more VM exits. By asynchronously prefetching these dirty pages, our system can significantly reduce the number of dirty pages need to be transferred (i.e., checkpoint size) during the sequential execution. Instead of tracking the dirty pages in the software-stack, which causes frequent VM exits, Phantasy leverages PML to record the dirty pages in hardware, further reducing the overhead caused by VM exits.

In addition to the amount of memory writes, the locality of memory writes also has an impact on the overhead reduction Phantasy achieves. If the application often writes to different memory pages (i.e., cold pages, high reuse distance), our prefetching mechanism will be more effective because the already prefetched pages are less likely to be written again. For example, although cholesky does not have a lot of memory writes, it does not write to the same pages very often, resulting in a relatively large overhead reduction (i.e., 50.37 percent at 5 ms epoch size).

Moreover, the page size is another key factor that can influence the performance of Phantasy since using smaller page size can reduce the overhead of transmitting dirty pages. A further experiment shows that using huge pages [33] (2 MB page size) increases the performance overhead by 9.55 percent on average compared to using 4 KB page size. However, we should also note that utilizing smaller pages may introduce additional overhead since higher amount of memory address translations will be required [34]. Therefore, we choose 4 KB as the default page size in Phantasy.

In summary, Phantasy significantly reduces the overhead of virtualization-based fault tolerance by lowering the dirty page tracking overhead and shortening the sequential dependency in checkpointing execution. To further investigate the overhead reduction achieved by Phantasy, we analyze the reduction in VM exits (Section 6.3.2) and in dirty pages (Section 6.3.3).

#### 6.3.2 Reduction in VM Exits

Fig. 7 presents the reduction in the amount of VM exits. As we can see in the figure, Phantasy significantly reduces the

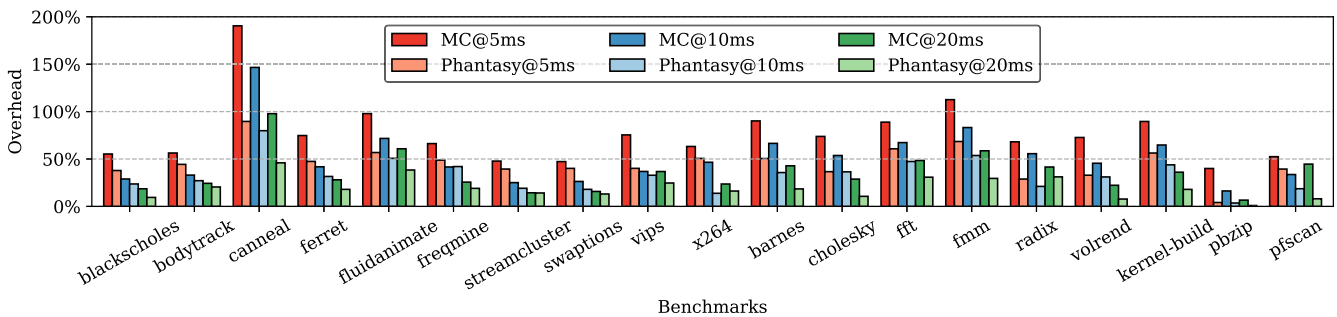


Fig. 6. The overhead of MC and Phantasy at different epoch sizes compared to the native VM execution (i.e., higher bars represent more overhead).

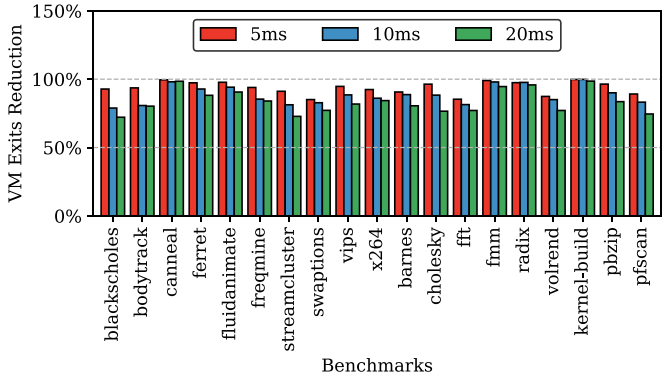


Fig. 7. The reduction in the number of VM exits achieved by Phantasy compared to MC.

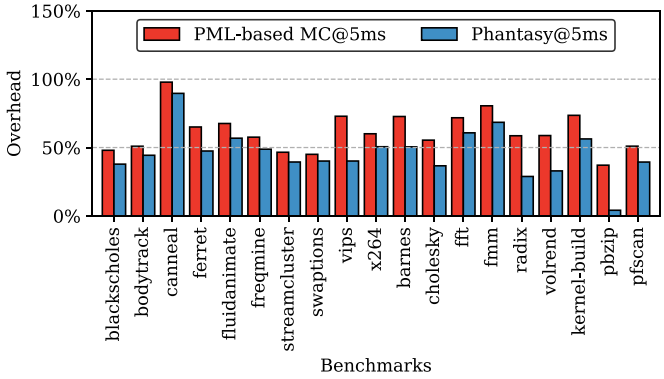


Fig. 9. The end-to-end overhead of Phantasy compared to PML-based MC.

VM exits by 88.40 percent on average. The reduction in VM exits decreases as the size of each epoch increases as illustrated in the figure. The reason is MC invokes only one VM exit for multiple writes to the same dirty page within the same epoch, which means the rate of VM exits generated drops as the epoch size grows. At 5 ms epoch size, Phantasy can reduce the VM exit by 93.94 percent. As we increase the epoch size to 10 ms and 20 ms, the reduction decreases to 88.41 percent and 82.86 percent, respectively.

If we compare the absolute numbers of VM exits, we can find that Phantasy reduces more VM exits for I/O intensive applications than CPU intensive applications, which explains why they tend to benefit a higher end-to-end overhead reduction. Because I/O intensive applications tend to generate more memory writes and systems like MC have to invoke one VM exit for each dirty page to track them, they naturally experience higher VM exits reduction.

### 6.3.3 Reduction in Dirty Pages in Checkpoints

In addition to VM exits, we find Phantasy can significantly reduce the number of dirty pages need to be transmitted to the secondary VM in checkpoints as demonstrated in Fig. 8. By asynchronously prefetching the dirty pages identified by PML, our system will have already transmitted a large fraction of the dirty pages at the end of each epoch, which means only the remaining dirty pages need to be transmitted in the next checkpoint. At 5 ms epoch size, Phantasy reduces the number of dirty pages in checkpoints by 51.77 percent for CPU intensive applications, and 52.82 percent for I/O intensive applications. If we compare the reductions among different epoch sizes, we can find the reduction increases as the epoch size grows, because the likelihood that our system can successfully prefetch each dirty page increases as epochs become longer, resulting in less dirty pages needed to be transmitted in the checkpoint. At 10 ms epoch size, Phantasy can reduce the dirty pages by 53.14 percent for CPU intensive applications, and 62.31 percent for I/O intensive applications. As we increase the epoch size to 20 ms, the reduction grows to 58.95 and 69.41 percent for CPU intensive applications and I/O intensive applications respectively.

### 6.3.4 Benefits of Innovations and Optimizations

To show the performance improvement contributed by PML and the asynchronous prefetching mechanism separately, we evaluate the end-to-end overhead of Phantasy compared to the PML-based MC which utilizes PML instead of software-based approach for dirty pages tracking. Fig. 9 illustrates that compared with PML-based MC, Phantasy can still reduce the overhead drastically by 26.98 percent on average. As a comparison, Phantasy incurs 38.88 percent less overhead than the original MC, which means the performance improvement contributed by our asynchronous prefetching is much greater than PML.

Fig. 10 quantifies the performance improvement due to each of Phantasy's optimizations. The results show that priority-based prefetching reduces the overhead by 3.6 percent on average, while unsignaled completion and double buffering contribute 1.2 and 1.35 percent improvement, respectively. The impact of unsignaled completion is highly

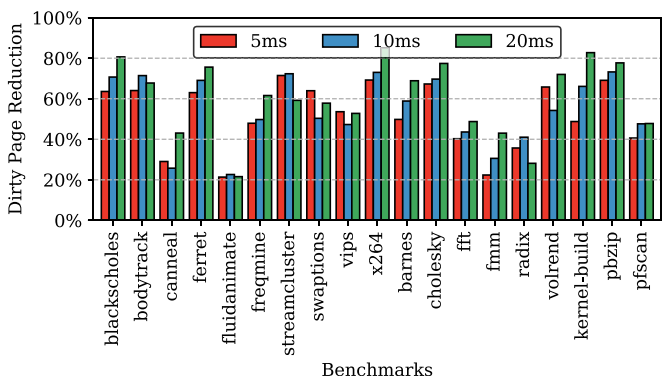


Fig. 8. The reduction in the number of dirty pages in checkpoints achieved by Phantasy compared to MC.

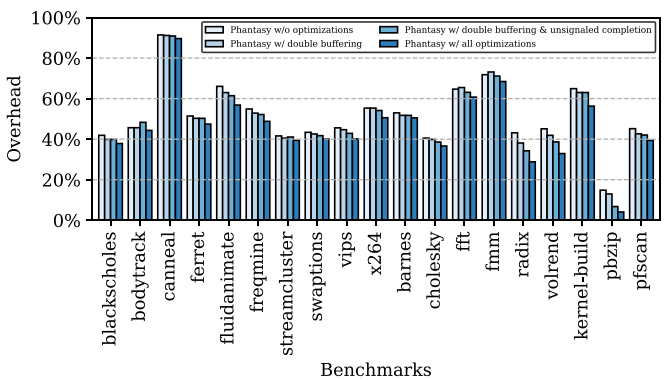


Fig. 10. The end-to-end overhead of Phantasy with different optimizations (at 5 ms epoch size).

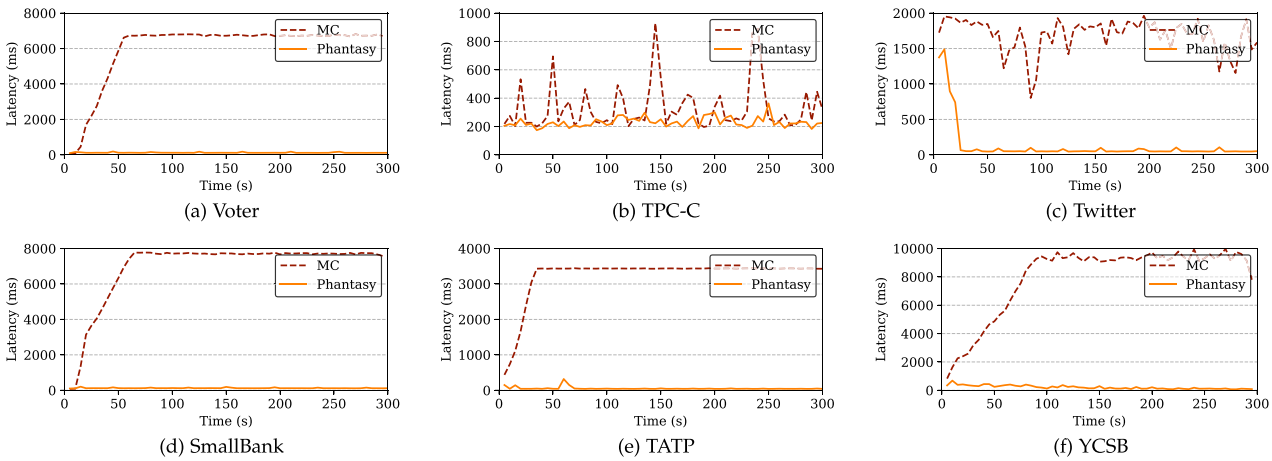


Fig. 11. Comparison of Phantasy and MC in query latency for six latency-sensitive applications from OLTP-Bench [30] (at 5 ms epoch size).

dependent on the amount of the dirty pages that the secondary VMM can prefetch, which explains why applications like bodytrack and streamcluster, which generate less dirty pages, benefit less from this unsignaled completion.

### 6.4 Latency for Latency-Sensitive Applications

Fault tolerance is critical for many latency-sensitive applications, especially ones that are mission-critical (e.g., database management systems, network functions virtualization services, and data caching services). However, the latency degradation introduced by the state-of-the-art virtualization-based fault-tolerant systems is so high as we illustrated in Section 2 that it is impractical to deploy such systems.

In this section, we evaluate the feasibility of deploying Phantasy for latency-sensitive applications by measuring the latency degradation and comparing to MC (Section 6.4.1). We then analyze in-depth the reduction achieved by Phantasy in the number of VM exits (Section 6.4.2) and in the number of dirty pages in checkpoints (Section 6.4.3).

#### 6.4.1 Reduction in Query Latency

To measure the query latency, we measure six latency-sensitive applications from OLTP-Bench [30] running on MySQL database, which is configured to run on MC and Phantasy for comparison. The measured query latency is shown in Fig. 11, the $x$-axis represents the timeline (i.e., each experiment runs for 300 seconds) and the $y$-axis shows the latency reported by OLTP-Bench [30].

On average, the Phantasy improves the query latency by 85.85 percent compared to MC. Specifically, MC is not able to sustain the queries for four out of the six applications (i.e., Voter in Fig. 11a, SmallBank in Fig. 11d, TATP in Fig. 11e, YCSB in Fig. 11f), as demonstrated in the initially increasing and quickly plateaued latency time series (i.e., queries start to time out as the system is overutilized). Although MC is able to sustain the queries for the other two applications (i.e., TPC-C in Fig. 11b and Twitter in Fig. 11c), the latency is quite bursty. For instance, the query latency for TPC-C increases from 250 ms all the way up to 900 ms (i.e., 3.6× degradation) around 150s in Fig. 11b. This is because of the aggravating queueing effect we discussed in Section 2, where the length of each epoch keeps growing when the checkpointing size increases and longer epoch in turn results in larger checkpoints.

On the contrary, Phantasy is able to sustain the incoming queries for all six applications at a much lower query latency (i.e., 85.85 percent reduction). Even for the worst case scenario (TPC-C in Fig. 11b), Phantasy can reduce the average latency by 30.84 percent compared to MC. In addition to the reduction in query latency, we can also observe that Phantasy can significantly reduce the variance of query latency, which is critical for latency-sensitive applications [14].

In summary, Phantasy realizes virtualization-based fault tolerance at a much lower latency, particularly 85.85 percent reduction compared to MC, which makes such systems practical for latency-sensitive applications. We then further analyze the reduction our system can achieve in the number of VM exits (Section 6.4.2) and dirty pages per checkpoint (Section 6.4.3).

#### 6.4.2 Reduction in VM Exits

Fig. 12 presents the number of VM exits per second Phantasy is able to reduce compared to MC. The reduction for TPC-C is lower because it is much more compute intensive than the other five applications, so the total number of VM exits it generates is lower than the other applications. The results show that tracking dirty pages using PML in hardware to reduce the number of VM exits has a direct positive impact on the query latency for latency-sensitive applications.

#### 6.4.3 Reduction in Dirty Pages

We then further characterize the reduction of dirty pages for these six latency-sensitive applications as shown in Fig. 13. In each figure, the $y$-axis on the left and the orange line

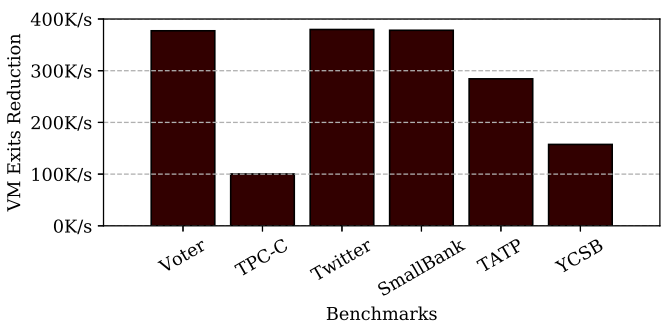


Fig. 12. The reduction in the number of VM exits per second achieved by Phantasy compared to MC.

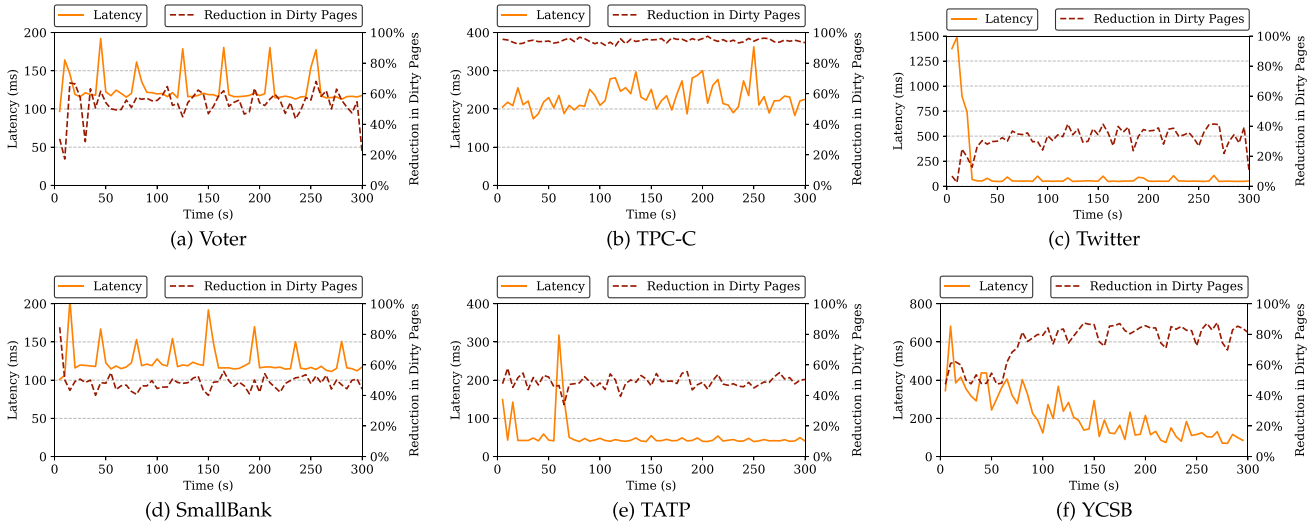


Fig. 13. The time series of the query latency and the reduction in dirty pages of six latency-sensitive applications from OLTP-Bench [30] running on Phantasy.

shows the query latency over time, and the *y*-axis on the right and the dashed red line represents the percentage of dirty page reduction of Phantasy comparting to MC. For example, SmallBank shown in Fig. 13d experiences a latency around 120 ms (i.e., orange line) with a few spikes and a reduction of around 50 percent (i.e., dashed red line) in dirty pages running on Phantasy. In aggregate, Phantasy is able to reduce the dirty pages that need to be transmitted in the checkpoints by 55.16 percent. Note that MC cannot sustain the queries and is timing out these queries for four out of the six applications, namely Voter in Fig. 13a, SmallBank in Fig. 13d, TATP in Fig. 13e and YCSB in Fig. 13f.

In addition, we also notice that the query latency and the dirty page reduction are inversely correlated, where drops in dirty page reduction often result in latency spikes. For example, the latency spike at 65s for TATP shown in Fig. 13e is inversely correlated with the drop in dirty page reduction, and the decreasing latency during the first 25s for Twitter shown in Fig. 13c is inversely correlated with the increasing trend of dirty page reduction. This inverse correlation further confirms our observation that asynchronously prefetching dirty pages to reduce the number of dirty pages that need to be transmitted in each checkpoint can directly reduce the query latency, making such virtualization-based fault-tolerant systems feasible for latency-sensitive applications.

## 7 Related Works

Researchers have proposed systems to provide extremely high availability by periodically checkpointing execution of the primary machine to a secondary replicated machine, so that the secondary machine can continue execution transparently in the event of machine failures on the primary machine. Bressoud and Schneider [5] first present and formalize the principles and protocols to implement software-only virtual machine-based fault tolerance systems. Friedman, Kama [35] and Napper et al. [36] present an implementation of such fault-tolerant systems on top of Java virtual machine. Remus [6] is one of the first systems that makes this mechanism practical by allowing speculative execution and asynchronous checkpointing and replication. Lu and Chiueh [10] also implement a speculative state transfer mechanism. Compared to their approach, Phantasy lowers the dirty page tracking overhead by leveraging the PML and shortens the sequential dependency in checkpointing execution by investigating a fundamentally different approach by asynchronously prefetching dirty pages using a pulling model. Zhu et al. [9], [11] present the idea of read-fault reduction and write-fault prediction to reduce the overhead of logging dirty pages, and introduce the concept of software-superpage to optimize the memory transfer between virtual machines. Tsao et al. [37] implement an efficient fault-tolerant system by using SSE instructions to achieve fine-grained dirty region tracking. VMware vSphere FT [38] is a commercial enterprise-grade system for providing continuous availability for applications by periodically taking incremental checkpoints of the VM states. RemusDB [39] exercises the idea of building high availability database management systems using virtual machine checkpointing, and presents optimizations catered for characteristics of such applications (i.e., memory intensive and sensitive to network latency). Tardigrade [13] addresses the same challenge by encapsulating execution into lightweight virtual machines, thereby avoiding having to synchronize unnecessary data between the primary and secondary machines like OS background tasks. Moreover, recent work [7] have also looked at synchronizing the primary VM and secondary VM using lazy checkpoints, which are only generated and applied to the secondary VM if its output diverges from the primary VM. Our work differs from such technique fundamentally by providing a different asynchronously communication channel for transmitting dirty pages, which can be applied to systems like Remus [6], Kemari [8], and COLO [7] to further complement their performance.

Moreover, we believe that many other works can potentially benefit from the idea of tracking dirty pages by leveraging PML and proactively pulling dirty pages through direct remote memory access via RDMA. For example, live VM migration faces the exact same technical challenge that how to efficiently track and transmit all the dirty pages. In order to achieve live VM migration, all the runtime states must be transferred from the source to the

destination without disconnecting the client or application [17]. Pre-copy, as a major approach to perform live VM migration, uses a similar high-level strategy as Phantasy. It first transfers all the dirty pages from source to destination in an asynchronous fashion while the VM is still running on the source. Then, if some pages change, they will be retransferred. Finally, it stops the source VM and transfers the remaining dirty pages. This idea was first proposed by Clark et al. [17]. VMotion [40] is one of the first systems that can migrate unmodified applications on the unmodified x86-based OS. To further improve the performance, some recent works demonstrated the benefit of using InfiniBand for VM migration [20], [41]. Moreover, to further expand the utility of live VM migration, some recent works focus on supporting the transparent, live wide-area migration of virtual machines [42], [43], [44].

## 8 Conclusion

In this paper, we have made the first attempt to leverage emerging processor (PML) and network (RDMA) features to achieve an efficient and low-latency fault tolerance. To realize a virtualization-based fault tolerance system that can be widely deployed in production environment, we first lower the dirty page tracking overhead by leveraging the PML. Then, we shorten the sequential dependency in checkpointing execution by investigating a fundamentally different approach by asynchronously prefetching dirty pages using a pulling model. Instead of waiting for all the dirty pages to checkpoint at the end of each epoch, with the help of RDMA, we design an asynchronous pull-based prefetching strategy to speculatively prefetch the dirty pages that recorded by PML by proactively pulling them to the secondary VM without interrupting the execution of the primary VM. By doing so, we can overlap dirty pages transport with VM execution, and therefore can potentially mask the vast majority or even all of the memory state synchronization overhead. We also discuss three more optimizations to further improve the system performance. By evaluating our system on 25 real-world applications, we demonstrate that Phantasy can significantly reduce the performance overhead by 38 percent on average, and further reduce the latency by 85 percent compared to the state-of-the-art virtualization-based fault-tolerant systems. Phantasy is now only compatible with shared storage, such as iSCSI (Internet Small Computer Systems Interface) and NAS (network-attached storage). In the future, we intend to extend this work to platforms using separate local disk. Furthermore, extending Phantasy to support a more complex failure recovery strategy which can deal with the case where the secondary VM fails is an interesting research problem and will be our future research topic.

## Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 61572044 and Grant No. 61170056). The contact author is Zhen Xiao.

## References

[1] J. Gray and D. P. Siewiorek, "High-availability computer systems," *IEEE Comput.*, vol. 24, no. 9, pp. 39–48, Sep. 1991.

[2] Amazon ec2 service level agreement. [Online]. Available: https://aws.amazon.com/cn/ec2/sla/

[3] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "Nonstop® advanced architecture," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun. 2005, pp. 12–21.

[4] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: Building high availability systems with commodity multi-core processors," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 470–481.

[5] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proc. 15th ACM Symp. Operating Syst. Principles*, 1995, pp. 1–11.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proc. 5th USENIX Symp. Networked Syst. Des. Implementation*, 2008, pp. 161–174.

[7] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "COLO: COarse-grained LOck-stepping virtual machines for non-stop service," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 3.

[8] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual machine synchronization for fault tolerance," in *Proc. USENIX Annu. Tech. Conf.* (Poster Session), 2008.

[9] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the performance of hypervisor-based fault tolerance," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–10.

[10] L. Maohua and C. Tzi-cker, "Fast memory state synchronization for virtualization-based fault tolerance," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2009, pp. 534–543.

[11] J. Zhu, Z. Jiang, Z. Xiao, and X. Li, "Optimizing the performance of virtual machine synchronization for fault tolerance," *IEEE Trans. Comput.*, vol. 60, no. 12, pp. 1718–1729, Dec. 2011.

[12] B. Gerofi and Y. Ishikawa, "RDMA based replication of multiprocessor virtual machines over high-performance interconnects," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2011, pp. 35–44.

[13] J. R. Lorch, A. Baumann, L. Glendenning, D. T. Meyer, and A. Warfield, "Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services," in *Proc. 12th USENIX Conf. Networked Syst. Des. Implementation*, 2015, pp. 575–588.

[14] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 52, pp. 74–80, 2013.

[15] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 456–468.

[16] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Jul. 2017, no. 325462–063US.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. 2nd Conf. Symp. Networked Syst. Des. Implementation - Vol. 2*, 2005, pp. 273–286.

[18] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 87–104.

[19] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 295–306.

[20] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2007, pp. 11–20.

[21] C. Isci, J. Liu, B. Abali, J. O. Kephart, and J. Kouloheris, "Improving server utilization using fast virtual machine migration," *IBM J. Res. Develop.*, vol. 55, no. 6, pp. 4:1–4:12, Nov. 2011.

[22] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. 24th Annu. Int. Symp. Comput. Archit.*, 1997, pp. 252–263.

[23] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2016, pp. 551–564.

[24] S. Ren, C. Li, L. Tan, and Z. Xiao, "Samsara: Efficient deterministic replay with hardware virtualization extensions," in *Proc. 6th Asia-Pacific Workshop Syst.*, 2015, Art. no. 9.

[25] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Leveraging hardware-assisted virtualization for deterministic replay on commodity multi-core processors," *IEEE Trans. Comput.*, vol. 67, no. 1, pp. 45–58, Jan. 2017.

[26] Mellanox Technologies, *RDMA Aware Networks Programming User Manual*, Rev 1.7, May 2015.
[27] D. Barak, "Working with unsignaled completions." 2014. [Online]. Available: http://www.rdmamojo.com/2014/06/30/working-unsignaled-completions/
[28] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Tech.*, 2008, pp. 72–81.
[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.
[30] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "OLTP-Bench: An extensible testbed for benchmarking relational databases," *Proc. VLDB Endow.*, vol. 7, no. 4, pp. 277–288, Dec. 2013.
[31] M. R. Hines, "QEMU: Features – Micro-checkpointing." 2015. [Online]. Available: https://wiki.qemu.org/Features/MicroCheckpointing
[32] M. R. Hines, "RDMA migration and RDMA fault tolerance for QEMU," *KVM Forum*, 2013.
[33] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 237–248.
[34] P. W. Frey and G. Alonso, "Minimizing the hidden cost of RDMA," in *Proc. 29th IEEE Int. Conf. Distrib. Comput. Syst.*, 2009, pp. 553–560.
[35] R. Friedman and A. Kama, "Transparent fault-tolerant Java virtual machine," in *Proc. 22nd Int. Symp. Reliable Distrib. Syst.*, 2003, pp. 319–328.
[36] J. Napper, L. Alvisi, and H. Vin, "A fault-tolerant Java virtual machine," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2003, pp. 425–434.
[37] P. J. Tsao, Y. F. Sun, L. H. Chen, and C. Y. Cho, "Efficient virtualization-based fault tolerance," in *Proc. Int. Comput. Symp.*, 2016, pp. 114–119.
[38] VMware, Inc., VMware vSphere 6 Fault Tolerance: Architecture and Performance, Tech. White Paper Jan. 2016.
[39] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield, "RemusDB: Transparent high availability for database systems," *VLDB J.*, vol. 22, no. 1, pp. 29–45, Feb. 2013.
[40] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 25–25.
[41] J. Zhang, X. Lu, and D. K. Panda, "High-performance virtual machine migration framework for mpi applications on sr-iov enabled infiniband clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 143–152.
[42] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *Proc. 3rd Int. Conf. Virtual Execution Environments*, 2007, pp. 169–179.
[43] A. Fischer, A. Fessi, G. Carle, and H. de Meer, "Wide-area virtual machine migration as resilience mechanism," in *Proc. IEEE 30th Symp. Reliable Distrib. Syst. Workshops*, 2011, pp. 72–77.
[44] S. K. Bose, S. Brock, R. Skeoch, and S. Rao, "Cloudspider: Combining replication with scheduling for optimizing live migration of virtual machines across wide area networks," in *Proc. 11th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2011, pp. 13–22.

**Shiru Ren** is currently working toward the PhD degree in the School of Electronics Engineering and Computer Science, Peking University. His research interests include virtualization technologies, operating system, fault tolerance, and distributed system. His recent research aims to implement efficient and low-latency virtualization-based fault tolerance using RDMA and PML.

**Yunqi Zhang** is working toward the PhD degree in the Computer Science and Engineering Department, University of Michigan. His research interest includes architecting data centers for high efficiency and low latency. He is a member of the ACM and IEEE.

**Lichen Pan** received the bachelor's degree from Peking University, in 2017. He is currently working toward the doctoral degree in the School of Electronics Engineering and Computer Science, Peking University. His research interests include virtualization technologies, fault tolerance, and cloud computing.

**Zhen Xiao** received the PhD degree from Cornell University, in January 2001. He is a professor with the Department of Computer Science, Peking University. After receiving of PhD degree he worked as a senior technical staff member with AT&T Labs - New Jersey and then a Research Staff Member with IBM T. J. Watson Research Center. His research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of the ACM and IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.