Thread-Based Live Checkpointing of Virtual Machines

Vasinee Siripoonya Kasidit Chanchio
Department of Computer Science
Faculty of Science and Technology, Thammasat University
Patumtani, THAILAND 12121
siripoonya@hotmail.com, kasidit@cs.tu.ac.th

Abstract— **Virtual machine checkpointing is the mechanism to save virtual machine state to a file for later recovery. Traditional checkpointing mechanisms can suffer a long delay and cause a long disruption of services since they have to stop virtual machines to save state, which could be large. In this study, a novel Thread-based Live Checkpointing (TLC) mechanism is proposed. This mechanism leverages the pre-copy live migration mechanism introducing a checkpoint thread, which is responsible for the majority of the checkpointing activities. While the checkpoint thread is saving the virtual machine state to persistent storage, the virtual machine thread is allowed to progress with normal execution. However, the virtual machine thread will be periodically interrupted to incrementally copy dirty memory pages to a hash table. The interruptions will occur until the final stage of checkpointing is reached. This approach is implemented in KVM and its performance evaluations are conducted using NAS parallel benchmarks. Experiments show that this approach can provide high levels of virtual machine responsiveness during checkpointing. It can also reduce the checkpointing overheads to as low as 0.53 times of that of the traditional approach, when operating on a virtual machine running memory intensive workloads.**

Keywords-virtual machine, checkpointing, virtualization

I. INTRODUCTION

Cloud computing has recently emerged as a popular computing platform. Partial reasons behind its successes are the advantages of virtualization technology such as server consolidation and supports for legacy applications. However, despite these advantages, virtual machines (VMs) in the Cloud are still vulnerable to hardware and software failures. A crash of a VM also crashes all applications running on it causing service disruptions and loss of computation. To cope with these problems, VM checkpointing can be employed to provide fault-tolerance by saving the state of a VM to persistent storage. If failures occur, the saved state can be used to resume computation. Another advantage of VM checkpointing over traditional process-level checkpointing is transparency. VM checkpointing mechanisms are highly transparent to applications and the guest OS of a VM since they are implemented at hypervisor-level. The mechanisms do not require any modification to applications and the guest OS, and do not depend on a particular OS kernel to operate.

Despite the advantages, implementing efficient VM checkpointing mechanism is a true challenge, especially when the targeted VMs have large working set sizes and run memory intensive workloads. Traditionally, hypervisors such as KVM and Xen [1, 2] already have a capability to save VM state to files, which could be leveraged for VM checkpointing. However, such approach requires the VM to stop during checkpointing, causing disruptions to interactive services and long checkpointing delays.

A. Motivations

There are four motivations that inspire our work. First, the existing hypervisors already have capabilities to save, restore, and live migrate a VM [1, 2]. We leverage these mechanisms to implement our solution. Second, since we consider a VM to be a computing instance in the Cloud [3], the VMs can have different service level agreements and priorities to utilize underlying hardware resources. In this study, we present our checkpointing mechanism as a new VM service capability. Third, the uses of multi-core CPUs are common on servers and suitable for multi-threaded applications. Our approach utilizes the multi-core architectures by creating a new thread to assist checkpoint operations. Finally, the decrease in price of memory [4] permits large memory installation on servers, which in turn allows our mechanism to practically store partial VM state information in memory before writing them to persistent storage.

B. The Proposed Solution

Based on the problems and motivations above, this paper presents the design and implementation of a novel VM checkpointing mechanism, namely the *Threaded-based Live Checkpointing (TLC)* mechanism. We define the term *Live Checkpointing* as the ability to perform checkpointing operations while allowing computation to progress. In the TLC design, a new thread, namely the checkpoint thread, is created to save a snapshot of VM state to a checkpoint file. At the same time, the VM thread can progress with its computation; however, TLC will interrupt the computation periodically to incrementally copy dirty pages to a hash table. Finally, after the checkpoint thread finishes saving the snapshot, it will collaborate with the VM thread to save the final part of the VM state and conclude the TLC operation. This paper also presents two optimization techniques to reduce the size of the checkpoint information. We have implemented a TLC prototype on KVM, a full virtualization hypervisor for Linux [1]. However, the TLC design can be adapted for other hypervisors as well.

978-0-7695-4489-2/11 $26.00 © 2011 IEEE
DOI 10.1109/NCA.2011.28

We have conducted a number of experiments to evaluate TLC overhead, latency, memory requirement, and responsiveness against those of the traditional VM checkpointing mechanism using four serial NAS parallel benchmark kernels [5]. In this paper, we define 1) the checkpoint overhead as the time increase on VM computation due to a checkpointing operation, 2) the checkpoint latency as the time used to complete the operation, and 3) memory requirement as the amount of memory needed during the operation, and 4) responsiveness as the interactive bandwidth the VM can respond to a remote computer during the operation.

Experimental results show that TLC performance depends highly on memory update behaviors of VM workloads. They also show the following benefits of TLC:

1. TLC provides high levels of responsiveness, even though the VM workloads are memory intensive.
2. The TLC mechanism reduces checkpoint overhead to as low as 0.53 times of that of the traditional stop-and-copy checkpointing mechanism when operating on a VM running the memory intensive FT (class B) benchmark.
3. TLC can complete a checkpointing operation within a finite amount of time. TLC checkpoint latency depends on the working set sizes of VMs.

However, there are tradeoffs for these performance gains. First, TLC needs to create a new checkpoint thread. Second, its checkpoint latency can take up to 1.9 times of the traditional one. Finally, it requires extra memory to hold the hash table, which can take up to 0.7 times of the VM's working set size.

Despite the tradeoffs, we believe that TLC is suitable for the Cloud especially for its *live* property, which allows VMs to progress during checkpointing even when they are running computation and memory intensive applications. This capability is important for VMs that have large working set sizes or host multiple applications at the same time. Moreover, TLC can be applied for live migration, especially for VMs running memory intensive workloads. Due to space limitation, we will discuss TLC live migration in a separate work.

Regarding the additional CPU resources required by TLC, we believe that they are justifiable for Cloud computing environments, where VMs have different priorities to utilize underlying hardware resources based on their service level agreements. When a high priority VM creates a checkpoint thread for its VM checkpointing operation, some kind of priority based scheduling mechanism, such as that of Linux, can be employed to manage CPU resources. Since the checkpointing thread is I/O bound, it is unlikely to consume all cycles of a CPU core.

In term of extra memory required by TLC, we argue that the memory sizes on modern servers are usually large and parts of it can be used to store checkpoint information. Since the memory prices tend to be decreasing [4], users can install more memory on their machines. Recent checkpointing works [6] have stored the entire checkpoint files in memory of local or remote host computers to improve checkpoint/restart performance. The TLC, on the other hand, only requires memory to temporarily store a portion of the checkpoint, not all of them, and will eventually write checkpoint information to a file. In case the physical memory on the server is not enough, TLC will automatically use virtual memory.

This paper organizes as follows. Section 2 discusses related works. We present the design and implementation of TLC as well as the correctness discussion in Section 3. Section 4 evaluates TLC prototype and discusses experimental results. Finally, Section 5 gives the conclusion and discusses future works.

II. RELATED WORKS

A. Levels of Operations

Traditionally, checkpointing is performed on a process. A process can be checkpointed at different levels of operations. Programmers may implement checkpointing mechanisms at *application-level* where they insert codes to save immediate results of their programs to checkpoint files [7]. Compilation and program analysis techniques are used to assist this kind of checkpointing [8, 9]. This approach is efficient; however, it is not transparent to users. Alternatively, programmers can use *user-level* checkpointing by linking their object codes to a checkpointing library such as libckpt[10] and Condor [11]. Although this approach is more transparent than the application-level approach, it depends on specific compile-time and runtime systems. Finally, checkpointing can be done at *system-level,* where the checkpointing mechanisms are integrated into OS kernels. Systems such as BLCR [12] and TICK [13] use this approach. Although providing checkpointing transparency, this approach is highly OS dependent.

TLC can be categorized as a *virtual machine-level* checkpointing mechanism. Instead of saving and restoring process state, this approach saves and restores state of a VM. The checkpointing mechanisms at this level are highly transparent to applications and guest OS. Since VMs are basic computing instances in the Cloud, VM checkpointing is an enticing solution to provide cloud reliability. However, major obstacles to this approach are in the development of efficient checkpoint-restart mechanisms for VMs that run demanding workloads and have large working set sizes. A few solutions have been proposed. We will discuss them in Section C below.

B. Checkpointing Mechanisms

In term of mechanisms, a common approach for checkpointing is the *stop-and-copy* approach. This mechanism stops the execution of a process while saving the entire process's state to a checkpoint file. On the other hand, *incremental chekpointing* is an optimized mechanism that attempts to reduce checkpointing overhead by saving only the changed state information from the previous checkpoint [14, 13]. This approach can help reduce the checkpoint size substantially. To further reduce checkpointing overheads, *diskless checkpointing* can be used to save process state to memory instead of disks [15]. Recent research [13, 6] shows that this approach can improve performance substantially. However, the checkpoint data will be lost if the host computer containing it fails. A combined approach, namely the *multi-*

level checkpointing, has been proposed and used on supercomputers at LLNL [6]. This approach combines the diskless and traditional stop-and-copy mechanisms together to improve checkpointing performance. This mechanism saves process state to a ramdisk on every short checkpoint interval and periodically flushed the saved state in the ramdisk to permanent storage. The extensive uses of ramdisk in this research confirm practical uses of diskless checkpointing.

An existing work that is closely related to ours is the concurrent checkpointing mechanism [16]. This approach creates two new threads, the copier and writer threads, to perform checkpointing operations. The copier thread copies memory pages to a buffer, while the writer writes pages in the buffer to disk. During checkpointing, the computing thread can continue as usual. In case the computing thread wants to modify a memory page, it must copy that page to the buffer first before making modification. Our approach differs from the concurrent checkpointing mechanism in two aspects. First, we use incremental checkpointing to periodically copy changed pages to a hash table rather than copying a page to the buffer on every memory write. Second, our checkpoint thread saves memory pages to disk directly, while the concurrent checkpointing copies pages to the buffer and uses the writer thread to save the buffer to disk.

Another mechanism that is close to our work is the Copy-On-Write (COW) checkpointing mechanism offered in libckpt [10]. In this mechanism, a cloned process is created (using fork system call) to write a memory snapshot to disk while allowing the original process to continue normal execution. The memory snapshot is write-protected at the level of the OS. If the original process wants to modify a memory page in the snapshot, the OS will make a new copy of that page, modify the new page, and operate on it from then on. If the original process updates memory intensively, this approach may cause a long delay on the original process due to a large number of page copying at the beginning of a checkpointing operation. The problem could be worse if the COW approach is applied on a VM running memory intensive workloads. Our mechanism avoids this problem by using incremental checkpointing on the VM thread rather than using the COW mechanism.

C. Virtual Machine Checkpointing

VM checkpointing can be leveraged from the existing capabilities available on hypervisors for saving and loading VM state to and from persistent storage. These mechanisms follow the stop-and-copy approach, which is non-live and can cause shortages of services as mentioned earlier.

VM checkpointing can also base on the traditional pre-copy live migration mechanism. KVM, for example, has the migration-to-file feature [1]. Moreover, the pre-copy live migration mechanism can be easily adapted to write migration information to a file instead of a network. This approach provides live checkpointing; however, it cannot efficiently handle the checkpointing of VMs with memory intensive workloads. Since the above pre-copy mechanism would save dirty pages to a file periodically and enter the final round of saving if and only if the number of dirty pages generated by the VM is below a certain threshold, the checkpointing of VMs with memory intensive workloads will take a long time and may complete after the execution of memory intensive applications on the VMs finishes.

REMUS [17] provides high availability to a Xen-based VM by periodically synchronizing VM state with a backup VM. It also supports VM checkpointing by redirecting data to a daemon process on the same host instead of to the backup VM. Thus, a backup VM or the backup daemon is required throughout the VM's lifetime. REMUS performs synchronization by copying dirty pages to the backup periodically, which could be as often as every 25 msec. Kemari [18] is similar to REMUS. However, Kermari performs synchronization only when the VM uses I/O devices.

Unlike REMUS and Kermari, TLC attempts to provide an efficient live checkpointing mechanism *not* VM fail-over. Therefore, instead of maintaining a backup VM, TLC merely creates a checkpoint thread on demand to accomplish its tasks. The checkpoint thread will be terminated when VM state is saved to persistent storage. Finally, TLC requires additional memory only during its checkpointing operation, not throughout the VM lifetime like in the fail-over case.

III. THREAD-BASED LIVE CHECKPOINTING

In this paper, we assume the crash failure model, where the VM is crashed when failures occur. The checkpoint files are stored on local or shared persistent storage. On recovery, the hypervisor resumes VM computation on healthy computers by loading state information from these files.

A. Design

The TLC mechanism is illustrated in Figure 1. From the figure, the mechanism consists of multiple stages of operations across two threads. The *VM thread* is the original thread that the hypervisor uses to perform VM execution. When the hypervisor receives a checkpointing request, it creates a *checkpoint thread* to assist the VM thread capturing and saving VM state to a checkpoint file. We define the VM state to consist of memory state, device state, and disk state of a VM. From Figure 1, we describe the TLC mechanism in four stages as follows:

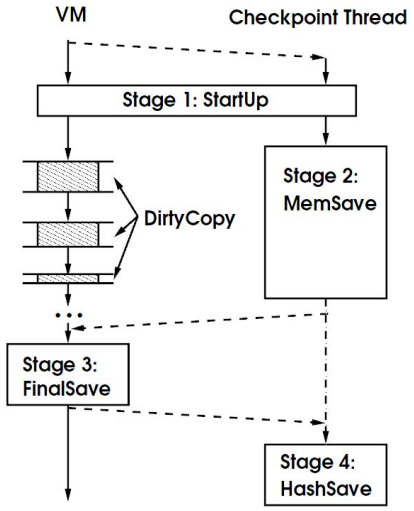

Figure 1. TLC Mechanism

Stage1: The checkpoint thread performs a start up operation, where data structures for capturing and saving VM memory state are initialized. The checkpoint thread allocates bit vectors to keep track of dirty pages, instructs hardware virtualization driver to perform dirty page tracking, initializes thread synchronization data structures, and opens a checkpoint file. The VM thread is paused during this stage.

Stage 2: After Stage 1, the checkpoint thread starts saving all memory pages to a checkpoint file. At the same time, the VM thread continues with computation. However, it will be interrupted every ***I*** interval. The interrupt handler would be invoked to copy all new dirty pages the VM generated during the past ***I*** interval to the hash table. We use *uthash* in our implementation [19]. Note that the size of the hash table will expand if and only if a new page is added. If the page already exists, the new contents will replace the old one. In our current implementation, we set ***I*** to 100 msec following the default value of KVM live migration. However, on each interruption, TLC interrupt handler will copy all dirty pages to the hash table at once. This is different from KVM live migration mechanism, where the maximum number of dirty pages saved each time is limited by a migration bandwidth.

At this stage, the checkpoint thread is responsible for saving VM memory snapshot at the moment a checkpointing operation starts, while the interrupt handler handles the capturing of new dirty pages generated afterward.

Stage 3: When the checkpoint thread finishes saving all memory pages, it will wait for the VM thread to perform operations in Stage 3. At this stage, the interrupt handler on the VM thread would be invoked as usual. However, once it learns that the checkpoint thread finishes Stage 2, the handler pauses VM computation and writes the last set of dirty pages as well as VM device state to the checkpoint file. It also handles disk state checkpointing, and then resumes VM computation.

Stage 4: Finally, the checkpoint thread saves contents of the hash table to the checkpoint file, closes the file, and terminates.

We have developed two optimization schemes to reduce the size of checkpoint data as follows.

Opt1: During Stage 2, the checkpoint thread will write contents of a memory page to the checkpoint file if and only if that page has not been copied to the hash table. This first optimization scheme can help reduce the amount of data the checkpoint thread has to save during Stage 2 substantially.

Opt2: TLC will exclude the last set of dirty pages saved to the checkpoint file during Stage 3 off the hash table to reduce the table's size. This also helps reduce time to save contents of the hash table to the checkpoint file at Stage 4.

Disk State Checkpointing: To save the VM disk image at each checkpoint, the copy-on-write disk storage such as KVM's qcow2 [20], btrfs [21], and zfs [22] can be used. In this work, the qcow2 disk image is used. Our VM disk image consists of two parts: the base and overlay disk images. The base disk image stores original data and remains unchanged, while the overlay disk stores all changes made to the base disk. The disk read operations first look for recent data from the overlay before retrieving the data from the base disk.

At Stage 3, after saving VM device state, TLC will create a new overlay layer on top of the current overlay disk, and change KVM internal data structures to make the new overlay image the working one. Now, there are three disk layers. The working overlay disk will be the only one that stores changes made by future computation after checkpointing. The rests are read-only. If failures later occur, the VM can be recovered using the checkpoint file together with the old overlay and base disk images. Disk contents in the old overlay would be consistent with the VM state in the checkpoint file since they are preserved. TLC uses the existing mechanism of KVM to restart the VM on recovery.

B. Implementation

We have modified the pre-copy live migration mechanism of KVM (kvm-88) to implement TLC. First, we have added codes to create the checkpoint thread when a checkpoint command is given on KVM command line console. Upon creation, the checkpoint thread is assigned to run on a different CPU core from that of the VM using the *sched_setaffinity* system call. Then, it enters Stage 1 to initialize values to TLC data structures, open a checkpoint file, and call KVM driver to keep track of dirty pages. Since KVM relies on hardware support to provide full virtualization, specific KVM driver routines must be invoked to instruct MMU to keep track of dirty pages and to obtain dirty bit information. This dirty page logging mechanism has been used extensively in KVM pre-copy live migration [1].

TLC uses two bit vectors for keeping track of memory page saving and checkpointing optimization. They are the *Save_Pages* and *Dirty_Pages* bit vectors. Each bit element in these vectors corresponds to a page in VM memory. Thus, the number of bits in each vector is equal to the number of memory pages of the VM. We use the *Save_Pages* vector to indicate memory pages that the checkpoint thread has to save to the checkpoint file. On the other hand, the *Dirty_Pages* vector is used to mark dirty pages that the interrupt handler has to copy to the hash table on each VM interruption. At Stage 1, we set every bit of the *Save_Pages* vector to 1 and that of the *Dirty_Pages* vector to 0. During Stage 2, the checkpoint thread will save a memory page to the file if and only if the corresponding bit value of that page in the *Save_Pages* vector is set. After the page is saved, the checkpoint thread will reset that bit. At the same time, when the interrupt handler is invoked on the VM thread, the handler will call KVM dirty page logging mechanism to retrieve dirty page information the VM generated during the last 100 msec interval, and set values in the *Dirty_Pages* vector accordingly. Then, the handler will copy contents of every dirty page to the hash table, and perform the Opt1 optimization by *resetting* the values of the *Save_Pages* elements corresponding to the dirty pages. Finally, it will reset the *Dirty_Pages* vector and resume VM computation.

Since the two threads can access both bit vectors at the same time, mutual exclusion must be applied to prevent race condition. In order to save some memory space, we allocate 2048 mutex variables for 1 GB (2^{19} pages) of memory rather

than assigning a mutex variable to each page. Each variable is used to provide atomic access to the two bit vectors in our implementation. The mapping of memory page addresses to a mutex variable is similar to that of direct mapped cache.

After the checkpoint thread finishes Stage 2, the VM thread will perform Stage 3 in the next round of the interruption. The handler will obtain dirty bit information from KVM driver, and then save all the dirty pages indicated in the *Dirty_Pages* vector to the checkpoint file. In performing the Opt 2 optimization, these pages will be excluded from the hash table to reduce the number of page saving at Stage 4.

C. Correctness and Performance Discussions

In TLC design, we have developed the following two mechanisms to guarantee correctness.

1). Since the two bit vectors in our implementation are modified and retrieved concurrently by two threads during Stage 2, TLC uses the mutual exclusion mechanism described earlier to prevent race conditions on them.

2). However, we do not use mutual exclusion to manage concurrent accesses to memory pages because it may cause long waiting time, and thus increase checkpoint latency. Therefore, during Stage2, it is possible for the checkpoint thread to save a memory page, whose contents are being updated by the VM thread at the same time. As a result, the contents of the saved page could be corrupted.

In TLC, operations in Stage 2 are allowed to generate corrupted pages. We handle this problem in two cases.

Case 1: if the next round of interruption on the VM thread occurs and Stage 2 has not yet finished, the updated page contents would be copied to the hash table and eventually saved to the checkpoint file at Stage 4. Thus, the updated contents would be recorded later than the corrupted contents in the checkpoint file.

On recovery, the VM will load contents in the checkpoint file from beginning to memory. As a result, the corrupted contents would be loaded to memory first and then overwritten later by the updated contents. Therefore, the contents in the VM memory would eventually be correct.

Case 2: In case the next iteration occurs after Stage 2 finished, the interrupt handler will perform operations in Stage 3 by saving the newly modified pages to the checkpoint file and excluded the previously stored contents of the page (if any) off the hash table. As a result, the correct contents of that page will be stored last in the checkpoint file, which would ensure correct memory contents on VM recovery.

In term of performance, once the checkpointing operation starts, TLC will complete the operation within a finite amount of time. Since TLC latency is defined by the amount of time to perform Stage 1 to 4, we will consider the timing on each stage. First, we know that Stage 1 take a constant time. The duration of Stage 2, on the other hand, has its upper bound value proportional to the VM memory size. Once the TLC starts Stage 3, there will be no more new dirty pages generated by the VM. Therefore, the size of the hash table and the number of dirty pages at that moment would be fixed. As a result, the time required for Stage 3 and 4 of TLC will be finite. Since the timing of every stage is finite, the time required for a TLC operation is also finite.

IV. EVALUATION

We have conducted two experiments to evaluate TLC. The former evaluates TLC performance and compares it against the traditional stop-and-copy mechanism of KVM. The latter uses *iperf* to measure VM responsiveness during checkpointing. All experiments are performed on a HP proliant ML350 G6 server with two Quad core Xeon 5500 2.40Ghz (VT-x) processors, 72 GB RAM, and a HP Smart Array with RAID 0 (256 MB disk cache) on two 250 GB SATA Disks. The server runs Ubuntu 10.10 with Linux 2.6.32 kernel. In each experiment, we invoke a VM running Fedora 11 with Linux 2.6.29 kernel and X11 GUI as the Guest OS. The VM is configured to use 1 CPU core and connect to LAN via a virtio network interface.

A. Checkpointing Performance

In the first experiment, we measure checkpointing performances by conducting a checkpointing operation while the VM runs a benchmark program. We have installed the serial version of NAS Parallel Benchmark (NPB3.3) kernel [5] EP.B (EP Class B), MG.B (MG Class B), IS.C (IS Class C), and FT.B (FT Class B) to generate various workloads on the VM. All benchmarks are computation intensive, but have different memory usage behaviors. Since the Working Set Size (WSS) of EP.B running on our VM is around 345 MB, we configure the VM for EP.B to have 512MB RAM. The VM running MG.B, on the other hand, required around 745MB WSS. Therefore, we configure it to have 1GB RAM. Finally, the VM running IS.C and FT.B use around 1.3GB and 1.6GB WSS, respectively. Consequently, we configure the VMs for these programs to have 2GB of RAM. All programs except the EP are memory intensive. Every data point in our experiments, unless stated otherwise, is an average of 10 runs.

We evaluate three checkpointing mechanisms in this experiment. First, we evaluate the TLC mechanism (denoted by **TLC)**. Second, we measure the performance of the traditional stop-and-copy mechanism of KVM (denoted by **savevm)**. Finally, we evaluate a variation of TLC called **T-onesave**. The T-onesave uses the checkpoint thread like TLC;however, it disables the incremental checkpointing during Stage 2, and lets the operations in Stage 3 save all dirty pages to the checkpoint file in one attempt. We compare TLC with the T-onesave to show the benefits of incremental checkpointing and the hash table in TLC design. Note that we denote **Normal** for normal execution without checkpointing. Every time when we run an experiment, we fresh invoke a VM and execute a benchmark program on it. After the WSS of the VM reaches its maximum, a checkpointing operation is launched. Every checkpointing operation is invoked live without stopping the VM. The VM execution will continue automatically after the checkpointing completes. The VM is configured to use the same clock timer with the host computer (using KVM's `-localtime` option). Therefore, checkpointing overheads would already be included in the timing results reported at the end of the benchmarks' execution. Note that disk write performances in our experiments are relatively high across-the-board due to the uses of HP 256 MB disk cache and KVM internal I/O buffers.

In this experiment, we evaluate the checkpointing performance in three aspects: checkpoint overhead, checkpoint latency, and memory requirement.

1) Overhead. Figure 4 shows normal execution time and the execution time when different checkpointing mechanisms are used on VMs running the benchmarks. From the Figure, the execution time of benchmarks running TLC is generally better than (or at least comparable to) that using other methods.

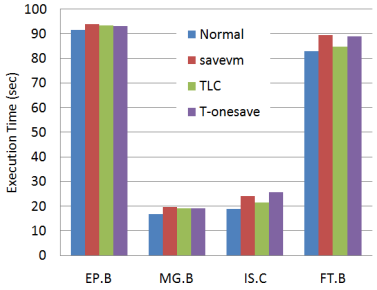

Figure 4. The execution time of 4 NAS benchmarks.

The checkpointing overhead is the amount of time increase on VM execution due to checkpointing operations. From the design in Figure 1, the overhead of TLC can be modeled as

$$Overhead = Startup + DirtyCopy + FinalSave,$$

where *Startup* is the start up time to set up necessary data structures (Stage 1), *DirtyCopy* represents the accumulated amount of time the VM periodically stops (every 100 msec) to save dirty pages to the hash table, and *FinalSave* is the time TLC spends to save the last set of dirty pages and device state to a checkpoint file (Stage 3).

The overhead of the T-onesave mechanism, on the other hand, consists of the *Startup* and *FinalSave* time only. However, the *FinalSave* of the T-onesave will generally be higher than that of TLC because all the dirty pages produced during the memory saving operations (Stage 2) of the checkpoint thread are saved at Stage 3 in one attempt. Finally, the overhead of the savevm is the time the VM stops to save all VM state to the checkpoint file.

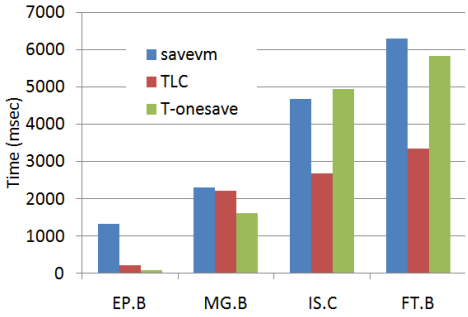

Figure 5. Checkpointing overheads of 4 NAS benchmarks.

We have measured the overheads of these methods and shown them in Figure 5. From the Figure, TLC demonstrates low overhead on the EP benchmark because the EP generates small amount of dirty pages during execution resulting in small *DirtyCopy* and *FinalSave* values. Note that the *Startup* time is negligible in all cases. For the IS and FT benchmarks, the TLC overheads are about 0.57 and 0.53 times of the overheads of savevm, respectively. The TLC overhead of MG is about 0.95 times of that of the savevm. It demonstrates much lower overheads than that of the savevm on the EP, IS, and FT benchmarks. TLC overheads are better than those of the T-onesave on the IS and FT benchmarks. However, TLC shows little higher overheads than those of the T-onesave on the EP and MG.

For the EP and MG benchmarks, which have high memory update locality, the TLC will most likely copy redundant dirty pages repeatedly to the hash table causing the *DirtyCopy* value to be high. On the other hand, the T-onesave waits to save all dirty pages to file only once. Therefore, the T-onesave would perform better than the TLC on these benchmarks.

In case of the IS and FT benchmarks, TLC demonstrates better performance than T-onesave for two reasons. First, the IS and FT benchmarks generate large numbers of dirty pages but have low memory update locality. Each benchmark produces a large number of dirty pages over a vast area in VM memory space. In case of TLC, the interrupt handler will copy most of the non-redundant dirty pages to the hash table during Stage 2. Therefore, TLC will have only a small amount of dirty pages left to save to the checkpoint file during Stage 3. On the other hand, T-onesave will have to save all dirty pages to file once at the end of the checkpointing operation (Stage 3). Second, saving data to a file is more costly than copying data to the hash table. Since T-onesave has larger numbers of dirty pages to save at Stage 3 than that of the TLC, T-onesave will suffer higher overheads than TLC as illustrated in Figure 5. Although the uses of incremental checkpointing and hash table in TLC are intended mainly for providing interactivity during a checkpointing operation, they also help improve the overhead performance in the IS and FT cases.

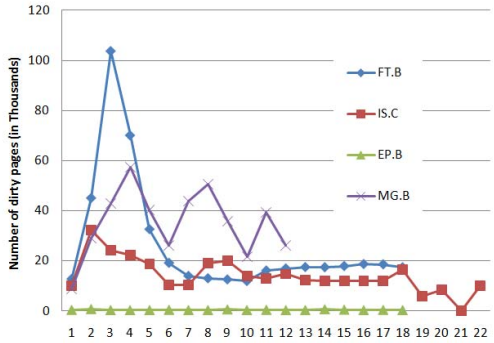

Figure 6. Number of dirty pages copied to a hash table every 100 msec by the interrupt handler of TLC.

To further analyze TLC activities, Figure 6 illustrates detailed information of dirty page copying activities of TLC taken from a sample run of each benchmark. Every data point in the graph represents a number of dirty pages TLC copied to the hash table every 100 msec. The x-axis in the figure represents each round the copying of dirty pages occurs, while the y-axis indicates the number of copied pages in thousands. From the Figure, the number of dirty pages of the FT benchmark is high (a little over 100,000 pages) at the beginning. Then, it reduces to around 20,000 pages in the middle and stays at that level till the final round. On the other hand, the number of copied dirty pages of the IS at the

beginning round is approximately 30,000 pages and then stays in the range of 10,000 to 20,000 pages in the later rounds. The average numbers of dirty pages copying per round are 14,165 and 26,436 pages for IS and FT, respectively. Note that the IS has more rounds than the FT because it produces lesser number of dirty pages on each round of copying. Therefore, the IS will spend lesser time copying dirty pages to hash table on each iteration. As a result, by the time the checkpoint thread finishes Stage 2, the IS will be interrupted more frequent than the FT.

Figure 6 also confirms high memory updates behavior of the MG. It shows that the number of dirty pages of MG is about 60,000 pages at the beginning and stays on average at 35,191 pages. The MG exhibits higher average copied dirty pages per round than that of the IS and FT. The number of dirty pages in the last round of the MG is also higher (at 26,217 pages) than those of the IS and FT (10,128 and 17,409 pages, respectively).

2) Checkpoint Latency. For the TLC mechanism, checkpoint latency is the length of time the checkpoint thread uses to save VM state information to a checkpoint file. It can be described as

$$Latency = Startup + MemSave + FinalSave + HashSave,$$

where *MemSave* is the time the checkpoint thread spends to save VM memory to a checkpoint file (Stage 2), and *HashSave* is the time it takes to save contents of TLC hash table to the file (Stage 4). On the other hand, the checkpoint latency of the savevm is the same as its checkpoint overhead.

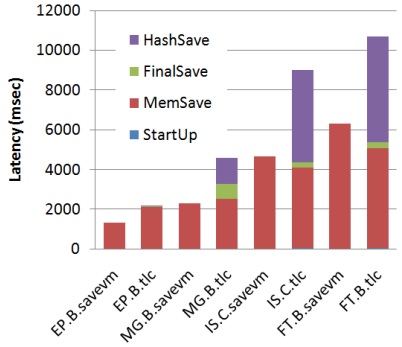

Figure 7. Analysis of the checkpoint latency of TLC and savevm mechanisms. The label on the x-axis shows a benchmark's name following by the name of a checkpointing mechanism.

From Figure 7, the checkpoint latencies of TLC on the four benchmarks are ranging from 1.6 to 1.9 times higher than those of savevm because they include the time to save memory pages stored in the hash table to the checkpoint file. The dominating factors of TLC latency are the *MemSave* and *HashSave* time. Although the optimization scheme of TLC can help reduce the number of memory pages copied during Stage 2 substantially (see next section), *MemSave* is still significant. On the other hand, the *HashSave* times are varied across benchmarks depending on the size of the hash table. From the Figure, the *HashSave* times of IS and FT are as high as the *MemSave* time, while the *HashSave* time of MG is about a third of the *MemSave*. The EP's *HashSave* time is very small due to low memory updates.

3) Memory Requirements. TLC requires additional memory for the hash table. Table 1 shows the average number of pages in VM memory, working set, and the hash table required by each benchmark. The table also reports the proportion of the hash table size against the working set size (Hash/Work Set in the Table).

From the Hash/Work Set record in Table, TLC requires memory space of approximately 0.7 times of the FT's WSS for the hash table. In case of the IS, the hash table size required is approximately 0.68 times of IS's WSS. The hash table sizes required for the MG and EP are 0.47 and 0.01 times of the WSS, respectively.

TABLE I. THE AVERAGE NUMBER OF PAGES REQUIRED BY TLC.

	EP.B	MG.B	IS.C	FT.B
VM memory	133184	264256	526400	526400
Working Set	108289.1	237016.1	400195.9	462750.3
Hash Table	1691.7	112459.5	273753.9	325022.3
Hash/Work Set	0.015	0.47	0.68	0.70

TLC optimization scheme can help reduce the number of pages to be saved to the checkpoint file substantially. Table 2 shows the number of pages TLC skipped saving due to the **Opt1** and **Opt2** optimizations on each benchmark (See Opt1 Skipped and Opt2 Skipped lines in the Table). The Table also shows the number of memory pages saved to checkpoint files in the Saved Pages line. Finally, the Skipped/Saved line shows the proportion of the total number of skipped pages (Opt1 Skipped + Opt2 Skipped) against the number of pages saved to the checkpoint file (Saved Pages). For the IS, the total number of skipped pages is approximately 0.43 times of the number of pages in the checkpoint file. The proportion of skipped memory for the MG, FT, and EP are 0.41, 0.3, and 0.01 times of the size of the checkpoint file, respectively.

TABLE II. TABLE SHOWING TLC OPTIMIZATION PERFORMANCE.

	EP.B	MG.B	IS.C	FT.B
Opt1 Skipped	1173	64853.2	191076.5	169797.9
Opt2 Skipped	307.6	38085.7	11761.8	14445.6
Saved Pages	108500.2	246536.7	471111.5	603529.1
Skipped/Saved	0.01	0.41	0.43	0.30

B. Responsiveness

In the previous experiment, TLC stops VM computation for only a short duration to save dirty pages every 100 msec. During this time, the VM is out of service. The maximum out-of-service time on each benchmark can indirectly imply VM responsiveness during a TLC operation. Figure 8 shows the maximum out-of-service time during Stage 2 and Stage 3 on every benchmark. We also show the maximum out-of-service time of the savevm and T-onesave mechanisms in the Figure. For the savevm and T-onesave, the maximum out-of-service time is equivalent to the checkpoint overhead. On the other hand, the maximum out-of-service time of TLC on the EP, MG, IS, and FT benchmarks are 0.03, 0.31, 0.08, and 0.08 times of those of the savevm, respectively.

In the second set of our experiments, we directly measure the responsiveness of the TLC mechanism and that of the traditional savevm using *iperf* utility. In our experimental setup, we run an *iperf* server and a benchmark program

concurrently on the same virtual machine. During the execution of the benchmark, we run the *iperf* client on another host computer on the same LAN with the host of the VM. The two hosts are connected via a Gigabit Ethernet network. We choose the MG and FT benchmarks for this experiment due to their memory intensive behaviors. Note that the VM's workloads in this experiment are different from those in the previous experiment since the VM in this experiment has an *iperf* server running along with the benchmarks.

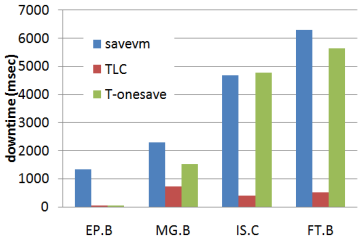

Figure 8. The maximum out-of-service time on the VM during a checkpointing operation.

Experimental results show that TLC exhibits much better responsiveness than that of savevm. TLC allows the VM to progress during checkpointing, while savevm stops the VM entirely. When TLC is performed on the VMs running the FT and MG benchmarks, there are little fluctuations of *iperf* bandwidth as shown in the dashed circle in Figure 9 and 11, respectively. Bandwidth reports during TLC checkpointing of the FT and MG are relatively high. On the other hand, the bandwidth drops to zero for several seconds when savevm is performed on the FT and MG as shown in the dashed circles in Figure 10 and 12, respectively.

V. CONCLUSION AND FUTURE WORKS

In this paper, we have proposed a novel Thread-based Live Checkpointing mechanism that allows VM checkpoint and VM computation to progress concurrently. We have implemented the TLC prototype on KVM and tested it with computation and memory intensive NPB benchmarks. Based on the experimental results, we believe that TLC is a promising checkpointing solution for Cloud computing due to its high transparency and live checkpointing capabilities. In our future works, we plan to implement TLC for multiprocessor VMs, and investigate a set of rules to automatically switch TLC operation to T-onesave when high memory update locality is detected.

[1] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," *Proc. of Linux Symposium*, 2007.

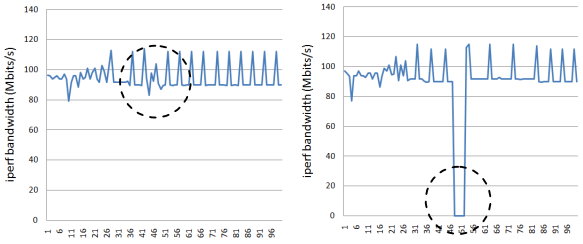

Figure 11 and 12 show the iperf client results on a VM running the MG benchmark using TLC and savevm mechanisms, respectively.

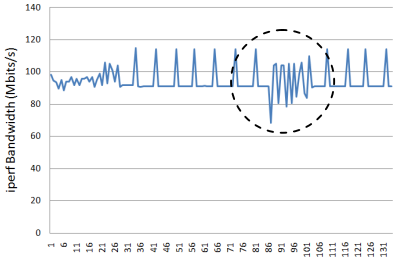

Figure 9. The iperf client results on a VM running FT benchmark and performing TLC mechanism.

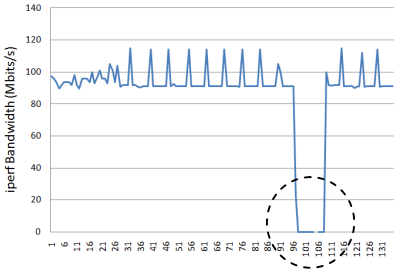

Figure 10. The iperf client results on a VM running FT benchmark and performing savevm mechanism.

[2] P. Barham, et al. "Xen and the Art of Virtualization," *Proc. of the ACM SOSP*, 2003.

[3] Amazon Web Services, http://aws.amazon.com/ec2/

[4] ITRS 2010 Update, Table 7a and 7b, http://www.itrs.net/

[5] NPB, http://www.nas.nasa.gov/Resources/Software/npb_changes.html

[6] A. Moody, et al. "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," *ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2010.

[7] A. Beguelin, E. Seligman and P. Stephan, "Application level fault tolerance in heterogeneous networks of workstations," *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[8] K. Chanchio, et al. "Data collection and restoration for heterogeneous process migration," *Software-practice & experiences*, 2002.

[9] E. N. Elnozahy, et al. "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.* 34, 3 (September 2002)

[10] J. S. Plank, et al. "Libckpt: Transparent Checkpointing under Unix," *Proc. of the 1995 Winter USENIX Tech Conf.* 1995.

[11] M. Litzkow and M. Solomon, "Supporting Checkpointing and Process Migration Outside the UNIX Kernel", *Usenix Conf.* 1992.

[12] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," *Proc. of SciDAC* 2006.

[13] R. Gioiosa, et al. "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers," *Proc. of the 2005 ACM/IEEE conference on Supercomputing* , 2005.

[14] E. N. Elnozahy, et al. "The performance of consistent checkpointing," *11th Symposium on Reliable Distributed Systems*, Oct 1992.

[15] J. S. Plank, K. Li, and M. A. Puening. "Diskless Checkpointing", *IEEE Transactions on Parallel and Distributed Systems*, 9(10), October, 1998.

[16] K. Li, J. Naughton, J. S. Plank: Low-Latency, Concurrent Checkpointing for Parallel Programs. *IEEE Trans. Parallel Distrib. Syst.* 5(8), 1994.

[17] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, and N. Hutchinson, "Remus: high availability via asynchronous virtual machine replication," In *Proc. of the 5th USENIX NSDI*, 2008.

[18] Y. Tamura, et al. "Kermari: virtual machine synchronization for fault tolerance", *Proc. USENIX'08* Poster Session, 2008.

[19] Troy D. Hanson, http://uthash.sourceforge.net/

[20] http://www.linux-kvm.org/page/Qcow2

[21] https://btrfs.wiki.kernel.org/index.php/Main_Page

[22] http://en.wikipedia.org/wiki/ZFS