Checkpointing Virtual Machines Against Transient Errors

Long Wang, Zbigniew Kalbarczyk, Ravishankar K. Iyer

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801
{longwang, kalbarcz, rkiyer}@illinois.edu

Arun Iyengar

IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
aruni@us.ibm.com

Abstract—**This paper proposes VM-µCheckpoint, a lightweight software mechanism for high-frequency checkpointing and rapid recovery of virtual machines. VM-µCheckpoint minimizes checkpoint overhead and speeds up recovery by saving incremental checkpoints in volatile memory and by employing copy-on-write, dirty-page prediction, and in-place recovery. In our approach, knowledge of fault/error latency is used to explicitly address checkpoint corruption, a critical problem, especially when checkpoint frequency is high. We designed and implemented VM-µCheckpoint in the Xen VMM. The evaluation results demonstrate that VM-µCheckpoint incurs an average of 6.3% execution-time overhead for 50ms checkpoint intervals when executing the SPEC CINT 2006 benchmark.**

Keywords-**checkpoint; virtual machine; error latency; high frequency; checkpoint corruption**

I. INTRODUCATION

Virtual machines (VMs), also called *guest systems*, are frequently deployed to host a variety of IT services, such as web services, virtual desktops, and databases. To ensure continuous service availability, these systems must be capable of tolerating runtime errors. Checkpoint and rollback techniques can be applied to enhance VM availability.

Virtual machine monitors (VMMs), such as VMware and Xen, provide mechanisms a) to save a VM state (by stopping the VM and dumping the execution state into persistent storage) and b) to migrate the VM to a remote node (e.g., [2]). Most existing VM checkpoint techniques [1][8][4] exploit these two mechanisms. For example, CEVM [1] and VNsnap [8] first use live migration to create a copy of the protected VM in memory and then dump the replica to disk offline.

This paper presents the description, implementation, and experimental assessment of VM-µCheckpoint, a VM checkpointing framework to protect both the guest OS and applications against runtime errors. Advantages of using VM-µCheckpoint include:

(i) *Small overhead* compared with the VM replica-based failover approach. This is achieved by using in-memory checkpoint and in-place recovery of VMs, i.e., recovery of a failed VM in its current context. No such checkpoint work has been done in the context of virtual environments.

(ii) *Alleviation of checkpoint corruption due to error-detection latency* by taking advantage of knowledge of error detection latency. Using knowledge of fault/error latency for explicitly handling checkpoint corruption is a novel solution to this important problem.

(iii) *High checkpointing frequency*—tens of checkpoints per second—which reduces the size of each increment when taking a checkpoint.

(iv) *Rapid recovery*—within one second— as compared to the stop-and-dump approach provided by VMMs.

As a result, checkpointing during the normal system operation and recovery in response to a guest VM or application failure are completely transparent to the client, i.e., the client does not see a service interrupt.

Traditional checkpointing techniques save checkpoints on disk in order to tolerate permanent failures. Several VM checkpointing techniques, including Remus [4], save checkpoints in the memory of another node. In this study, we propose saving checkpoints in the memory of the same node.

VM-µCheckpoint is designed as a complementary approach to disk-based VM checkpointing rather than its replacement. By providing a rapid recovery, VM-µCheckpoint significantly reduces the failure rate of VMs due to transient errors [1]. We created analytical and probabilistic models (not presented in this paper due to space limitations) to assess the availability improvement when using VM-µCheckpoint. The model-based analysis can be found in [15].

The major contributions of this paper are:

- Design, implementation, and integration of VM-µCheckpoint in Xen VMM. The VM-µCheckpoint implementation does not introduce any changes to the guest VMs or applications. Copy-on-Write (CoW), dirty-page prediction, and pre-saving algorithms are designed and implemented to achieve high performance. The key innovations in the proposed algorithms are (i) the use of dirty-page prediction and pre-saving, which are not

[1] At the same time, the checkpoint kept by VM-µCheckpoint can be dumped to disk at a sufficiently infrequent rate to minimize overhead. This means that in the event of a node fails, the VM and the jobs in the node can be restarted from the last valid disk checkpoint.

supported by the default Xen's CoW mechanism, and (ii) a mechanism to overcome the inefficiency of Xen's CoW in supporting high-frequency periodic checkpointing.

- Use of knowledge of the error detection latency to derive checkpoint intervals that minimize the possibility of checkpoint corruption. Our model-based analysis (in [15]) shows that the availability of guest VMs and applications is improved from 99% to 99.98%, assuming a highly reliable hypervisor (MTTF of 625 days in our study).

An experimental assessment of VM-μCheckpoint using (i) *SPEC benchmark programs.* The evaluation shows that VM-μCheckpoint incurs an average of 6.3% overhead for SPEC benchmark programs with 50 ms checkpoint intervals[2]. This choice represents a design tradeoff between keeping checkpoint size small and minimizing chances of checkpoint corruption due to latent errors. (ii) *Apache server, an example network applicatio*n. The results show 17.5% throughput reduction when taking a checkpoint every 50ms. This overhead is significantly lower than the existing VM checkpointing techniques, e.g., Remus [4].

II. RELATED WORK

Checkpoint and rollback techniques have been extensively studied in the literature. Checkpoints can be taken in different levels (application, runtime library, compiler, operating system, virtual machine, or hardware). Here we focus on checkpoint techniques in the virtual machine level.

Most existing VM checkpoint/replication techniques are based on live migration of VMs (e.g., VMWare VMotion [11] and Xen Live Migration [2]), which continually transmit dirty pages of a VM from a source node to a destination node. These techniques exploit the live migration mechanism for the purposes of VM checkpointing, VM rejuvenation, load-balancing, and fast VM forking.

CEVM [1], VNsnap [8], and VM Snapshots [3] are disk-based VM checkpointing techniques. These techniques employ VM live migration or CoW to create a replica image of a VM with low downtime incurred; then they write the image to disk offline. An ongoing project on VM checkpointing [3] tries to provide a generic API in Xen products for saving a VM snapshot to disk on demand. Basically, the VM memory is scanned and saved to files while the VM runs.

VM-μCheckpoint differs from disk-based VM checkpointing in that we aim to i) provide high-frequency checkpointing and rapid recovery of VMs, which allows VM failures to be masked to clients, and ii) propose a mechanism to alleviate checkpoint corruption in high-frequency checkpointing, corruption that has significant impacts on service availability. Disk-based VM checkpointing is too costly and is unable to keep up with the high frequency of rapid checkpointing (tens per second).

The existing approach closest to our work is Remus [4], which maintains a backup VM on a separate physical node by periodically transmitting the VM's dirty pages to the backup. Similarly to VM-μCheckpoint, Remus is a method of high-frequency VM checkpointing and failover. But VM-μCheckpoint considers error behavior and emphasizes reliability/availability improvement, while Remus focuses on migration overhead and lacks a study of error behavior or reliability/availability. Because checkpoint corruption is not handled in Remus (fail-stop errors are assumed), our technique is preferable at improving service availability.

Another relevant work is FTC-Charm++ [16], which checkpoints MPI programs. FTC-Charm++ uses in-memory checkpointing, keeping two identical checkpoint copies in the memories of two nodes in order to tolerate single node failure. However, Charm++ is not automated; the programmer must specify what and when to checkpoint. Charm++ is not aimed at virtual environments, and importantly, checkpoint corruption is not considered.

III. VM-μCHECKPOINT OVERVIEW

Figure 1 illustrates how VM-μCheckpoint is deployed to protect virtual machines (guest systems) running on top of a hypervisor. The *protected VM* in the figure is the guest system to be checkpointed. Another guest system on the same physical machine is selected to be the *checkpointing VM* where VM-μCheckpoint is installed. The checkpointing VM can be a guest system dedicated to the checkpointing service; it need not be a privileged guest system, such as Dom0 in Xen. The hypervisor and the kernel of the checkpointing VM are instrumented to support checkpointing and recovery.

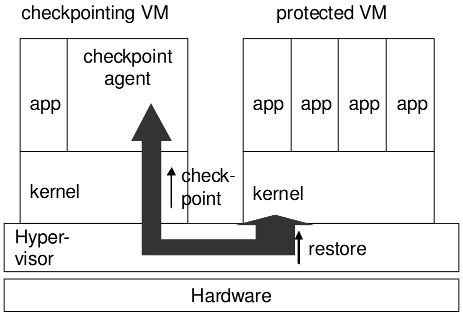

Figure 1: Deployment of VM-μCheckpoint

The starting point of the proposed approach is the observation that *short-latency errors are dominant.* This is demonstrated by several previous fault injection experiments, including recent work on error injection into Linux kernel [1] that shows about 95% of crashes occur within 100 million CPU cycles (or within 50ms on a 2GHz processor) after an error occurrence. Fault injections into processor micro-architecture [9] also show small error

[2] The recent work of fault-injection into Linux kernel [1] shows that, about 95% of crashes occur within 100 million CPU cycles (or within 50ms on a 2GHz processor) after an error occurrence. We select a checkpoint interval of 50ms in experiments to cover the latency for 95% of errors.

latencies, and state-of-the-art error detection techniques (e.g., [13][12][14]) have helped to limit error latency to low values.

At the same time, it has been shown in many studies that *a vast majority of failures are transient* (up to 95%). Furthermore, our experiments on IBM Power series systems also demonstrate that, in VM environments, errors impacting the hypervisor rarely affect more than a single guest VM [7].

Latency-driven selection of the checkpoint interval. We define a parameter T_B as a user-specified bound on error latency. By setting the checkpoint interval greater than an acceptable latency bound (e.g., 95th percentile) we effectively bound the probability of a latent/undetected error affecting the checkpoint to be small (in the best case, <5%). In addition, by always holding two checkpoints in sequence and, on detecting an error, reverting to the earlier checkpoint, we further reduce the probability of checkpoint corruption. This is primarily due to two factors: (i) the earlier checkpoint is taken at a time that is at least T_B in the past, and (ii) since by choice (per the latency distribution) the chance of an error getting detected (or causing a failure) in an interval T_B is greater than 95%, we can have confidence that the checkpoint is error free if no error has been detected.

A user-level process in the checkpointing VM, referred to as the *checkpoint agent* in Figure 1, takes a checkpoint of the protected VM periodically, at intervals of T_{ck} (where $T_{ck} > T_B$), and stores the checkpoint in the checkpointing VM. Since at each checkpoint our copy-on-write (CoW) implementation indentifies and stores the needed state information, the checkpoint agent stores only a small fraction of the protected VM state rather than the entire system image. This approach allows the checkpoint agent to store checkpoints of multiple guest systems on the same physical machine using a small amount of memory.

IV. CHECKPOINTING ALGORITHMS

At the beginning of a checkpoint interval, register states in the protected VM are saved in the checkpoint, and all memory pages are set as read-only. From that point on, any write to a read-only page triggers a page fault, the original data of the page are copied into the checkpoint kept in the checkpoint agent memory, and the stored memory page is set as writable. Consequently, the checkpoint consists of pre-write data of pages updated within a given checkpoint interval.

As mentioned previously, the two most recent checkpoints are kept at all times. The following actions are performed when an error in the protected VM is detected or causes a failure: i) the most recent checkpoint is copied back into the current state of the protected VM, and ii) the earlier checkpoint of the two kept ones is copied into the system memory to restore the correct state. This method restores the system to the state when the earlier checkpoint, called the *committed* checkpoint, was taken, a state that is unlikely to have been corrupted. Note that a checkpoint is a pre-write state of modified pages, i.e., a state at the beginning of the corresponding checkpoint interval. Therefore, corruption of a later checkpoint does not cause the rollback to fail. Any corrupted page in the later checkpoint is overwritten by the correct state preserved in the earlier checkpoint. A detailed explanation of this method follows.

Figure 2 illustrates timelines of this checkpointing/ recovery scheme. Two complete checkpoint intervals, $[t_0, t_1)$ and $[t_1, t_2)$, are shown in Figure 2. The horizontal axis at the top of the figure represents error-free execution of the protected VM, while the horizontal axis at the bottom represents execution when an error occurs at t_{f_s}. The error is detected (or the application/system fails) at t_{f_d} (the error latency $T_f = t_{f\_d} - t_{f\_s} \le T_B$). At t_{f_d}, the two most recent checkpoints are those taken at t_0 and t_1. We first restore the data preserved during the time interval $[t_1, t_{f\_d})$ into the protected VM and then restore the data preserved during $[t_0, t_1)$ to roll back the system to the state at time t_0.

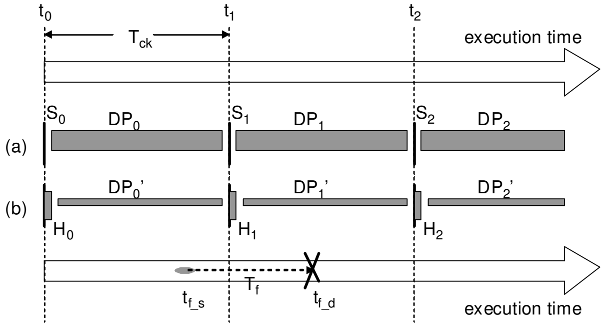

Figure 2: Timelines for two checkpoint strategies: (a) CoW-B and (b) CoW-P

In the algorithm described above called CoW Basic (CoW-B), setting all memory pages as read-only at the beginning of a checkpoint interval potentially results in a large number of page faults and a significant performance overhead. An optimized version of the basic algorithm called CoW Pre-saving (CoW-P) is designed to reduce the resulting page faults (checkpoint-caused page faults are reduced by 75% when the checkpoint interval is 50ms in our experiments, see Section V.B).

The CoW-B algorithm. This algorithm is depicted as the timeline (a) in Figure 2. Here are the notations used in our discussion:

t_i Beginning time of the i^{th} checkpoint interval
S_i State of the protected VM at time t_i.
DP_i Dirty Pages – data of the memory pages preserved by VM-μCheckpoint's mechanism during $[t_i, t_{i+1}]$.
S_t State of the protected VM at any time t ($t \in [t_i, t_{i+1}]$).
$DP_i(t)$ Data of the memory pages preserved by VM-μCheckpoint's mechanism during $[t_i, t]$ for any time t ($t \in [t_i, t_{i+1}]$).

The following operation reflects the inherent relationship between S_i, S_t, and $DP_i(t)$:

$$S_i = restore(S_t, DP_i(t)), \quad (1)$$

where $restore(S_t, DP_i(t))$ denotes an operation of copying the data preserved in $DP_i(t)$ into their corresponding memory pages in S_t to restore the system to state S_i.

In the example error scenario shown in Figure 2 $t_1 \le t_{f\_d} \le t_2$, $T_f \le T_B \le T_{ck}$, and S_0 is the last committed checkpoint. Applying the operation (1) twice, we can derive the expression that depicts restoration of S_0:

$$\begin{aligned} S_0 &= restore(S_1, DP_0(t_1)) \\ &= restore(restore(S_f, DP_1(t_{f\_d})), DP_0), \end{aligned} \quad (2)$$

where $S_{t_1} = S_1$, $DP_0(t_1) = DP_0$, and S_f denotes the system state at t_{f_d}. At the restoration time t_{f_d}, S_f, $DP_1(t_{f\_d})$ and DP_0 are all available, and we can restore the memory state of the protected VM into S_0. After restoration, neither $DP_1(t_{f\_d})$ nor DP_0 is valid any more, as the system is now in state S_0. They are discarded after the restoration.

The CoW-P algorithm. Figure 2 (b) shows the timeline of the CoW-P. This algorithm reduces the number of page faults by predicting the pages to be updated in the upcoming checkpoint interval. The predicted pages are then pre-saved in the checkpoint when this interval begins (H_0, H_1, and H_2 are the pre-saved pages in Figure 2 (b)). These pre-saved pages are marked as writable and do not raise page faults.

The typical checkpoint intervals (selected in our approach) range from tens of milliseconds to several seconds. Due to the space and time locality of memory accesses, pages that were updated recently tend to be updated again in the near future. Therefore, the pages dirtied in the previous checkpoint interval are used to predict the pages to be updated in the upcoming interval.

Specifically, a page table supported by current-generation processors maintains an entry for each memory page. The page entry has two control bits—the write permission bit and the dirty bit—which are leveraged for our prediction. (We manipulate the shadow copy of this page table maintained by the VMM, rather than the page table in the guest operating system. In this way, the guest system's use of its page table is not interfered with.) The write-permission bit controls whether the page is writable, and the dirty bit shows whether the page has been updated since the dirty bit was last cleared. At the beginning of a checkpoint interval, both of the bits for non-dirty pages are cleared (i.e., set as read-only and not dirty). While the pages dirtied in the previous checkpoint interval are saved in checkpoint, their write permission bits are set to allow writes to them, and their dirty bits are cleared to enable tracking of whether they will be updated during the upcoming interval. If a page dirtied in the previous interval is not updated during the upcoming interval, then next time (i.e., after this upcoming interval), this page is not pre-saved and is set as read-only.

Discussion. Both error latency and checkpoint overhead are considered when selecting a checkpoint interval T_{ck}. Checkpointing with a larger interval incurs smaller overhead but causes a longer output delay and a larger checkpoint size (because output is held until the corresponding checkpoint is committed). Hence, there is a trade-off in T_{ck} selection. For example, if a small output delay is desired, a small T_{ck} is preferred, as long as T_{ck} is larger than the selected T_B and the checkpoint overhead is acceptable.

Error detection latency depends on error detection techniques (e.g., [13][12][14]). Note that error detection is not in the scope of this paper. In order to obtain the distribution of error detection latency, we inject errors into a target system and measure the latency from error activation to the occurrence of system or application failure. We conducted an analytical model to study the impacts of T_{ck} on checkpoint corruption and system availability [15]. Based on the analytical model and the obtained error latency distribution, we can select the proper T_{ck}.

VM-μCheckpoint recovers a guest system and applications in the system from any transient hardware error or transient software error, including both application and system errors. Transient hardware errors include those occurring in the processor (functional units, registers, caches, buses, and control logics) and in memory due to events such as radiation or current disturbances. Transient software errors, or Heisenbugs [5], include exceptional conditions (e.g., a counter overflow and an interrupt arrival with a bad timing), occasional device driver faults, race conditions, and corrupted parameter or data due to bad transmission. Note that transient failures of the checkpointing VM are handled by an immediate restart of the failed checkpointing VM.

VM-μCheckpoint cannot guarantee recovery if either of the following holds: (i) *Checkpoint corruption*. There is a small but finite probability of checkpoint corruption. In this case, VM-μCheckpoint aborts recovery and restarts the VM and the interrupted jobs. (ii) *Failure of the hypervisor due to a transient fault.* In this case, we first restart the hypervisor and restart all jobs executing prior to the failure. If this is unsuccessful, the system rolls over to an adjacent physical node and restarts.

While this paper focuses on the design, implementation, and analysis of the memory state checkpointing in VM-μCheckpoint, I/O checkpointing can be dealt with by adapting the output-commit mechanism applied in [4][10].

V. EXPERIMENTAL EVALUATION

Fully working prototype of VM-μCheckpoint is implemented in Xen VMM. The source codes of the Xen hypervisor and the checkpointing VM are instrumented while there is no change to the protected VM[3]. Details of algorithms and the overall implementation can be found in [15].

The testbed consists of a physical machine with an AMD Athlon 2800 (1.8G Hz) processor and 1.5GB memory.

[3] The I/O recovery mechanism is not implemented in the current prototype.

(Because hardware is virtualized by Xen, VM-ucheckpoint should work on SMP platforms as well, though we have not tried such experiments.) There are two guest systems (Linux 2.6.18) running on top of Xen 3.3.1 in the testbed. The Dom0 is selected as the checkpointing VM, and the other guest system, a DomU, is the protected VM. 512MB and 1GB memory are assigned to the checkpointing VM and the protected VM, respectively. We use two VMs in our experiments to measure performance overhead accurately in a relatively simple deployment.

A. *Experiment Setup*

Workload of SPEC CINT 2006. SPEC2006 is widely accepted in industry for performance benchmarking. For example, *milc* is a scientific application used for millions of node hours at DOE and NSF supercomputer centers, and *gcc* is a full-featured compiler with 365k lines of source code. A set of SPEC CINT 2006 benchmark programs are executed in the protected guest system with VM-μCheckpoint deployed. A suite of experiments are conducted involving each of these benchmark programs: (i) a baseline case with no checkpoint; (ii) CoW-B algorithm deployed with the four checkpoint intervals of 1000ms, 600ms, 200ms, and 50ms; and (iii) CoW-P algorithm deployed with the same four intervals. A given program executes with the same input across all experiments.

Program execution times are measured, and normalized execution times are illustrated in Figure 3. (While 95% confidence intervals of execution times are computed, these are not presented to avoid cluttering in Figure 3) Normalized execution time is computed by dividing program execution time by the execution time in the corresponding baseline case.

Workload of a web server. We conduct experiments to study how VM-μCheckpoint affects Apache web server throughput when the web server runs on the protected guest system. Web clients reside on three physical machines with each machine hosting 50 clients. These clients request the same load of web pages, one request immediately after another, from the server simultaneously via a 100Mbps LAN. The output-commit mechanism is disabled in these experiments, and consequently, we compare our performance with Remus results when the output commit is also disabled.

Figure 4 illustrates the measured server throughput as a function of checkpoint intervals. The same load of web requests is processed in these experiments. The percentages indicated along the data points on the graph represent the ratio between the throughput measured with the checkpoint deployed and the throughput when checkpoint is not deployed.

B. *Results*

The major findings in our experiments are summarized below:

a) VM-μCheckpoint achieves much better performance than existing migration-based VM checkpointing. For a workload of SPEC CINT 2006 benchmark and a checkpoint frequency of 20 times per second (T_{ck}=50ms), an average of 6.3% overhead is incurred when CoW-P is deployed. With the same checkpoint algorithm and checkpoint frequency, Apache server throughput is reduced by 17.5%. In contrast, Remus [4], a migration-based VM replication/checkpoint technique, reports approximately 50% overhead in their experiments for the same checkpoint frequency.

VM-μCheckpoint's performance varies with the workload. Generally, workload applications with larger data set have more overhead. For example, *milc* is a scientific application dealing with larger data set while *gcc* deals with smaller data set. As a result, VM-ucheckpoint incurs more overhead for milc workload than that for gcc (Figure 3).

b) The CoW-P algorithm performs much better than CoW-B. With CoW-P deployed with 50ms checkpoint intervals, Apache throughput is 82.5% of the baseline performance, which is larger than the 74.3% when CoW-B is deployed. We also measure the number of page faults and the checkpoint size (not shown due to space limitations). Dirty page prediction and pre-saving effectively reduce page faults by 75% when the checkpoint interval is 50ms (details can be found in [15]).

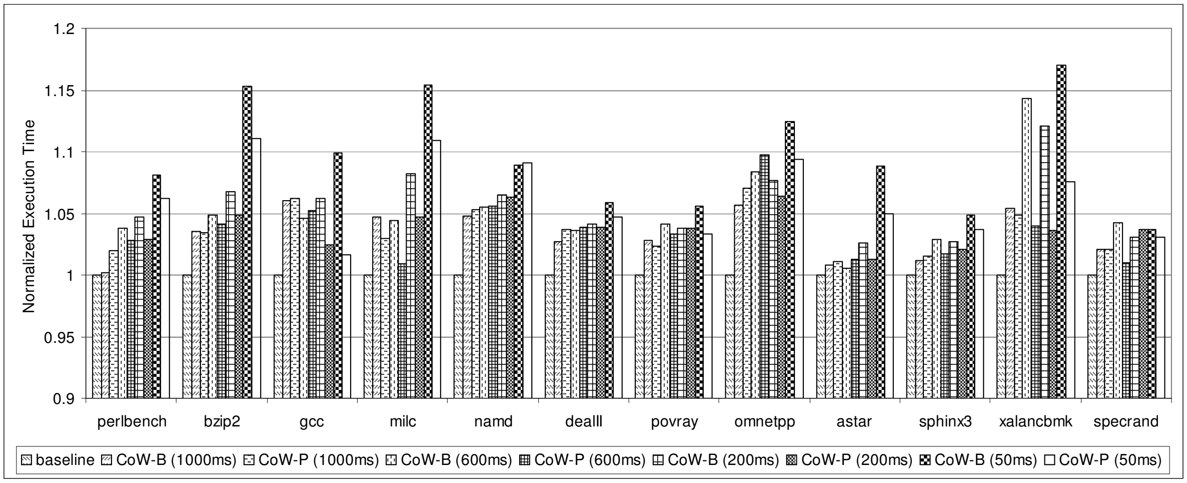

Figure 3: Experiment results in terms of execution time of SPEC CINT 2006

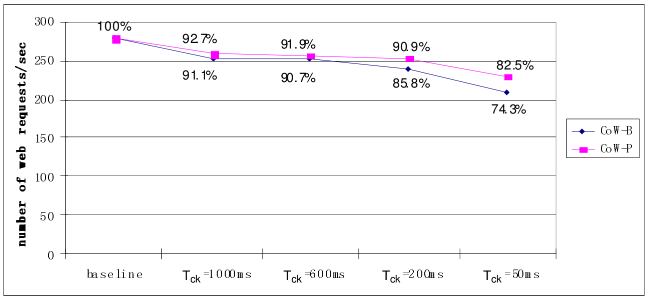

Figure 4: Experiment results in terms of Apache web server throughput

c) Average checkpoint sizes are very small, less than 2% of the size of the entire system state, when the checkpoint interval is 50ms. With CoW-P deployed at a checkpoint interval of 50ms, the average checkpoint size is 654.5 memory pages or 2.6MB, while the size of the entire system state during the experiment is up to 206MB. The observed maximum checkpoint size is less than 8MB, less than 4% of the entire system state size. When the checkpoint interval is increased to 1000ms, most checkpoints are less than 10,000 pages, and the average size is 2162.4 pages (8.6MB, or 4.2% of the entire state).

C. Virtual Machine Recovery

Experiments are conducted a) to test the ability of the proposed technique to correctly recover a virtual machine and b) to measure the recovery time. In this analysis, we consider application failures as a means to error detection. For this purpose, a small custom program is developed that causes a segmentation failure after executing for a while. The instrumented hypervisor-level exception handler then issues an "error detected" request via a divided-by-zero exception.

The SPEC CINT 2006 benchmark programs run as the workload on the protected virtual machine. The custom program is launched to generate a failure while the workload is running. The protected virtual machine is then rolled back to the last committed checkpoint. The measured recovery time ranges from 144ms to 1017ms with the average of 639.4ms (the 95% confidence interval is 639.4ms ± 193.1ms) in our experiments.

VI. CONCLUSIONS

This paper proposes VM-μCheckpoint, a lightweight VM checkpointing technique that a) addresses the problem of checkpoint corruption in high-frequency checkpointing and b) minimizes overhead by placing checkpoints in memory and performing in-place recovery in a virtual environment. We show that it is important to take into account the expected times for errors to manifest themselves in determining checkpoint intervals. VM-μCheckpoint was implemented in the Xen VMM. Experimental results showed that the proposed technique achieves much better performance than existing techniques based on VM live migration. Overhead is low with VM-μCheckpoint: the average program execution time overhead for the SPEC CINT 2006 benchmark when VM-μCheckpoint is deployed at a checkpoint frequency of 20 times per second is 6.3%. Moreover, checkpoint size is small: an average of less than 2% of the entire system state in our experiments when the CoW-P algorithm is applied with 50ms checkpoint intervals.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants, CNS-05-24695, CNS-05-51665, and ACI-0121658 ITR/AP, the Gigascale Systems Research Center (GSRC/MARCO), IBM Corporation, and Boeing Corporation.

REFERENCES

[1] K. Chanchio et al., An Efficient Virtual Machine Checkpointing Mechanism for Hypervisor-based HPC Systems, *High Availability and Performance Computing Workshop*, 2008

[2] C. Clark et al., Live Migration of Virtual Machines. *Proc. of Networked Systems Design and Implementation,* 2005.

[3] P. Colp, VM Snapshots, *Xen Summit*, 2009, *http://www.xen.org/files/xensummit_oracle09/VMSnapshots.pdf*

[4] B. Cully et al., Remus: High Availability via Asynchronous Virtual Machine Replication, Proc. of *Networked Systems Design and Implementation*, 2008.

[5] J. Gray, Why Do Computers Stop and What Can Be Done About It? *Proc. of Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[6] W. Gu et al., Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors, *Proc of Conference on Dependable System and Networks*, 2004.

[7] W. Gu et al., Fault Inject Based Study of Fault Resilience of Hypervisor, University of Illinois Urbana-Champaign Report 2007.

[8] A. Kangarlou et al., VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Down*time, Proc. of Conference Dependable Systems and Networks*, 2009.

[9] M. Li et al., Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design, *Architectural Support for Programming Languages and Operating Systems*, 2008.

[10] J. Nakano et al., ReVive I/O: Efficient Handling of I/O in Highly Available Rollback-Recovery servers, *HPCA* 2006.

[11] M. Nelson et al., Fast Transparent Migration for Virtual Machines, *USENIX* 2005.

[12] K. Pattabiraman et al., Automated Derivation of Application-aware Error Detectors Using Static Analysis. Proc. of *International On-Line Testing Symposium*, 2007.

[13] G. Reis et al., SWIFT: Software Implemented Fault Tolerance, *Proc. of Symposium on Code Generation and Optimization*, 2005.

[14] J. Smolens et al., Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth, *Architectural Support for Programming Languages and Operating Systems*, 2004.

[15] L. Wang, et al., Checkpointing Virtual Machines Against Transient Errors: Design, Modeling, and Assessment. *https://netfiles.uiuc.edu/longwang/www/VM-ucheckpoint.pdf.*

[16] G. Zheng et al., FTC-Charm++: An In-Memory Checkpoint-based Fault Tolerant Runtime for Charm++ and MPI, *IEEE International Proc. of Conference on Cluster Computing,* 2004.