VM-μCheckpoint: Design, Modeling, and Assessment of Lightweight In-Memory VM Checkpointing

Long Wang, *Member, IEEE*, Zbigniew Kalbarczyk, *Member, IEEE*, Ravishankar K. Iyer, *Fellow, IEEE*, and Arun Iyengar, *Fellow, IEEE*

Abstract— Checkpointing and rollback techniques enhance reliability and availability of virtual machines and their hosted IT services. This paper proposes VM-μCheckpoint, a light-weight pure-software mechanism for high-frequency checkpointing and rapid recovery for VMs. Compared with existing techniques of VM checkpointing, VM-μCheckpoint tries to minimize checkpoint overhead and speed up recovery by means of copy-on-write, dirty-page prediction and in-place recovery, as well as saving incremental checkpoints in volatile memory. Moreover, VM-μCheckpoint deals with the issue that latency in error detection potentially results in corrupted checkpoints, particularly when checkpointing frequency is high. We also constructed Markov models to study the availability improvements provided by VM-μCheckpoint (from 99% to 99.98% on reasonably reliable hypervisors). We designed and implemented VM-μCheckpoint in the Xen VMM. The evaluation results demonstrate that VM-μCheckpoint incurs an average of 6.3% overhead (in terms of program execution time) for 50ms checkpoint intervals when executing the SPEC CINT 2006 benchmark. Error injection experiments demonstrate that VM-μCheckpoint, combined with error detection techniques in RMK, provides high coverage of recovery.

Index Terms—checkpoint corruption, checkpoint model, error latency, incremental checkpoint, high-frequency checkpoint, transient error

1 Introduction

Virtual machines (VMs) are popularly deployed to host a variety of IT services. 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) like VMware, Xen, and KVM, provide mechanisms to save a VM state (i.e., stop the VM and dump the execution state in persistent storage) and migrate the VM to a remote node (e.g., [6]). Most of the existing VM checkpoint techniques [17][23][7] exploit these two mechanisms. For example, CEVM [17] and VNsnap [23] first use live migration to create a replica of the protected VM in memory and then dump the replica to disk offline.

Traditional checkpointing techniques save checkpoints on disk to tolerate failures. Several VM checkpointing techniques, including Remus [7], save checkpoints in the memory of another node. Here, we propose saving checkpoints in the memory of the same node. Our model study (Section 5) shows that VM availability is largely increased with checkpoints in the same node's memory on reasonably reliable hypervisors.

Specifically, this paper presents the design, model study, and experimental assessment of VM-μCheckpoint, a VM checkpointing framework to protect both VMs and applications in the VMs against runtime errors. When an error occurs silently in memory (e.g. due to radiation), ECC can correct this error. This paper targets the types of errors not detected/corrected by ECC memory, including hardware transient errors occurring to the registers, caches, buses, control logics, as well as software transient errors.

VM-μCheckpoint supports in-place recovery of failed VMs using in-memory checkpointing. Advantages to using VM-μCheckpoint include (i) *small overhead* as compared with the replica-based failover approach, (ii) *high checkpointing frequency* (tens of checkpoints per second), which reduces the size of each increment when taking a checkpoint, (iii) *addressing checkpoint corruption due to latency of error detection* by modeling error latency characteristics and dealing with checkpoint corruption properly (checkpoint corruption is not negligible in high-frequency checkpointing, e.g. checkpoint every 50ms), and (iv) *rapid recovery* (within one second) as compared with the stop-and-dump approach (provided by VMMs as a basic capability).

As a result, checkpointing (during the normal system operation) and the recovery (in response to a failure of a VM and/or application) are almost completely transparent, i.e., the client does not see a service interruption.

VM-μCheckpoint significantly improves the availa-

- *Long Wang is with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598. E-mail: wanglo@us.ibm.com.*
- *Zbigniew Kalbarczyk is with the University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: kalbarcz@illinois.edu.*
- *Ravishankar K. Iyer is with the University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: rkiyer@illinois.edu.*
- *Arun Iyengar is with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598. E-mail: aruni@us.ibm.com.*

bility of VMs against transient failures by providing rapid recovery (detailed discussion of these failures given in Section 5). Moreover, VM-μCheckpoint is designed as a complementary approach to disk-based VM checkpointing rather than its replacement. The checkpoint kept by VM-μCheckpoint can be dumped to disk (to tolerate node failure) at a sufficiently infrequent rate to minimize overhead. Such an extension of VM-μCheckpoint to support disk-based checkpoint is described in Section 3.1.

The major contributions of this paper are:

- Design of VM-μCheckpoint and its implementation in Xen VMM. The VM-μCheckpoint takes in-memory incremental checkpoints (i.e., each increment includes only memory pages modified during the current checkpoint interval) and employs in-place restoration (i.e., recovering a failed VM into its current context) to minimize overhead. Copy-on-Write mechanism and dirty-page prediction are exploited to further minimize checkpoint overhead.
- Model study of VM-μCheckpoint's effectiveness in enhancing service availability: i) A data-driven analytical model is constructed and used to evaluate the probability of checkpoint corruption. The analysis shows that VM-μCheckpoint with proper parameter settings can significantly reduce the probability of recovery failures due to checkpoint corruption, from a high of 31.7% to a low of less than 2% as compared to current methods. ii) A comprehensive Markov model is constructed to evaluate the availability improvement when VM-μCheckpoint is used. The model shows the availability of VMs and applications increases from 99% to 99.98% on reasonably reliable hypervisors (these hypervisors have an MTTF of 1.7 years or larger).
- An experimental assessment of VM-μCheckpoint runtime performance shows that VM-μCheckpoint incurs low overhead with typical checkpoint intervals (an average of 6.3% overhead for SPEC benchmark programs with 50 ms checkpoint intervals[1]).
- Error injection experiments demonstrate that VM-μCheckpoint, deployed in the RMK framework [31] to leverage a variety of existing error detection techniques, has high coverage of error recovery (100% for system crashes and corrupted data, as well as 80% for system hangs in our experiments).

2 Architecture

Our light-weighted checkpoint approach, VM-μCheckpoint, is based on a well-known observation that short-latency errors are dominant. Fault injection experiments such as those reported in [1] show that 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 [26] also show small error latencies. Moreover, state-of-the-art error detection techniques (e.g., [3][2][13]) help to limit the error latency to low value.

Figure 1 depicts the deployment of VM-μCheckpoint for recovering VMs from errors. We leverage the RMK framework [31] for the deployment because RMK provides a modular platform that allows for flexible installation of error detection and recovery mechanisms.

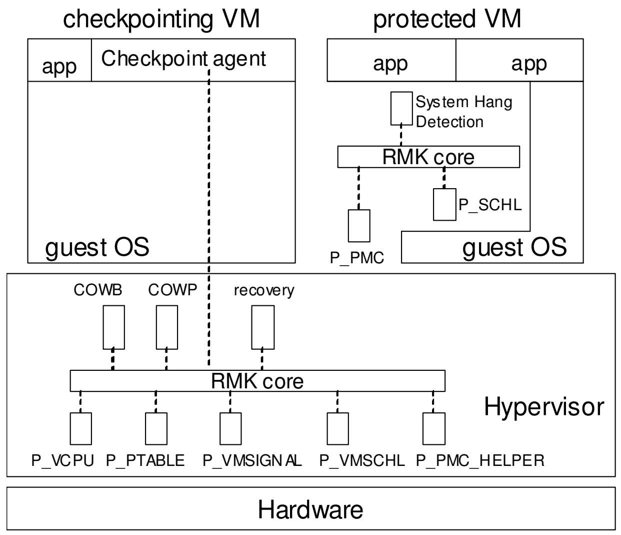

Figure 1: Error Detection/Recovery in RMK

Background of the RMK framework. The RMK was proposed as a device driver in standalone systems. It exploits processor-level features (debugging and monitoring facilities available in the current generations of processors), and OS-exported interfaces to define a set of basic services. These basic services are called *RMK pins*, which are analogous to hardware pins in providing clearly defined functionalities and inputs/outputs. The pins are employed to support mechanisms of error detection and recovery, referred to as *RMK modules*. RMK pins and RMK modules are communicated in a *publish-subscribe* fashion, i.e., RMK pins publish pin-specific events to the RMK framework and RMK modules subscribe certain events from the RMK framework.

Figure 1 illustrates how we extend the basic RMK scheme in [31] to support a virtualized environment and to accommodate the VM-μCheckpoint mechanism. Similar to the RMK in a standalone system, the RMK in the protected VM is installed as a device driver; no kernel source recompilation is needed. In the hypervisor (Xen in our prototype), we implemented a new hypercall that encapsulates the hypervisor-level RMK (a hypercall is like a system call for a hypervisor). The hypervisor source code is instrumented and recompiled for this purpose.

VM-μCheckpoint consists of three RMK modules in the deployment (Figure 1): *COWB, COWP,* and *recovery*. They implement the two checkpoint algorithms and the recovery algorithm of VM-μCheckpoint. Details of the algorithms are presented in Section 3 below. Certain RMK pins support VM-μCheckpoint. For example, the RMK pins P_VCPU and P_PTABLE wrap

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

the hypervisor functionalities of manipulating virtual CPUs and the shadow page table, respectively; P_VMSIGNAL intercepts received signals and P_VMSCHL intercepts scheduling of VMs by the hypervisor for detection of VM failures. These RMK pins are implemented as instrumentation to the hypervisor kernel.

By deploying our checkpoint scheme in RMK, we can take advantage of a number of error detection techniques provided in RMK, including crash detection, system hang detection, etc. When an error of the protected VM is detected the VM is recovered by the VM-μCheckpoint mechanism.

3 Checkpoint Algorithms

A user-level process in the checkpointing VM, named *checkpoint agent* in Figure 1, takes a checkpoint of the protected VM periodically at an interval of Tck. The checkpoint is stored in the checkpointing VM. At each checkpoint the Copy-on-write (CoW) mechanism is invoked to identify and store the needed state information. As a result, the checkpoint agent stores only a small fraction of the protected VM state rather than the entire system image, and checkpoints of multiple protected VMs on the same physical machine can be stored in the checkpointing VM.

At the beginning of a checkpoint interval, all memory pages of the protected VM are set as read-only. From that point on, any write to a read-only page triggers a page fault, original data of the page is copied into the checkpoint kept in the checkpoint agent memory, and this memory page is set as writable. *So the checkpoint therefore consists of prior-to-update data of those pages updated within the interval.* Any transient errors that originate as incorrect values written to memory can be recovered by copying back the checkpoint data of the updated pages. Besides updated pages, new pages and deleted pages are handled accordingly to save the prior-change state of these pages.

It is possible that a checkpoint gets corrupted by an error before the error is detected. We keep two most recent checkpoints at any time (called *dual-checkpoint* scheme) for reducing the probability of a corrupted checkpoint failing error recovery. Our model (Section 4) shows that combining proper selection of T_{ck} and the dual-checkpoint scheme greatly reduces the probability of recovery failures due to checkpoint corruption.

The COWB algorithm. The basic algorithm of VM-μCheckpoint, COWB, is depicted as the timeline (a) in Figure 2. $[t_0, t_1)$ and $[t_1, t_2)$ are two checkpoint intervals. 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 at t_{f_d}. At t_{f_d} 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.

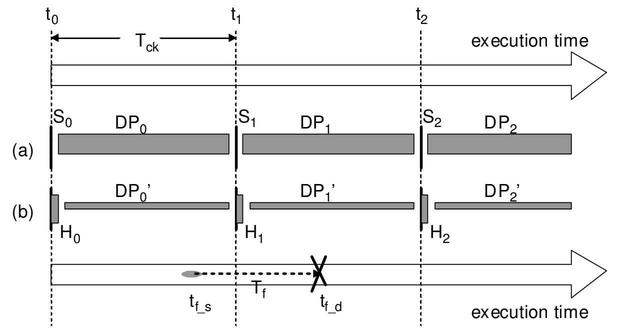

Figure 2: Timelines for two checkpoint strategies: (a) COWB and (b) COWP

We formalize the checkpoint problem and our algorithms using the following notations:

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 inherent relationship between S_i, S_t, and $DP_i(t)$:

$$S_i = restore(S_t, DP_i(t)), \tag{1}$$

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 Figure 2 we apply the operation (1) twice and have

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

where $S_{t_1} = S_1$, $DP_0(t_1) = DP_0$, and S_f denotes the system state at t_{f_d}. At t_{f_d} when error recovery occurs, 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 recovery.

The COWP algorithm. A large number of page faults are incurred in COWB because all memory pages are set as read-only at the beginning of each checkpoint interval. We design an optimized version of this basic algorithm, called Copy-on-Write Pre-saving (COWP), to reduce the number of page faults and corresponding performance overhead (checkpoint-caused page faults are reduced by 75% when the checkpoint interval is 50ms in our experiments, see Section 7.3).

Specifically, COWP predicts the pages to be updated in the upcoming checkpoint interval and pre-saves the predicted pages in the checkpoint when this interval begins. These pre-saved pages are marked as writable and do not raise page faults. Typical checkpoint intervals selected in our scheme range from tens of milliseconds to several seconds. Due to the space and time locality of memory accesses, the pages dirtied in the previous checkpoint interval are used as prediction of the pages to be updated in the upcoming interval. Two control bits —the write permission bit and the dirty bit— of each memory page entry maintained in current-generation processors are leveraged for implementing the COWP algorithm. Figure 2 (b) shows the timeline of the COWP.

More formally, let H_i denote data of the memory pages updated in the checkpoint interval $[t_{i-1}, t_i)$ (H_0 is obtained by profiling system execution before t_0), DP_i' denote data of the pages preserved by COWP during $[t_i, t_{i+1})$, and $DP_i'(t)$ be data of the pages preserved by COWP during $[t_i, t)$ for any $t \in [t_i, t_{i+1}]$. Then we use H_i as prediction of DP_i. Using the *restore()* operation defined in (1), we have:

$$S_i = restore(S_t, DP_i'(t) \cup H_i). \tag{3}$$

Due to the inaccuracy of dirty page prediction, Hi includes data of pages that are not updated in $[t_i, t]$. Similar to the discussion on COWB, the expression that represents restoration of S_0 is derived as:

$$\begin{aligned} S_0 &= restore(S_1, DP_0'(t_1) \cup H_0) \\ &= restore(restore(S_f, DP_1'(t_{f\_d}) \cup H_1), DP_0' \cup H_0), \end{aligned} \tag{4}$$

where $S_{t_1} = S_1$, $DP_0'(t_1) = DP_0'$.

3.1 Disk-based Checkpointing

In order to recover a VM against a permanent failure or a hypervisor failure, VM checkpoints can be saved in disks. Figure 3 illustrates the extension of VM-μCheckpoint to support disk-based VM checkpoint.

Specifically, in addition to the in-memory checkpointing described above, the checkpoint agent scans the protected VM and saves every memory page (i.e. the SCANDATA in Figure 3). Suppose the scan starts in $[t_0, t_1)$ and finishes in $[t_k, t_{k+1})$. We define an operation *collect* such that $collect(DP_0, DP_1)$ merges the data in DP_0 and DP_1: if a memory page is saved in both DP_0 and DP_1, only the data of the page in DP_0 is preserved after the merge. Then,

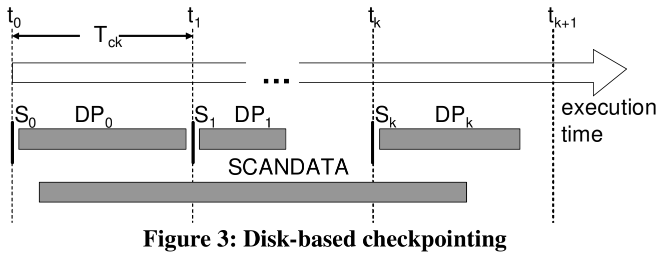

Figure 3: Disk-based checkpointing

$$\begin{aligned} &DPS = collect(...collect(collect(DP_0, DP_1), DP_2)..., DP_k) \\ &S_0 = collect(DPS, SCANDATA), \end{aligned}$$

where DPS is the t_0-state of the memory pages which are modified (updated/created/deleted) during $[t_0, t_{k+1})$. So the checkpoint agent keeps DP_0, DP_1, …, DP_k as well as SCANDATA in order to support disk-based checkpoint. After t_{k+1} the checkpoint agent writes *collect(DPS, SCANDATA)* to disk, the VM checkpoint at t_0.

4 Checkpoint Corruption Model

In this section, we construct a model of checkpoint corruption scenarios to study how proper selection of T_{ck} and the dual-checkpoint scheme in VM-μCheckpoint greatly reduces the probability of failing to recover the protected VM due to error detection latency.

Two factors are important in determining checkpoint corruption: error occurrence instant and error detection latency. We use T_B to denote a bound on error detection latency. By setting T_{ck} greater than an acceptable latency bound T_B (for example, 95th percentile) we impose a certain bound on the probability of a latent/undetected error affecting the checkpoint (less than 5% in the best case for this example).

The following three assumptions are made in our model to simplify the analysis while still providing valuable insight into checkpoint corruption behavior:

(i) Unmasked errors are eventually detected by either application-level (e.g., embedded assertions) or system-level (e.g., application failure, exception handling, kernel panic) detection mechanisms; only an error detection can trigger checkpoint-based recovery. Here an unmasked error is defined as a transient error that remains alive throughout the program life and is not overwritten but is manifested by the program. Error injection study [1] shows that an unmasked error, i.e. a manifested error, only leads to one of the following results: crash/abortion, hang, or fail-silence violation (silent data corruption), and fail-silence violation cases are rare (no more than 2.3% in the experiments). The crash/abortion and hang failures are regarded as two forms of detection in our study because typically they should be detected in reliability-enhanced systems to trigger checkpoint-based recovery. Because the probability of fail-silence violation cases is very small and such cases can be regarded as "being detected after a very long time" in our model study without losing the model's expressiveness, it is reasonable to assume unmasked errors are eventually detected.

(ii) The probability of error occurrence is uniformly distributed during any given checkpoint interval. Note that a checkpoint interval in high-frequency checkpoint schemes typically ranges from tens of milliseconds to seconds or at most minutes. During so short a period a transient error, triggered either by a hardware bit-flip due to radiation or current disturbance, or by a software issue such as race condition, an incorrect parameter or bad transmission, is a completely independent event with respect to the elapsed time in the period. The assumed uniform distribution well captures this independence of the transient error.

(iii) Error latency is exponentially distributed. Error injection study [1] illustrates the distribution of error latency measured in all experiments. The illustration shows that, basically the count of error latency values measured in the experiments largely reduces as the value continues to increase, but there is still a long tail of the error latency values. In the reliability area if it is unable to accurately obtain the analytical formulation of a random variable's distribution but the distribution has aforementioned characteristics, we often model the random variable as exponentially distributed to simplify the analytical tractability while still provide insightful discussion. Examples are the assumptions of exponential distribution for "service time at a server in a queuing network" and "time required to repair a component that has malfunctioned" in [25], the classical textbook on probability (page 120).

The checkpoint corruption model is constructed for a given unmasked error occurrence. We first identify the checkpoint interval in which the error occurrence falls. As an example, Figure 4 (a) shows that an error occurs between *chkpt0* and *chkpt1*. The time offset of the error occurrence relative to the time of *chkpt0* is denoted as *a* ($0 \le a < T_{ck}$). The system continues execution after the error occurrence, and the error is detected after latency of *l* (also shown in Figure 4 (a)). *a* and *l* are two random variables. *a* is uniformly distributed within [0, T_{ck}), and *l* is exponentially distributed at a rate λ. Then, the *pdf* (probability distribution function) for *a* is given by:

$$pdf(a): f(x) = \frac{1}{T_{ck}}, x \in [0, T_{ck})'$$

and the *pdf* for *l* is given by:

$$pdf(l): g(y) = \lambda e^{-\lambda y},\ y \in [0, +\infty).$$

Because the error latency is independent of the error occurrence, *a* and *l* are independent random variables. Then, we can derive the *pdf* for *a+l* as follows:

$$pdf(a+l): h(x,y) = f(x)g(y), x \in [0, T_{ck}), y \in [0, +\infty).$$

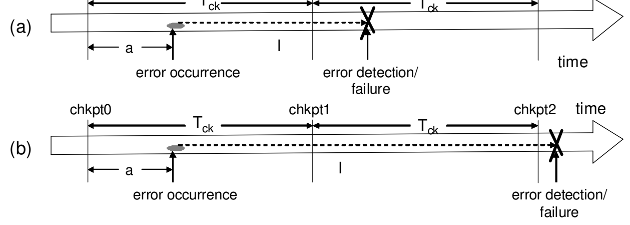

Figure 4: Timeline of checkpoint corruption scenarios

Note that any checkpoint taken after the occurrence of an unmasked error and before the detection of the error (e.g., *chkpt1* in Figure 4(a)), is corrupted. In our model, this condition of checkpoint corruption is represented as $a < T_{ck}\ \&\ a + l > T_{ck}$. The probability of the error corrupting this checkpoint is:

$P\{a < T_{ck}\ \&\ a + l > T_{ck}\} = P\{a + l > T_{ck}\} = 1 - P\{a + l \le T_{ck}\}$ for $a \in [0, T_{ck})$.

$$P\{a + l \le T_{ck}\} = \iint\limits_{x+y \le T_{ck}} h(x,y)dxdy = \iint\limits_{x+y \le T_{ck}} f(x)g(y)dydx$$

$$= \int_0^{T_{ck}} \frac{1}{T_{ck}} \int_0^{T_{ck}-x} \lambda e^{-\lambda y} dydx = \int_0^{T_{ck}} \frac{1}{T_{ck}} (1 - e^{-\lambda(T_{ck}-x)})dx = 1 - \frac{1}{\lambda T_{ck}}(1 - e^{-\lambda T_{ck}})$$

Consequently,

$$P\{a < T_{ck}\ \&\ a + l > T_{ck}\} = \frac{1}{\lambda T_{ck}}(1 - e^{-\lambda T_{ck}}) \qquad (5)$$

Selecting T_{ck} to cover short-lived errors. To mitigate checkpoint corruption, we want to select the checkpoint interval to be larger than the latency of most errors. To do that, it is crucial to get realistic estimates of error latency. How does one obtain such estimates in practice? Two methods come to mind: (i) analyze detection characteristics of detectors deployed in the system/application and (ii) inject faults into the target application/system to measure error latency. For example, according to Gu et al. [1], 95% of Linux kernel crashes have error latency less than 100M CPU cycles (or 50ms on 2G Hz processors). If we use this data in our model, then $P\{l \le T_B\} = p = 0.95$ for T_B=50ms. As $P\{l \le T_B\} = 1 - e^{-\lambda T_B}$, we can compute the value λ = 0.0599 (1/ms) for the test system in [1].

If we select 50ms as the checkpoint interval T_{ck}, i.e., a value covering 95% of error latency, the probability of checkpoint corruption is 31.7% (computed using formula (5)). For $P\{l \le T_B\} = 1 - e^{-\lambda T_B} = 0.99$ and λ = 0.0599, we can compute the value T_B=77ms. So when we increase the checkpoint interval to 77ms that covers 99% of error latency, the probability of checkpoint corruption is reduced to 21.5% (computed by formula (5)).

Dual checkpoint. Selecting the checkpoint interval to be larger than a given bound of latency T_B reduces checkpoint corruption probability. However, it does not ensure that any error occurrence with latency less than T_B does not corrupt a checkpoint. A dual-checkpoint scheme (as shown in Figure 4 (b)) is necessary to provide this assurance. In this scheme, two checkpoints are kept at any time, and the older of the two (*chkpt1* in Figure 4 (b)) is the rollback target during recovery. In this scenario *chkpt1* is corrupted only when $a < T_{ck}\ \&\ a + l > 2T_{ck}$. Then the probability of checkpoint corruption is:

$P\{a < T_{ck}\ \&\ a + l > 2T_{ck}\} = P\{a + l > 2T_{ck}\} = 1 - P\{a + l \le 2T_{ck}\}$

for $a \in [0, T_{ck})$.

$$P\{a + l \le 2T_{ck}\} = \iint\limits_{x+y \le 2T_{ck}} h(x,y)dxdy = \iint\limits_{x+y \le 2T_{ck}} f(x)g(y)dydx$$

$$= \int_0^{T_{ck}} \frac{1}{T_{ck}} \int_0^{2T_{ck}-x} \lambda e^{-\lambda y} dydx = \int_0^{T_{ck}} \frac{1}{T_{ck}} (1 - e^{-\lambda(2T_{ck}-x)})dx = 1 - \frac{e^{-\lambda T_{ck}}}{\lambda T_{ck}}(1 - e^{-\lambda T_{ck}})$$

Consequently,

$$P\{a < T_{ck}\ \&\ a + l > 2T_{ck}\} = \frac{e^{-\lambda T_{ck}}}{\lambda T_{ck}}(1 - e^{-\lambda T_{ck}}) \qquad (6)$$

Table 1 lists the checkpoint corruption probabilities for different error latency percentiles in single-checkpoint and dual-checkpoint scenarios. When dual-checkpoint is used for T_{ck} of 50ms (covering latency of 95% of errors), the checkpoint corruption probability in the dual-checkpoint scheme, i.e. the probability of corrupting the older checkpoint, is largely reduced to 1.59% (using formula (6)). As a comparison, selection of T_{ck} as 115ms in the single-checkpoint scheme has a checkpoint corruption probability of 14.5%, though 115ms is longer than twice of 50ms. As another comparison, selection of T_{ck} as 20ms in the dual-checkpoint scheme has the checkpoint corruption probability of 17.6%, much larger than 1.59%. It is clear that both the dual-checkpoint scheme and the proper selection of T_{ck} must be combined for reducing the checkpoint corruption probability.

Table 1: Checkpoint corruption probabilities in different scenarios

Error latency percentile (*p*)	*p*-percentile point of error latency as T_{ck} (ms)	Prob. of checkpoint corruption in single-checkpoint at T_{ck}	Prob. of checkpoint corruption in dual-checkpoint at T_{ck}
70%	20	58.3%	17.6%
90%	38	39.1%	4.05%
95%	50	31.7%	1.59%
99%	77	21.5%	0.21%
99.9%	115	14.5%	0.015%

5 Availability Model

We construct a Markov model to study the availability improvement provided by VM-μCheckpoint to protected VMs. The model captures failure and recovery behavior of all the involved components.

Modeled Behavior. Specifically, the following behaviors are modeled for different types of failures:

a) *Transient failure of a protected VM* (or an application in the VM): the VM is recovered by the checkpointing VM if there is no checkpoint corruption that VM-μCheckpoint cannot handle (see Section 4). When there is such corruption, a new VM is started on the same physical host, and the jobs being executed at the time of the failure are re-executed from the beginning.

b) *Transient failure of the checkpointing VM:* the checkpointing VM is restarted on the same physical host and begins to receive checkpoints from protected VMs; at the same time, the protected VM is still available for job execution. When a protected VM fails during the failure/restart of the checkpointing VM, our recovery protocol first restarts the checkpointing VM, and then restarts the protected VM (and as before, interrupted jobs are restarted from beginning).

c) *Failure of the hypervisor, and permanent failure of the protected VM or checkpointing VM:* a hypervisor is started either on the same physical node or on another node, the checkpointing VM is started on the hypervisor, and then the protected VM is started (we assume that the disk images of VMs can be loaded from any physical host, which is true in most of current virtualized environments).

Model. Exponential distribution is assumed for the time to failure and the recovery time for all the components. For ease of explanation and brevity, here we present the availability model for only one protected VM on top of the hypervisor. Following notations are used in the model:

λ_v	Rate of the hypervisor software failure and all permanent failures.
λ_s	Failure rate of the checkpointing VM alone.
λ_p	Failure rate of the protected VM alone.
r_v	Restart rate of the hypervisor software.
r_s	Rate of restarting the checkpointing VM, including saving the first checkpoint.
r'_p	Rate of successfully recovering a protected VM by VM-μCheckpoint.
r_p	Rate of recovering a protected VM when VM-μCheckpoint fails to do that (due to checkpoint corruption). Job re-computation is considered as recovery overhead.
p_c	Probability of checkpoint corruption given an error.

The Markov model in Figure 5 depicts the failure/recovery behavior of the system with VM-μCheckpoint deployed. The state of the system is denoted as a vector *(k, j)*, where *k* represents the state of the protected VM and *j* indicates the state of the checkpointing VM (Figure 5 gives more details). The model consists of two sub-models: (a) the sub-model for transient failures of the protected VM and the checkpointing VM and (b) the sub-model for permanent failures and hypervisor failures.

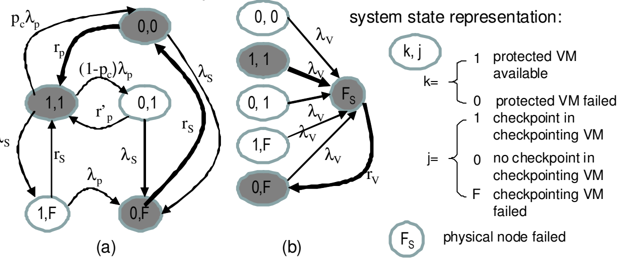

Figure 5: Availability model for one protected VM (a) sub-model for transient failures of guest VMs, (b) sub-model for permanent failures and hypervisor failures

Figure 5 captures all the failure/recovery behavior described above (a), b) and c)). For example, when a permanent failure occurs during execution of the protected VM, the entire physical node fails, and a hypervisor is restarted on a physical node (shown as a sequence of transitions, (1,1)->F_S->(0,F), in sub-model (b)). Then the checkpointing VM and the protected VM (including interrupted jobs) are recovered in turn (shown as (0,F)->(0,0)->(1,1) in sub-model (a)). The checkpoint corruption probability derived from the checkpoint corruption model in Section 4 is multiplied by the failure rate of a protected VM to obtain the rate of failures causing corruptions unhandled by VM-μCheckpoint.

Solving the Model. The Markov model is solved by computing the equilibrium condition, i.e., the "input flow" into each state equal to the "output flow" out of the state [25]. We use the mathematics tool package CLAPACK [27] to solve these equations and obtain the probability of the system staying in each state. Solving the model is required for our availability study in Section 5.1.

We extended the availability model in Figure 5 to capture the behavior of disk-based checkpointing (Figure 3). We also generalized the model in Figure 5 to cases when there are *n* VMs on top of a hypervisor. Due to the space constraints these models are not presented here; they are available upon request.

5.1 Availability Study

After the model is solved, we obtain the availability of the protected VM by adding up the probabilities of the system staying in the states (1,1) and (1,F). To demonstrate the performance of our technique in availability enhancement, we also construct the Markov models for both the baseline case and an existing technique of VM checkpointing based on live migration (Remus [7]) for comparison.

Availability Model for Baseline. In the baseline case, a VM is on top of the hypervisor and there is no any checkpoint of the VM. When a failure occurs to the VM, the VM is restarted with all interrupted jobs restarted from the beginning. The model in Figure 6 captures the baseline behavior with n VMs on top of a hypervisor.

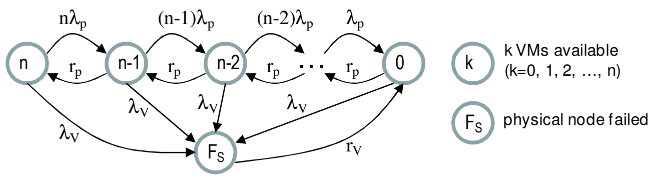

Figure 6: The availability model for baseline (*n* protected VMs)

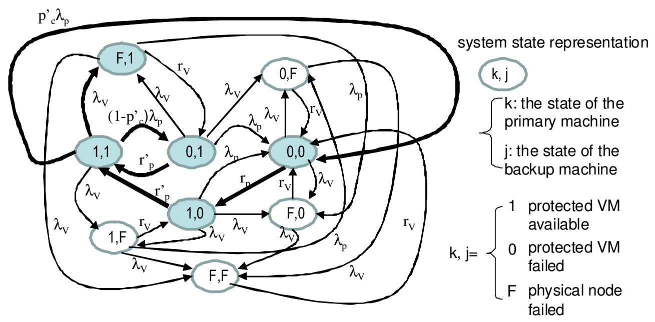

Figure 7: The availability model for Remus (1 protected VM)

Availability Model for Remus. Remus maintains a backup of a VM on a remote host by migrating the state from the primary to the backup periodically (e.g., every 50ms). When the VM fails, the backup VM begins to execute from the last checkpoint. The model in Figure 7 captures the Remus behavior (both checkpoint and recovery) for the cases in which there is only one VM on top of a hypervisor. Due to the space constraints detailed presentation and explanation of this model is not given here, and is available upon request.

Model parameters. The parameters selected in our model are based on previous empirical study of off-the-shelf servers. In the availability study for Windows servers [28] it is reported that, though the average availability of the servers is around 99.9% there are also servers with availability around 99% or less. [28] also reports the MTTR (mean time to recovery) for all the failures as 0.25 hour (or 15 mins). This MTTR includes the response time of an administrator who discovers the failure and restarts the failed machine with appropriate recovery.

So in the availability models for VM-μCheckpoint we set the following parameters:

$r_v = r_s = 1/15$min

$r_p = 1/(0.5*$average job duration), because the VM is restarted immediately and the mean job recomputation during recovery is half of the average job duration

$r'_p = 1/600$ms $= 100$, as overhead around 600ms is measured in our experiments

$\lambda_v = 1/15000$hours $+ 1/3$years $= 0.000001492$/min. 1/15000hours = 1/1.712year is the hypervisor failure rate; 1/3 years is the permanent failure rate (according to presentations made by several Intel engineers in DARPA and other forums)

$\lambda_s = 1/15000$min, for the checkpointing VM with 99.9% availability ($r_s = 1/15$min)

$\lambda_p = 1/1500$min (for a protected VM with 99% availability without considering job recomputation) or 1/15000 min (for a protected VM with 99.9% availability)

$p_c = 1.59\%$, the value is derived in the checkpoint corruption model for T_{ck}= 50ms, which covers 95% of error latency

In this parameter selection, λ_v is much smaller than λ_s or λ_p because hardware and hypervisor is usually assumed to be much more reliable than the server software and the operating system. This is a realistic assumption (also assumed in [24]) because the hypervisor kernel is small (e.g., 434KB for Xen-3.3.1 vs. 1.5MB for Linux 2.6.18) and hence, verification and test of the hypervisor code is relatively easier.

The parameter values above are also used in the availability models for the baseline and Remus (so the recovery rate of Remus is also 1/600ms), except that a different p_c value is selected in the model for Remus. As VM-μCheckpoint uses the dual-checkpoint scheme with the checkpoint interval of 50ms in our model study, for fair comparison, we consider the Remus case with the checkpoint interval of around 100ms because the single-checkpoint scheme is used in Remus. According to the data reported in Section 4, the checkpoint corruption probability is 14.5% in a single-checkpoint scheme at the checkpoint interval of 115ms. Therefore, we select p_c=15% in the Remus model (according to an experimental study in [29], the probabilities of checkpoint corruption range from 27% to 41% for different application workloads; so 15% is a conservative value).

Results. The availability values computed from these models are compared in Table 2. Our results are better than Remus's for all the experiment cases. For example, for average job duration of 8 hours (i.e., $1/r_p$=240 min) on a 99%-available server (λ_p=1/1500min), we achieve an availability of 99.7% while Remus achieves 97.7%.

Table 2: Availability comparison with checkpoint corruption (note that *$1/r_p$ = 0.5*average job duration*)

	$1/r_p$ (min)	15	60	240	1440
λ_p=1/1500min	VM-uchkpt	99.98%	99.92%	99.7%	98.2%
	Remus	99.8%	99.4%	97.7%	87.4%
	Baseline	99.0%	96.1%	86.2%	51.0%
λ_p=1/15000min	VM-uchkpt	99.99%	99.98%	99.93%	99.6%
	Remus	99.98%	99.94%	99.76%	98.6%
	Baseline	99.90%	99.6%	98.4%	91.1%

The better results are achieved because *the impact of checkpoint corruption on availability is much larger than that of permanent failures in high-frequency checkpointing*. As transient failures are much more frequent than permanent failures, our model shows that VM-

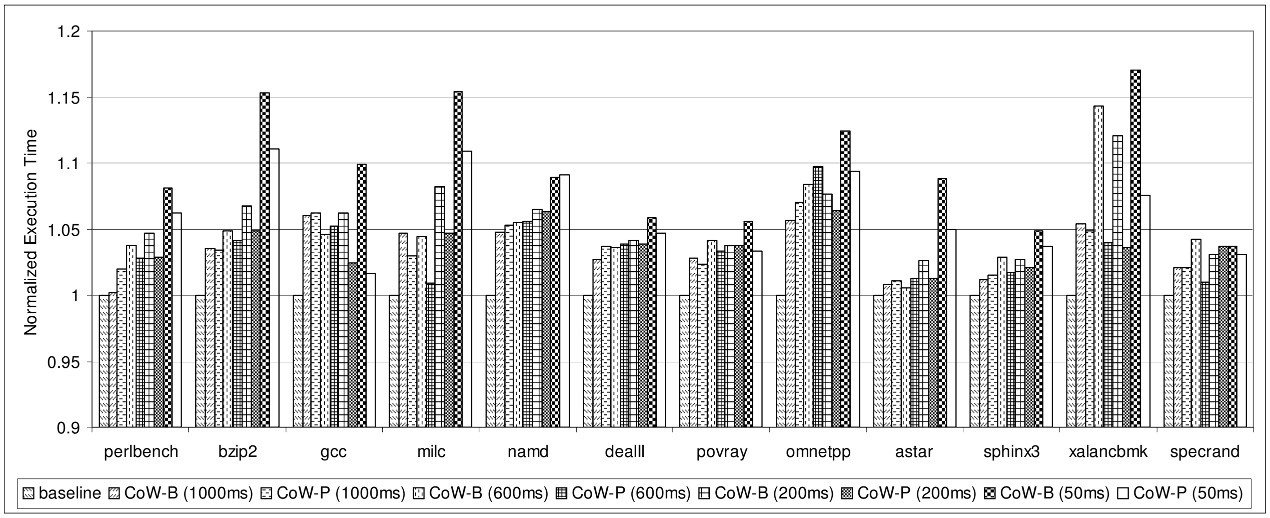

Figure 8: Experiment results in terms of execution time of SPEC CINT 2006

μCheckpoint loses little in handling permanent failures, but gains much more in reducing checkpoint corruption by i) selecting the proper checkpoint interval to cover 95% of errors' detection latency and ii) applying a dual-checkpoint scheme to provide the 100% assurance for recovering these 95% of errors (the rest errors are probabilistically covered).

6 Discussions

Error model. VM-μCheckpoint recovers VMs and applications in the VMs from any transient hardware error or transient software error (including both application error and system error) that escape detection of ECC. These transient hardware errors include those occurring on processors (functional units, registers, caches, buses, and control logics) due to events such as radiation or current disturbance. Transient software errors, or Heisenbugs [4], include exceptional conditions (e.g., counter overflow and interrupt arrival with bad timing), occasional device driver fault, race conditions, and corrupted parameter or data due to bad transmission.

We target the errors that escape detection of ECC, which originate as values incorrectly computed and written to memory. The checkpoint stores the prior-to-update data of the updated pages. The rest part of the VM resident in memory is regarded as correct state.

When there is corruption of checkpoint that VM-μCheckpoint is unable to address, VM-μCheckpoint aborts recovery and restarts the VM and the interrupted jobs. Moreover, when a transient error in the hypervisor causes the entire hypervisor to fail, we first restart the hypervisor and restart all jobs executing prior to the failure; if this is unsuccessful, we move to an adjacent physical node and restart the hypervisor.

I/O handling. This paper focuses on the analysis, design, modeling, and implementation of memory-state checkpointing in VM-μCheckpoint. For I/O handling, we can adapt the output-commit mechanism applied in [7][16] to fit into VM-μCheckpoint. In this mechanism, output of a system is held (i.e., not delivered to hardware devices) until a checkpoint is taken. This mechanism masks recovered errors of the system, i.e., these errors are not viewed by other components (disks, network cards, nodes, etc.).

Here is how VM-μCheckpoint can be extended to support I/O checkpoint. The checkpoint agent in VM-μCheckpoint is designed to hold and release output of the protected VM; if preferred, a copy of input to the protected VM is saved in the checkpoint agent for replay. The hypervisor and the checkpointing VMs are instrumented to provide support to the checkpoint agent for this purpose. Note that the checkpoint agent maintains two pools of held outputs and saved inputs in the dual-checkpoint scheme.

7 Experimental Performance 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[2].

The testbed consists of a physical machine with an AMD Athlon 2800 (1.8G Hz) processor and 1.5GB memory. There are only two VMs (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 VM (a DomU) is the protected VM. 512MB and 1GB memory are assigned to the Dom0 and the DomU, respectively. We use only two VMs in experiments in order to accurately measure performance overhead in a relatively simple deployment.

We summarize major findings in our experiments below:

a) VM-μCheckpoint achieves much better performance than existing migration-based VM checkpoint. For workload of SPEC CINT 2006 benchmark and checkpoint frequency of 20 times per second

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

(T_{ck}=50ms), an average of 6.3% overhead is incurred when COWP is deployed. With the same checkpoint algorithm and checkpoint frequency Apache server throughput is reduced by 17.5%, compared with cases without any checkpoint mechanism applied. In contrast, Remus [7], a migration-based VM replication/checkpoint technique, reports approximately 50% overhead in their experiments for the same checkpoint frequency. If we reduce the frequency to 5 times per second the average overhead is only 3.8% for the SPEC CINT 2006 workload and around 9% for the Apache web server (using COWP).

b) The speedup the COWP algorithm gains over COWB is significant when checkpoint frequency is high. With COWP deployed with 50ms checkpoint intervals Apache throughput is 82.5% of the baseline performance, which is larger than 74.3% when COWB is deployed.

c) Checkpoint sizes are relatively small with short checkpoint intervals selected in our experiments. The results show that, with 50ms checkpoint intervals (using COWP) all checkpoint sizes are less than 2000 memory pages (8MB) with an average of 655 pages (2.6MB), for the SPEC CINT 2006 workload (the size of the entire system state is up to 51461 memory pages, or 206MB).

7.1 Program Execution Time

A set of SPEC CINT 2006 benchmark programs are executed in the protected VM with VM-μCheckpoint deployed. A suite of experiments are conducted involving each of these benchmark programs: (i) a baseline case (no checkpoint), (ii) COWB algorithm deployed with 4 different checkpoint intervals (1000ms, 600ms, 200ms, and 50ms), and (iii) COWP algorithm deployed with the same 4 intervals. A given program executes with the same input across all experiments.

Program execution times are measured. The execution times normalized against the execution time in the corresponding baseline case are illustrated in Figure 8. The following can be observed from Figure 8:

i) For all programs the impact of the checkpoint on the program execution time is no more than 11% (the normalized execution times are no more than 1.11) and the average overhead is 6.3% (the average of the normalized execution times is 1.063) when the COWP algorithm is deployed with 50ms checkpoint intervals. Comparing with around 50% overhead in Remus this is great improvement. If we increase the checkpoint interval to 200ms, the average overhead is now 3.8% (using COWP).

ii) The performance overhead increases as checkpoint frequency grows.

iii) Use of COWP gains larger speedup over COWB for high checkpoint frequency. This is because the pre-saving in COWP reduces the number of page faults when the checkpoint interval is small. For low-frequency checkpoint the pre-saving does not provide much improvement; in certain cases it even degrades checkpoint performance. Such performance degradation can be observed in experiments for *perlbench* and *omnetpp* in Figure 8. This result is due to the fact that memory access locality plays a significant role when checkpoint intervals are short. With a checkpoint interval as large as 1s there are (in general) a lot of mispredictions, and the pre-saving does a lot of wasteful work of preserving pages not to be updated.

7.2 Web Server Throughput

Apache web server runs on the protected VM in our experiments. Web clients reside on three physical machines with each machine hosting 50 clients. These clients simultaneously request the same load of web pages, one request immediately after another, from the server via a 100Mbps LAN. The output-commit mechanism is disabled in these experiments (as the I/O handling is not the focus of this paper), and consequently, we compare our performance with Remus results when the output commit is also disabled.

Figure 9 shows the measured server throughput with VM-μCheckpoint deployed at different checkpoint intervals. The percentages indicated along the data points on the graph represent the ratio between the throughput measured with the checkpoint deployed and the throughput in the baseline case (when checkpoint is not deployed). The three observations in Section 7.1 are confirmed by the throughput results.

i) The throughput is reduced by 17.5% when checkpoint is taken 20 times per second. Remus reports approximately 50% overhead for SPECweb benchmark with the same checkpoint frequency (with output commit disabled). If the checkpoint interval is increased to 200ms in VM-μCheckpoint, the throughput is reduced by only 9%. Our overhead results are conservative because we run a stressful load of web requests in experiments, and the typical server workload is not as intensive.

ii) Checkpoint overhead increases with higher checkpoint frequency.

iii) The COWP algorithm improves performance over COWB, especially in cases with small checkpoint intervals (the gaps between the two curves keep increasing as the checkpoint interval decreases in Figure 9).

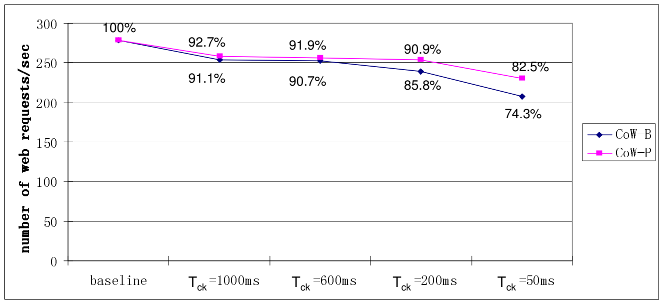

Figure 9: Impacts of VM-μCheckpoint on Apache web server throughput (a percentage represents the ratio between the corresponding throughput and the baseline throughput, e.g., 82.5%=229.8/278.7)

7.3 Overhead Measurement

The number of checkpoint-caused page faults is a direct measurement of the time overhead of VM-μCheckpoint (other page faults are not checkpoint overhead). Checkpoint size, i.e., the number of memory pages kept in a checkpoint, is the space overhead measurement.

A number of experiments are conducted to study VM-μCheckpoint's overhead with COWB or COWP at different checkpoint intervals. In each of these experiments, 12 programs of SPEC CINT 2006 benchmark were executed in a sequential way. We measured the numbers of checkpoint-caused page faults and the checkpoint sizes (in terms of numbers of memory pages) in each checkpoint interval (e.g., 50ms) of the experiment duration.

Table 3: Average checkpoint-caused page faults in experiments

Algorithm	T_{ck}(ms)	Overall	①perl-bench	③gcc	④milc	⑥dealII
COWB	50	491.1	342.4	396.0	1112.7	235.8
	1000	1057.2	1076.6	1398.1	1298.0	996.0
COWP	50	124.7	139.7	172.2	35.7	163.3
	1000	521.5	182.0	527.9	319.7	842.8

Table 4: Average checkpoint sizes (in number of memory pages) in experiments

Algo-rithm	T_{ck}(ms)	Over-all	①perl-bench	③gcc	④milc	⑥dealII
COWB	50	491.1	342.4	396.0	1112.7	235.8
	1000	1057.2	1076.6	1398.1	1298.0	996.0
COWP	50	654.5	565.7	626.3	1154.6	524.2
	1000	2162.4	1260.0	2144.6	1873.9	3151.6

Results. The overall checkpointing overheads measured throughout the experiments, as well as the overheads for several individual programs, are presented in Table 3 (time overhead) and Table 4 (space overhead). The major findings from the experimental results are summarized below.

- *Average checkpoint sizes are very small, less than 2% of the size of the entire system state when checkpoint interval is 50ms.* Table 5 shows that with COWP 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 51,461 memory pages (206MB). The maximum checkpoint size observed is less than 8MB (2000 pages; due to space constraints the figure showing the distribution of checkpoint sizes is not given here), 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).
- *Dirty page prediction and pre-saving effectively reduce page faults by 75% when the checkpoint interval is 50ms* (124.7 page faults in COWP, while 491.1 in COWB, as shown in Table 3). When the checkpoint interval is 1000ms, COWP still achieves 51% reduction of page faults (1057.2 reduced to 521.5). The reduction percentage is less for larger checkpoint intervals, as there is more memory access locality within shorter intervals.

8 Error Injection Experiments

Detections of a variety of errors are provided by the RMK, including detection of VM crashes and hangs. As discussed in Section 2, VM-μCheckpoint consists of three RMK modules in the deployment (Figure 1): *COWB, COWP,* and *recovery*. The error detections and the VM-μCheckpoint are integrated by the RMK framework, which is briefly described here (VM crash is used as the example error in the description):

When a protected VM crashes, an exception is raised by the hardware and reported to the hypervisor. The P_VMSIGNAL pin (illustrated in Figure 1) intercepts this exception, determines whether the exception is caused by a crash (e.g., segmentation fault) or not (e.g., a page fault), and produces an EVT_ERRORDETECTED event if the exception is raised by a crash. Detections of other errors (e.g. system hangs) also raise this event to the RMK framework.

The *recovery* module receives the event EVT_ERRORDETECTED from the RMK, then suspends the failed VM and invokes the checkpoint agent for recovering the VM state with its checkpoint (the *recovery* module knows whether COWB or COWP is currently used).

Table 5 lists the experiment results. Three kinds of errors were injected, as described below.

Table 5: Results of Error Injection Experiments

Experiments	Fault/Error	Injected Faults/Errors	Activated and Detected	Recovered
Signal-triggered crashes (checkpoint interval= 100ms)	Sends SIGTERM to the application in the protected VM (fail-stop)	35	35	35
Bit flips into kernel registers (checkpoint interval= 100ms)	Suspends the protected VM, flips a bit in a register value in the VM, and resumes the VM	85	31 activated and detected	31
System hangs (threshold of 200 ms, checkpoint interval= 500ms)	Loads a device driver which runs an infinite loop in the protected VM	30	30	24*

*Recovery failed in several experiments due to inconsistent shared state between the protected VM and Dom0 for I/O operations

8.1 Signal-Triggered Crash

We use signals to emulate VM crashes in these experiments. We send a SIGTERM signal to the application process; this signal traps the processor into the hypervisor; the P_VMSIGNAL pin in the hypervisor injects an error of a VM crash by generating the EVT_ERRORDETECTED event; the recovery module then restores the checkpoint to the protected VM. We conducted 35 experiments, and in all of them the protected VM was successfully recovered.

8.2 Bit Flips in VM Kernel Registers

We also injected bit-flip faults into kernel registers of the protected VM. The error injector is placed outside the protected VM, i.e., in the dom0, to avoid repeating the same error injection after recovery from the checkpoint. Our experiments showed that if the error injector is inside the protected VM, the VM is recovered from the checkpoint successfully after error detection, but the same error is injected again immediately after recovery as the error injector process was running before the error was injected. A simple program is used as the workload in these error injection experiments.

The error injector in the dom0 first suspends the protected VM via the hypervisor. As a result, the state of the virtual CPUs in the VM is saved in a hypervisor data structure called *vcpu_guest_context.* Then we randomly select a bit in the generic register file (i.e. EAX, EBX, ESI, EDI, ESP, EIP, CS, SS, ES, etc.) in the vcpu_guest_context and flip it. We resume the protected VM, and the flipped value is written back to the corresponding register in the virtual CPUs.

We did tens of experiments and found that the activation rate is fairly small (i.e., only 31 out of 85 injected faults get activated). All of the activated and manifested errors in the experiments are detected by the P_VMSIGNAL pin and are successfully recovered.

8.3 System Hangs

We injected system hangs in the protected VM as follows: we loaded in the VM a device driver that ran an infinite loop. In the 30 experiments conducted, there are 6 cases in which the recovery from the checkpoint fails.

We looked into the details of the cases when the recovery failed, and discovered the failures were related to a certain I/O issue: the shared state between the protected VM and the dom0 for handling I/O operations is inconsistent with the state of the protected VM after recovery from the checkpoint. Figure 10 illustrates the details of the inter-domain shared memory for I/O operations in Xen.

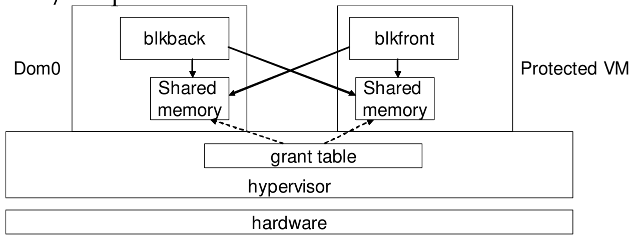

Figure 10: Inter-Domain Shared Memory for I/O Operations in Xen

The Xen hypervisor uses a split driver model for handling I/O operations (network, disk, etc.). The blkfront is the front end of the driver in the protected VM, and the blkback is the back end in Dom0 (as shown in Figure 10). Shared memory is used to facilitate I/O data transfer. These shared states include request ring buffer, producer/consumer pointers (blkfront and blkback follow a producer-consumer model), buffers, protocol status for the split driver, event channel state, etc.

When a request arrives at either the blkfront or the blkback, a shared buffer is created in the protected VM or the dom0 to host I/O data, and this buffer is registered through the grant table mechanism in the hypervisor. After processing of the request, the buffer is released through the grant table.

After the protected VM is recovered from checkpoint, there are cases when the blkfront expects a shared buffer to be present and registered in the hypervisor's grant table. But this may not be true. The recovery then fails because a non-existent buffer is accessed. Our experiments show that this scenario happens when the error detection latency is large. That is why we only observed failure of recovery in cases when system hangs are injected (200 ms is used as the threshold value for detecting VM hangs).

To handle this problem, we will instrument the hypervisor and the blkback driver in the dom0 to save the shared memory and the grant table in the checkpoint. This work is in our next stage of the research focusing on I/O checkpoint.

8.4 Recovery Overhead

Besides the error injection experiments for testing the correctness of the checkpoint/recovery mechanism of VM-μCheckpoint, we also measured the recovery time to evaluate the performance. The hypervisor-level exception handler is instrumented to provide the measurement information. The SPEC CINT 2006 benchmark programs ran as the workload on the protected VM in these experiments. The measured recovery time depends on the number of memory pages restored during recovery. As most of checkpoint sizes range from several hundred to several thousand memory pages (shown in Table 4), the measured recovery time ranges from 144ms to 1017ms with the average of 639.4ms (the 95% confidence interval is 639.4ms ± 193.1ms).

9 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, as they are more relevant to our objective.

VM checkpointing. Most existing VM checkpoint/ replication techniques are based on live migration of VMs (e.g., VMWare VMotion [5] and Xen Live Migration [6]), 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 [17], VNsnap [23], and VM Snapshots [12] are techniques of disk-based VM checkpointing. These techniques employ VM live migration or copy-on-write to create a replica image of a VM with low downtime incurred; then they write the image to disk offline. Another project on VM checkpoint [12] tries to provide a generic API in Xen product for saving a VM snapshot to disk on demand. Basically, the VM memory is scanned and saved to files while the VM runs simultaneously. Copy-on-write is exploited to save the original data of modified VM state during checkpointing.

VM-μCheckpoint is different from these disk-based VM checkpointing in that we aim at i) providing high-frequency checkpointing and rapid recovery of VMs with low overhead, which allows VM failures to be masked to clients, and ii) proposing a mechanism to address checkpoint corruption in high-frequency checkpointing. Checkpoint corruption has large impacts on service availability when checkpoint frequency is high, as shown in our model study in Section 4 and 5.

The existing approach that is closest to our work is Remus [7], which maintains a backup VM on a separate physical node by periodically transmitting the VM's dirty pages to the backup. Similar to VM-μCheckpoint, Remus is a mechanism of high-frequency VM checkpointing and failover. But VM-μCheckpoint focuses on error behavior and reliability/availability improvement, while Remus focuses on migration overhead. No study of error behavior or reliability/availability is reported in [7]. As checkpoint corruption is not handled in Remus (fail-stop errors are assumed in Remus), our technique is better in improving service availability, as shown by our availability study in section 5.1.

Other techniques that may be relevant to VM checkpointing are briefly described as follows. Bradford et al. [22] focus on migrating persistent state of a VM across WAN so that the VM can migrate to a node that does not share storage with the original node. [11] revises Xen live migration to fit in a self-migration scenario. [10] and [8] implement proactive VM rejuvenation based on live migration, and [9] uses live migration for load-balancing. Potemkin [19] employs copy-on-write to share data between VMs for efficiently provisioning VMs. Another technique of fast VM forking is [20]. Though one may use these techniques to checkpoint a VM by periodically forking a shadow VM and tearing down out-dated shadow VMs, spawning a VM and tearing down a VM involve a lot of overhead not necessary for checkpointing. Moreover, error analysis and reliability/availability study is an integral part of a checkpointing technique for failure mitigation.

Multi-checkpoint mechanisms. As far as we know, none of the existing checkpoint techniques considers handling checkpoint corruption by explicitly studying error detection latency and including a bound of the latency as a parameter, though multi-checkpoint mechanisms can be leveraged to deal with checkpoint corruption. IBM System Z [18] allows multiple checkpoints of an application to be recorded in persistent storage on demand. Ping-Pong checkpoint [21] maintains two checkpoints to deal with incomplete checkpoint due to errors during the checkpointing procedure. None of these techniques study the characteristics of error detection latency to address the checkpoint corruption.

In-place restoration. Hardware-level checkpoint techniques [14][15] use special hardware to take and store a checkpoint. When an error is detected, the checkpoint saved in the special hardware is restored into the architecture state of the physical machine, including register file and memory. For example, the first update of a memory word or a register during a checkpoint interval is preserved in special hardware in SafetyNet [14].

10 CONCLUSIONS

This paper proposes VM-μCheckpoint, a lightweight VM checkpointing technique, which minimizes overhead by placing checkpoints in memory and performing in-place recovery. VM-μCheckpoint provides high-frequency checkpointing (e.g. 20 times of checkpoints per second) and rapid recovery of VMs. As checkpoint corruption has large impacts on the probability of recovery failures when checkpoint frequency is high, VM-μCheckpoint explicitly addresses checkpoint corruption based on study of the characteristics of error detection latency. We constructed Markov models to study the error detection latency and system availability under different checkpoint mechanisms. The results of the model study clearly show that VM-μCheckpoint effectively handles checkpoint corruption and largely improves service availability by means of i) proper selection of checkpoint interval based on the knowledge on error detection latency and ii) the dual-checkpoint scheme.

VM-μCheckpoint was implemented in the Xen VMM. Experimental results show that the proposed technique achieves much better performance than existing techniques based on VM live migration. There is an average of 6.3% overhead in terms of program execution time for the SPEC CINT 2006 benchmark when VM-μCheckpoint is deployed at a checkpoint frequency of 20 times per second. (An approximately 50% overhead is reported in a previous technique [7] at the same checkpoint frequency.) Moreover, the checkpoint size is small in VM-μCheckpoint: an average of 2.6MB in our experiments when the COWP algorithm is applied with 50ms checkpoint intervals.

We conducted error injection experiments by deploying VM-μCheckpoint in RMK to leverage the existing error detection techniques in RMK. The error injection experiments demonstrate that VM-μCheckpoint has high coverage of error recovery (100% for system crashes and corrupted data in our experiments).

REFERENCES

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

[2] Karthik Pattabiraman et al. Automated Derivation of Application-aware Error Detectors using Static Analysis. *International On-Line Testing Symposium*, 2007.

[3] G. A. Reis et al. SWIFT: Software Implemented Fault Tolerance, *In Proc. 3rd International Symposium on Code Generation and Optimization*, 2005.

[4] Jim Gray, Why Do Computers Stop and What Can Be Done about It? *In Proc. Fifth Symposium on Reliability in Distributed Software and Database Systems*, 1986.

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

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

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

[8] A. B. Nagarajan et al. Proactive Fault Tolerance for HPC with Xen Virtualization, *Proceedings of International Conference on Supercomputing*, 2007.

[9] T. Wood, et al. Black-box and Gray-box Strategies for Virtual Machine Migration, *Networked Systems Design and Implementation*, 2007.

[10] Tobias Distler et al. Efficient State Transfer for Hypervisor-based Proactive Recovery, *2nd Workshop on Recent Advances on Intrusion-Tolerant Systems*, 2008.

[11] Jacob Gorm Hansen et al. Self-migration of Operating Systems, *11th Workshop on ACM SIGOPS European Workshop*, 2004.

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

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

[14] D. J. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery, *International Symposium on Computer Architecture*, 2002.

[15] Milos Prvulovic et al. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors, *International Symposium on Computer Architecture*, 2002.

[16] Jun Nakano, et al. ReViveI/O: efficient handling of I/O in highly-available rollback-recovery servers. *HPCA* 2006.

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

[18] Developer for System z, Version 7.0, Enterprise COBOL for z/OS, Version 3.4, Programming Guide, *http://publib.boulder.ibm.com/infocenter/ratdevz/v7r1m1/index.jsp?topic=/com.ibm.ent.cbl.zos.doc/topics/tpchk04.htm*

[19] M. Vrable et al. Scalability, Fidelity, and Sontainment in the Potemkin Virtual Honeyfarm, *The ACM Symposium on Operating Systems Principles*, 2005.

[20] H. A. Lagar-Cavilla et al. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing, *ACM European Conference on Computer Systems*, 2009.

[21] Le Gruenwald Le et al. Survey of Recovery in Main Memory Databases, *Engineering Intelligent Systems* 4/3, Sept. 1996.

[22] Robert Bradford et al. Live Wide-area Migration of Virtual Machines Including Local Persistent State, *The 3rd International Conference on Virtual Execution Environments*, 2007.

[23] Ardalan Kangarlou et al. VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Downtime, *Dependable Systems and Networks*, 2009.

[24] Hans P. Reiser et al. Hypervisor-Based Redundant Execution on a Single Physical Host, *European Dependable Computing Conference*, Supplemental Volume, 2006.

[25] Kishor S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications, 2nd Edition,* John Wiley & Sons, Inc., New York. 2002.

[26] Man-Lap 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.

[27] CLAPACK (f2c'ed version of LAPACK), http://www.netlib.org/clapack/

[28] J. Xu et al. Networked Windows NT System Field Failure Data Analysis, *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 1999.

[29] S. Chandra, P. M. Chen, The Impact of Recovery Mechanisms on the Likelihood of Saving Corrupted State, *International Symposium on Software Reliability Engineering,* 2002.

[30] Gu et al. Fault Inject Based Study of Fault Resilience of Hypervisor, University of Illinois Urbana Champaign report 2007.

[31] L. Wang, Z. Kalbarczyk, W. Gu, R. K. Iyer, Reliability MicroKernel: Providing Application-Aware Reliability in OS, *IEEE Transactions on Reliability*, Vol. 56, No. 4, Dec. 2007.

Long Wang obtained Master degree in Computer Science from University of Illinois at Urbana Champaign, Urbana, IL in 2002, and got Ph.D. degree in Electrical and Computer Engineering from University of Illinois at Urbana Champaign, Urbana, IL in 2010. Then Dr. Wang joined IBM Thomas J. Watson Research Center, Yorktown Heights, NY in 2010. He has published more than 18 papers in top conferences and journals. His current research interests include Fault-tolerance and Reliability of Systems and Applications, Dependable and Secure Systems, Distributed Systems, Cloud Computing, Operating Systems, System Modeling, Measurement and Assessment. Dr. Wang is a member of the IEEE.

Zbigniew Kalbarczyk bio.

Ravishankar K. Iyer bio.

Arun Iyengar bio.