Evaluation of a Tenant Level Checkpointing Technique for SaaS Applications

Hong Zhu, Basel Yousef, and Muhammad Younas
Department of Computing and Communication Technologies
Oxford Brookes University, Oxford, OX33 1HX, UK
Email: hzhu@brookes.ac.uk, basel.yousef-2011@brookes.ac.uk, m.younas@brookes.ac.uk

***Abstract*—Tenant level checkpointing is a novel fault-tolerance technique proposed in our previous work for large-scale software-as-a-service applications. This paper evaluates the technique by a theoretical analysis of the technique and an empirical study using workload benchmarks of real large-scale cluster systems. We propose the notion of accumulated delay as a measurement of the performance of checkpointing techniques. It reflects the effect of checkpointing on system performance as experienced by the end users. Both theoretical analysis and empirical study show that tenant level checkpointing significantly outperforms traditional bulk checkpointing as measured by the amount of accumulated delays.**

***Keywords*-Software-as-a-Service; Multi-tenancy; Fault Tolerance; Checkpoint-and-Rollback.**

I. Introduction

Fault-tolerance is one of the most challenging issues of cloud computing [1]. The past years have seen a rapid growth of research in this area. However, little has been reported on fault-tolerance of system scale failures. Such a failure causes lost of services to a large number of users and takes a long time to recover. For example, in two recent incidents, Salesforce.com losted its services to 100,000+ clients for more than 10 hours.

Saleforce's CRM is a typical multi-tenancy Software-as-a-Service (SaaS) application. A SaaS application can be in one of two types of system architectures [2], [3]: *multi-instance* or *multi-tenancy*. The former employs virtual machines to install multiple instances of the software so that each instance of the software only serves one tenant. In the latter, each instance of the software delivers services to multiple tenants. It is widely accepted in the industry that the multi-tenancy architecture has advantages over multi-instance in terms of cost reduction, scalability, maintainability, and usability [4].

However, the multi-tenancy architecture is vulnerable to system scale failures. For this reason, Salesforce backs up all data to a tape storage on a nightly basis. This traditional checkpoint-and-rollback fault tolerance technique is unsatisfactory for SaaS applications. In fact, Salesforce's tenants also use third party facilities for backing up their own data. Addressing this problem, in [5], we proposed a new approach called *tenant-level checkpointing* (TLCP) and implemented a prototype called *Tench*. In this approach, instead of saving the whole system's state, each checkpointing (CP) only saves a part of system state related to a specific tenant. A number of experiments with the system have also been conducted to evaluate its performance. In this paper, we present an extended evaluation of the proposed approach in order to prove its advantages over existing techniques, i.e. the bulk checkpointing (BCP) techniques.

The main contributions of the paper are: (a) we propose the notion of accumulated delay as a measurement of the performances of SaaS CP techniques. We argue that it reflects the effect of CP on system performances as experienced by the end users and formally prove its properties. (b) We compare TLCP against BCP using accumulated delays as the performance measurement and theoretically prove that TLCP outperforms BCP. (c) We conduct an empirical study using the workload benchmark of five real large scale cluster systems. This provides an empirical evidence of the validity of TLCP as a solution to SaaS applications.

The remainder of the paper is organised as follows. Section 2 briefly reviews related work on CP. Section 3 presents a theoretical analysis of the TLCP technique. Section 4 reports on the empirical study. Section 5 concludes the paper with a discussion of future work.

II. Related Work

The existing works on CP in cloud computing have targeted fault tolerance research in terms of resource [6], infrastructure [7]–[10], platform [11], [12], and system levels [13]–[21], etc. As Yang *et al* argued [22], with the increase of system scale, system level CP will soon hit the so-called reliability wall and become infeasible for exascale supercomputing. An approach to overcome this problem is to reduce the demand on CP operations and to improve their effectiveness by taking into full consideration of application architectures.

In this paper, we are concerned with CP for SaaS in the multi-tenancy architecture. There exists limited work in this area. In a wider context, Lu et al. [23] proposed a technique for CP at application level. They introduced the notion of virtual clusters, which consists of a set of virtual machines deployed on a number of physical servers and managed as a single entity. CP operations are performed on virtual machines to achieve global optimization. They built on top of existing works on VM CP, replication [24]–[26] and live migration [27]. These facilities have been exploited in balancing service work load [28], reducing system energy consumption [29], and reducing users monetary cost [30]. These work all assume that a virtual machine is created for each user and CP operations are performed on virtual machines. Thus, they are only suitable for SaaS applications in multi-instance architecture but not in multi-tenancy.

In comparison with existing CP techniques, TLCP reduces the pressure of CP on system performance by only saving a partial state of the system that is related to a tenant and only when a CP is needed and is suitable to the tenant.

III. Theoretical Analysis

In this section, we define a formal model of CP techniques in the context of SaaS, and compare TLCP against BCP.

A. Notations and General Assumptions

A SaaS application in multi-tenancy architecture provides services to a set $\Gamma = \{T_i | i \in 1, \cdots, N\}$ of tenants and each tenant $T \in \Gamma$ has a number of users. In general, Γ varies from time to time as new tenants joins and existing tenant quit. However, without lost of generality, we assume Γ to be constant during each CP operation.

Each request r from a user belongs to one and only one tenant. We use $Tenant(r)$ to denote the tenant that request r belongs to. We will use $RecvT(r)$ and $RespT(r)$ to denote the time moments that a request r is received and the response is sent to the user, respectively. We define $LenT(r) = RespT(r) - RecvT(r)$.

Given time moments t and t' $(t < t')$, we use $Req[t, t']$ to denote the set $\{r_1, r_2, \cdots, r_K\}$ of all service requests received between t and t', and $Req^T[t, t']$ to denote the subset of requests in $Req[t, t']$ from the users of a tenant T. Since each request belongs to one and only one tenant, we have that, for all time moments t and t',

$$Req[t, t'] = \bigcup_{T \in \Gamma} \left(Req^T[t, t'] \right) \quad (1)$$

$$Req^T[t, t'] \cap Req^{T'}[t, t'] = \emptyset, \ if \ T \neq T'. \quad (2)$$

Let $St(t)$ be the state of the system at the time moment t. We assume that $St(t)$ can be decomposed into a number of sub-states $St^T(t)$ of the tenants $T \in \Gamma$ plus a sub-state $St^C(t)$ of system core. We use $Vol(s)$ to denote the volume or size of a state s. Here, we assume that a service request from a tenant T only changes the sub-state of the tenant T. It does not affect the sub-state of any other tenants, nor the system's core state. The system's core state is only changed due to (a) operations on the tenants, for example, adding or removing a tenant from the system, and (b) operations on system resources, such as adding or removing compute nodes and storage spaces, etc. This assumption is based on the fact that for all SaaS applications that we know, the data from different tenants are separable from each other for security reasons. This assumption can be formally expressed in Equation (3). For all time moments t,

$$Vol(St(t)) = Vol(St^C(t)) + \sum_{T \in \Gamma} Vol(St^T(t))) \quad (3)$$

Note that, depending on the CP mechanism used, a CP operation could save the whole state at that time or just a part of the state modified since the previous CP operation if an incremental CP is applied. Irrespective of the CP technique being used, we use $ChP(t)$, $ChP^T(t)$ and $ChP^C(t)$ to denote the CP data at time moment t for the whole system, for tenant T and for the system core, respectively. Since CP data is either the state or a part of it, from (3) we have that

$$Vol(ChP(t)) = Vol(ChP^C(t)) + \sum_{T \in \Gamma} Vol(ChP^T(t)) \quad (4)$$

Existing CP techniques all extract and save the state for all tenants rather than tenant-by-tenant. In the sequel, we call such a CP *bulk checkpointing* (BCP). We use $St^B(t)$ and $ChP^B(t)$ to denote the state of all tenants and the BCP data at a time moment t, respectively. Thus,

$$Vol(St^B(t)) = \sum_{T \in \Gamma} Vol(St^T(t)) \quad (5)$$

$$Vol(ChP^B(t)) = \sum_{T \in \Gamma} Vol(ChP^T(t)) \quad (6)$$

We assume that a synchronised CP technique is employed. Note that, the time required to synchronise the processes in a system is independent to the size of CP data but a factor related to system complexity. In contrast, the time required to extract CP data and save the data into a storage is usually a function of the size of the CP data. We use $Delay(s)$ to denote the length of time required to CP state s. Thus,

$$Delay(s) = Latency(Vol(s)) + \Delta \quad (7)$$

where Δ and $Latency(x)$ are the times required synchronisation and for creating a CP data of size x.

Our previous experiments with Tench [5] shows that the time required to extract and save a CP data is linear to its volume; see Figure 1. Thus,

$$Delay(x) = \alpha x + \beta \quad (8)$$

where x is the size of the CP data, α and β are constants determined by the particular CP technique.

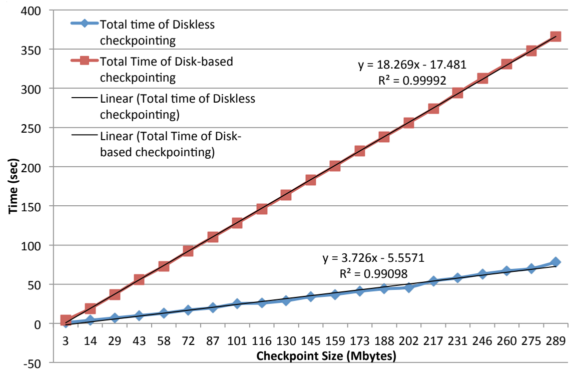

Figure 1. Latency of CP against Data Sizes

B. Measuring the Effect on System's Performance

In the literature, the impact of CP operations on system performance is mostly measured by the latency, i.e. the time required to complete a CP operation. It is fair when used to compare two CP techniques that affect the same set of users, but unfair when used to compare TLCP against BCP, because the latter affects all users while the former affects only the users of one tenant. Therefore, here we propose an alternative measure called *accumulated delay*.

Let t and t' be the time moments that a CP operation starts and finishes, respectively. Two types of service requests may be affected by a CP operation.

First, for a service request r received during the time period $[t, t']$, it may have to be queued and held until the CP finishes, if it affects the part of state space being saved by the CP. This set of queued requests depends on the particular CP technique. For example, for BCP, all service requests that are issued to the SaaS application during the CP period $[t, t']$ must be blocked due to CP. So, the set of blocked service requests for BCP is $Req[t, t']$. But, for TLCP, only those requests from the tenant will affect the part of state that are extracted and saved. Thus, the set of blocked service requests for a TLCP for tenant T is $Req^T[t, t']$, which is only a subset of $Req[t, t']$.

Second, for a service request being processed in the system at time moment t, it has to be paused during the CP operation, if it affects the part of state space being saved by the CP. Again, this set of paused requests depends on the particular CP technique. For BCP, the set of paused requests includes all service requests being processed at the time moment t. But, for TLCP, we only need to pause those requests that affect the part of state of the tenant.

Therefore, a CP operation Ch can be characterised by its starting and finishing time moments t and t', the CP data ChP, the set of blocked requests Q and the set of paused requests P. In the sequel, we denote $Ch = \langle t, t', ChP, Q, P\rangle$.

Now, we define the notion of the accumulated delay.

Definition 1: Let $Ch = \langle t, t', ChP, Q, P\rangle$ be a CP operation. The *accumulated delay* caused by Ch, denoted by $AccDelay(Ch)$, is formally defined as follows.

$$AccDelay(Ch) = \delta_{Ch} \times ||P|| + \sum_{r \in Q} (t' - RecvT(r)) \quad (9)$$

where $\delta_{Ch} = t' - t$, $||X||$ is the size of set X. ■

Informally, accumulated delay is the total amount of delay that users experience due to a CP operation.

C. Properties of Accumulated Delays

We now analyse the properties of accumulated delay. Due to space limitation, the formal proofs of the lemmas and theorems are omitted.

Let $Ch = \langle t, t', ChP, Q, P\rangle$ be any given CP operation.

1) Timing of CP operations: The following lemma gives an estimation of accumulated delays.

Lemma 1: Assuming service requests are submitted evenly during the period $[t, t']$, we have,

$$AccDelay(Ch) \approx (\frac{1}{2}||Q|| + ||P||) \times (t' - t) \quad (10)$$

■

Note that, in Lemma 1, the time period $[t, t']$ is the period of CP operation. As we will see in Section V, this assumption is valid in the real world.

Let $D(d)$ be the request submission density (or submission rate) at time moment t. Then, we have the following theorem about the relationship between accumulated delay and service request submission density.

Theorem 1: Assuming service requests are submitted evenly during the CP period, we have that

$$AccDelay(Ch) \approx \left(\frac{1}{2}\delta^2 + L_t\delta\right) d_{(t,t')} \quad (11)$$

where $d_{(t,t')}$ is the average request submission density over the CP period $[t, t']$, $\delta = t' - t$, L_t is the average job length for processing service requests around t. ■

Note that given a CP technique, the latency δ of a CP only depends on the size of CP data. If CP operations are performed periodically, for example, once a week or once a day, for a given CP technique, the value of δ is mostly a constant. Therefore, Theorem 1 implies that, according to the measurement of accumulated delays, CP is better to be performed when the submission density $d_{(t,t')}$ is minimal and L_t is also minimal. This matches the common sense, and also the current practice in industry, that CP operations are best performed when the system is at the lowest demand and the lowest workload.

2) Speed of CP: Let S be the speed of CP, i.e. the volume of CP data saved per time unit. We have the following theorem about the relationship between CP speed and the accumulated delay.

Theorem 2: Assuming service requests are submitted evenly during the CP period, we have that

$$AccDelay(Ch) \approx \frac{d_{(t,t')}V}{S} \cdot \left(\frac{V}{2S} + L_t\right) \quad (12)$$

where $V = Vol(ChP)$. ■

Theorem 2 implies that accumulated delay of a CP is $O(1/S)$. This means that the gain of increasing CP speed diminishes rapidly as the speed increases. Given a CP technique, the CP speed is determined by the hardware and software processing power. Investment in hardware and software to perform CP is not cost effective.

D. Comparing TLCP with BCP

Now, we compare TLCP against BCP.

We use $Bulk(t)$ to denoted a BCP starting at time moment t. Then, we have that

$$Bulk(t) = \langle t, t', ChP^B(t), Q^B(t,t'), P^B(t) \rangle, \quad (13)$$

where

$$t' - t = \alpha(Vol(ChP^B(t)) + \beta, \quad (14)$$
$$Q^B(t,t') = Req[t,t'], \quad (15)$$
$$P^B(t) = \{r | RecvT(r) < t < RespT(r)\} \quad (16)$$

In the sequel, we write δ_B to denote $t' - t$ of the *bulk* CP.

For a TLCP for tenant T starting at time moment t, denoted by $TenCh^T(t)$, we have that

$$TenCh^T(t) = \langle t, t', ChP^T(t), Q^T(t,t'), P^T(t) \rangle, \quad (17)$$

where

$$t' - t = \alpha(Vol(ChP^T(t)) + \beta, \quad (18)$$
$$Q^T(t,t') = Req^T[t,t'], \quad (19)$$
$$P^T(t) = \{r | RecvT(r) < t < RespT(r), \; Tenant(r) = T\} \quad (20)$$

Similarly, in the sequel we write δ_T to denote $t' - t$ of the tenant CP $TenCh^T(t)$ of tenant T.

Since $P^T(t) = \{r | r \in P^B(t), Tenant(r) = T\}$, we have

$$P^B(t) = \bigcup_{T \in \Gamma} P^T(t) \quad (21)$$

and, if $T \neq T'$, we have that $P^T(t) \cap P^{T'}(t) = \emptyset$.

By Equation (7), we have that

$$\delta_B = \sum_{T \in \Gamma} \delta_T + ||\Gamma||\Delta. \quad (22)$$

Because $||\Gamma||\Delta$ is negligible in comparison with δ_B, Equation (22) means that the total time of performing CP for all tenants once is approximately equal to the time to perform a BCP.

Let C be a set of CP operations. We extend the notation $AccDelay(ch)$ to $AccDelay(C)$ as follows.

$$AccDelay(C) = \sum_{c \in C} AccDelay(c). \quad (23)$$

Let $TenCh^*(t) = \{TenCh^T(t) | T \in \Gamma\}$, which means to make a CP for every tenant once. The following theorem compares $AccDelay(Bulk(t))$ against $AccDelay(TenCh^*(t))$, which is the sum of the accumulated delays for CP every tenant once.

Theorem 3: Assuming service requests are submitted evenly during the CP periods, we have that

$$\frac{AccDelay(Bulk(t))}{AccDelay(TenCh^*(t))} = O(N), \quad (24)$$

where $N = ||\Gamma||$ is the number of tenants in the system. ■

By Theorem 3, decomposing a large volume of CP data into a number of smaller blocks is an effective way to reduce the accumulated delay. This means that TLCP is a valid solution to CP SaaS.

IV. Empirical Study

This section reports a set of simulation experiments that validate the above theoretical analysis using workload benchmarks of real world large-scale cluster systems.

A. Design of the Experiments

1) The Datasets: The benchmarks used in our experiments are log files in the Parallel Workload Archive [31]. Each log file records the dynamic requests of services submitted to a parallel cluster system in a consecutive period of operation. The five most recent log files from the archive are used; see Table I.

Table I
Log Files Used as Simulation Benchmarks

Name	System	Location	Start time	End time
RICC	RIKEN	Japan	00:04:55, 1/5/2010	23:58:08, 30/9/2010
PoolA	Intel Netbatch, Cluster A	Israel	02:00:01, 1/11/2012	01:59:59, 1/12/ 2012
PoolB	Intel Netbatch, Cluster B	US west coast	17:00:01, 31/10/2012	15:59:59, 30/11/2012
PoolC	Intel Netbatch, Cluster C	US west coast	17:00:01, 31/10/2012	15:59:59, 30/11/2012
PoolD	Intel Netbatch, Cluster D	US west coast	17:00:01, 31/10/2012	15:59:59, 30/11/2012

Table II
Overall Parameters of the Benchmark

Parameter	RICC	PoolA	PoolB	PoolC	PoolD
Tenants	121	31	20	33	20
Duration (days)	183	30	31	31	31
Valid reqs (K)	448	10,980	6,576	12,901	8,449
Reqs / day (K)	2.4	366.0	212.1	416.1	272.5
Reqs/day/tenant(K)	0.02	11.81	10.60	12.61	13.63

Each log file contains a sequence of jobs submitted to the cluster. The following data in the log files are used:
- *Job Number*: A unique identifier of a request.
- *Group ID*: A unique tenant ID of a job.
- *Submit Time*: The time moment when the job is submitted.
- *Run Time*: The length of time used to process the job.
- *Requested Memory*: the required size of memory in KB.
- *Number of processors*: the number of processing units used.
- *Memory used*: the memory space used per processor.

Table II gives the information about the benchmarks. Strictly speaking, these benchmarks are not log files of SaaS applications, which are not available as far as we know. However, they reflect the operations of large-scale cluster systems and provide credible data to validate the theoretical model.

2) Simulation Algorithm: The simulation program calculates the following data from the log file:
- *CP volume*: It is the sum of memory space used by service requests between two consecutive CP operations.
- *Length of CP time*: For a given CP speed S, the CP time is calculated from the volume of the CP according to Equation

(8), where $\alpha = 1/S$ and $beta$ is set to be 0 since it is negligible.
- *Number of requests blocked*: It is the number of requests submitted during the CP operation.
- *Submission time of requests blocked*: The submission times of those blocked requests are recorded together with their tenant identifiers.
- *Number of requests paused*: It is the number of requests submitted before a CP operation but unfinished when the CP operation starts.
- *Accumulated Delay*: The accumulated delay of a CP is calculated according to the definition of accumulated delay.

3) Simulation Parameters: The simulation program takes a set of parameters as input, which include:
- *CP Interval*. In the simulations, both bulk and tenant CP take place periodically. The CP Interval is the interval between two consecutive CP operations.
- *CP Start Time*. It is the time when a CP operation starts. It is set at the time when the system's workload and demand are relatively low.
- *CP Speed*. It is the speed that CP data are collected and saved. For each simulation, it is set to allow a BCP to finish within a few hours.

The simulation produced the data of BCP and tenant CP operations. From these data, statistical analysis is performed.

B. Main Results of The Experiments

The results of the experiments demonstrate that the main results of theoretical analysis are consistent with the empirical data.

1) The Validity of the Assumption Underlying Lemma 1: For each CP operation, we compare the accumulated delay obtained directly in simulation against the estimation calculated using Equation (10). The differences between them are calculated according to the formula $(R - E)/R$, where R and E are the real and the estimated accumulated delay, respectively. The results are summarised in Table III.

Table III
SUMMARY OF THE ANALYSIS OF HYPOTHESIS 1

Benchmark	Type	# ChPs	Avg Diff (%)	St Dev Diff
RICC	Bulk	20	-0.19	0.0209
PoolA	Bulk	28	3.07	0.0276
PoolB	Bulk	28	0.15	0.0279
PoolC	Bulk	28	-0.98	0.0492
PoolD	Bulk	28	1.09	0.0298
RICC	Tenant	164	0.24	0.0245
PoolA	Tenant	334	-0.43	0.1417
PoolB	Tenant	214	-0.20	0.0317
PoolC	Tenant	238	0.01	0.0294
PoolD	Tenant	345	0.05	0.0392

Table III shows that, for both BCP and TCP operations, the average difference between real accumulated delays and estimated accumulative delays are very small percentages. Moreover, for each set of CP operations, the standard deviation of the differences is also very small. This means that a concentrated distribution of the differences occurs around the centre as shown in Figure 2.

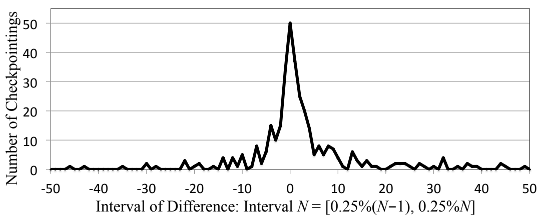

Figure 2. Distribution of the Diffs between Real and Estimated AccDelays

2) Validity of The Power Law of CP Speed: Theorem 2 predicted a power law of the effect of CP speed on accumulated delays. In the experiments, we simulated CP operations with various CP speeds using the Intel Netbatch Pool-A benchmark. The observed changes in the accumulated delay are shown in Figure 3.

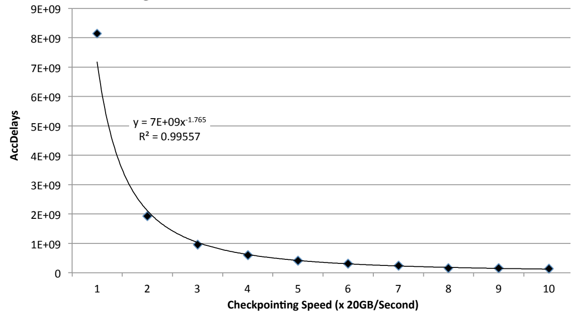

Figure 3. Power Law of Accumulated Delay w.r.t. Speed

The simulation data clearly shows that the relationship between CP speed and the accumulated delay follows a power law, where the trend line in Figure 3 is a function $f(x) = 7 \times 10^9 x^{-1.765}$ with $R = 0.99557$.

3) Comparison of TCP with BCP: Theorem 3 predicted that TCP outperforms BCP in the order of $O(N)$. Our benchmark contains fixed numbers of tenants. Thus, Theorem 3 cannot be validated directly. Therefore, we calculated the following ratio for each CP:

$$\frac{AccDelay(Bulk(t))}{AccDelay(TenCh^*(t))}. \tag{25}$$

The results are summarised in Table 4, where column "#A.T." is the average number of active tenants. Columns "Min" and "Max" are the minimal and maximal values of the ratios. Columns "Avg" and "St Dev" are the average values of the ratios and their standard deviations.

Table IV
RATIOS OF BULK OVER TENANT CP ACCDELAYS

Benchmark	#A.T.	Min	Max	Avg	St Dev
RICC	49.35	2.38	380.65	47.3	84.86
PoolA	21.11	3.64	9.9	6.85	1.49
PoolB	13.68	2.89	6.25	4.00	0.71
PoolC	15.11	2.65	20.59	4.37	3.53
PoolD	15.21	2.9	16.89	10.6	3.29

Table IV shows that the ratios vary in a wide range. In general, TLCP significantly reduces the impact of CP on system's performance. It has not achieved its best performance on these benchmarks because the benchmarks are not from a real SaaS application.

V. Conclusion

This paper presents a theoretical analysis and empirical study for the evaluation of TLCP. Both results clearly demonstrated the advantages of TLCP over traditional BCP as a fault tolerant facility for SaaS applications. In particular, TLCP can significantly reduce the interruption to the operation of a SaaS application.

The simulation experiments generated a large volume of data that enable us to observe the performances of CP without actually owning the hardware and software systems of large scale. It also enables us to explore various parameters of the system to observe their impacts on various aspects of performances. More details of the observations that we have made in the experiments will be reported separately.

As a future work, we are studying the mechanisms to optimise the timing of CP so that every tenant can perform TLCP operations with minimal accumulated delays.

References

[1] T. Kraska and B. Trushkowsky, The New Database Architectures, *IEEE Internet Comput*, vol.17, no.3, pp.72-75, May 2013.

[2] T. Kwok, T. Nguyen, and L. Lam, A Software as a Service with Multi-tenancy Support for an Electronic Contract Management Application, in *Proc. of ICSC 2008*, pp.179-186.

[3] W. T. Tsai, X. Y. Bai and Y. Huang, Software-as-a-service (SaaS): perspectives and challenges . *Sci China Inf Sci*, May, 2014, Vol. 57, pp. 1-15.

[4] H. Koziolek, The SPOSAD Architectural Style for Multi-tenant Software Applications, in *Proc. of ICSA 2011*, pp. 320-327.

[5] B. Yousef, H. Zhu, and M. Younas, Tenant Level Checkpointing of Meta-data for Multi-tenancy SaaS, in *Proc. of SOSE 2014*, pp. 148-153.

[6] I. Jangjaimon and N.-F. Tzeng, "Design and implementation of effective checkpointing for multithreaded applications on future clouds," in *Proc. of IEEE CLOUD 2013*, pp. 438-445.

[7] A. Agbaria and R. Friedman, Virtual-machine-based heterogeneous checkpointing, *Softw. - Pract. & Exp.* vol.32, no.12, pp.1175-1192, 2002.

[8] T.C. Bressoud and F.B. Schneider, Hypervisor-based fault tolerance, *ACM Trans. Comput. Syst.* vol. 14, no. 1, pp. 80-107, Feb. 1996.

[9] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, Checkpointing virtual machines against transient errors, in *Proc. of 2010 IEEE 16th International On-Line Testing Symposium*, 2010, pp. 97-102.

[10] B. Nicolae and F. Cappello, BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots, in *Proc. of Sc 2011*, pp. 34:1-34:12.

[11] T. Bressoud and M. Kozuch, "Cluster fault-tolerance: An experimental evaluation of checkpointing and mapreduce through simulation," in *Proc. of CLUSTER 2009*, pp. 1-10.

[12] B. Azeem and M. Helal, "Performance evaluation of checkpoint/restart techniques: For MPI applications on Amazon cloud," in *Proc. of INFOS 2014*, pp. PDC49-PDC57.

[13] D. Hakkarinen and Z. Chen, "Multilevel diskless checkpointing," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 772-783, 2013.

[14] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *Proc. of CCGrid 2010*, pp. 63–72.

[15] G. Belalem and S. Limam, "Fault tolerant architecture to cloud computing using adaptive checkpoint," *International Journal of Cloud Applications and Computing*, vol. 1, no. 4, pp. 60–69, Oct.-Dec. 2011.

[16] I. Jangjaimon and N.-F. Tzeng, "Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 396–409, 2015.

[17] D. Sun, G. Chang, C. Miao, and X. Wang, "Building a high serviceability model by checkpointing and replication strategy in cloud computing environments," in *Proc. of ICDCSW 2012*, pp. 578–587.

[18] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-L. Wang, and F. Cappello, "Optimization of cloud task processing with checkpoint-restart mechanism," in *Proc. of SC 2013*, pp. 64:1–64:12.

[19] H. Jin, T. Ke, Y. Chen, and X.-H. Sun, "Checkpointing orchestration: Toward a scalable hpc fault-tolerant environment," in *Proc. of CCGrid 2012*, pp. 276–283.

[20] V. H. Ha and E. Renault, "Improving performance of cape using discontinuous incremental checkpointing," in *Proc. of HPCC 2011*, pp. 802–807.

[21] Y. Liu, H. Zhu, Y. Liu, F. Wang, and B. Fan, "Parallel compression checkpointing for socket-level heterogeneous systems," in *Proc. of HPCC 2011*, pp. 468–476.

[22] X. Yang, Z. Wang, J. Xue, and Y. Zhou, "The reliability wall for exascale supercomputing," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 767–779, Jun. 2012.

[23] P. Lu, B. Ravindran, and C. Kim, "VPC: Scalable, low downtime checkpointing for virtual clusters," in *Proc. of SBAC-PAD 2012*, pp. 203–210.

[24] A. Colesa, I. Stan, and I. Ignat, "Transparent fault-tolerance based on asynchronous virtual machine replication," in *Proc. of SYNASC 2010*, pp. 442–448.

[25] H. Liu, H. Jin, X. Liao, C. Yu, and C.-Z. Xu, "Live virtual machine migration via asynchronous replication and state synchronization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 12, pp. 1986–1999, 2011.

[26] H. Goudarzi and M. Pedram, "Energy-efficient virtual machine replication and placement in a cloud computing system," in *Proc. of CLOUD 2012*, pp. 750–757.

[27] F. Xu, F. Liu, L. Liu, H. Jin, B. Li, and B. Li, "iaware: Making live migration of virtual machines interference-aware in the cloud," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 3012–3025, 2014.

[28] D. Singh, J. Singh, and A. Chhabra, "High availability of clouds: Failover strategies for cloud computing using integrated checkpointing algorithms," in *Proc. of CSNT2012*, pp. 698–703.

[29] S. Mondal and J. Muppala, "Energy modeling of virtual machine replication schemes with checkpointing in data centers," in *Proc. of BdCloud 2014*, pp. 633–640.

[30] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud," in *Proc. of CLOUD 2010*, pp. 236–243.

[31] F. Dror, Parallel Workloads Archive, 2005. [Online]. http://www.cs.huji.ac.il/labs/parallel/workload/.[Accessed:Nov-2014].