Optimization of Cloud Task Processing with Checkpoint-Restart Mechanism

Sheng Di[1,2], Yves Robert[3,4], Frédéric Vivien[3], Derrick Kondo[5], Cho-Li Wang[6], Franck Cappello[1]
1. Argonne National Laboratory, USA, cappello@mcs.anl.gov
2. INRIA, Saclay, France, sheng.di@inria.fr
3. ENS Lyon and INRIA, France, Yves.Robert@ens-lyon.fr, Frederic.Vivien@inria.fr
4. University of Tennessee Knoxville, USA
5. INRIA, Grenoble, France, derrick.kondo@inria.fr
6. The University of Hong Kong, Hong Kong, clwang@cs.hku.hk

ABSTRACT

In this paper, we aim at optimizing fault-tolerance techniques based on a checkpointing/restart mechanism, in the context of cloud computing. Our contribution is three-fold. (1) We derive a fresh formula to compute the optimal number of checkpoints for cloud jobs with varied distributions of failure events. Our analysis is not only generic with no assumption on failure probability distribution, but also attractively simple to apply in practice. (2) We design an adaptive algorithm to optimize the impact of checkpointing regarding various costs like checkpointing/restart overhead. (3) We evaluate our optimized solution in a real cluster environment with hundreds of virtual machines and Berkeley Lab Checkpoint/Restart tool. Task failure events are emulated via a production trace produced on a large-scale Google data center. Experiments confirm that our solution is fairly suitable for Google systems. Our optimized formula outperforms Young's formula by 3-10 percent, reducing wall-clock lengths by 50-100 seconds per job on average.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Fault tolerance; D.4.5 [**Operating Systems**]: Reliability—*Checkpoint /restart, Fault-tolerance*

General Terms

Theory, Reliability, Experimentation, Performance

Keywords

Cloud Computing, Checkpoint-Restart Mechanism, Optimal Checkpointing Interval, Google, BLCR

1. INTRODUCTION

SC13, November 17 - 21 2013, Denver, USA

http://dx.doi.org/10.1145/2503210.2503217.

Cloud computing [1] is becoming a compelling paradigm in provisioning elastic services and on-demand resources. The cloud model investigated in this paper is based on Platform-as-a-Service (PaaS) [1], where users can compose complex *requests* (or *jobs*) based on off-the-shelf web services. Each job could be made up of one or more tasks and each task execution requires multiple types of resources. Each task is executed in a particular virtual machine (VM) instance, whose resources (such as CPU rate, memory size) are isolated via virtual resource isolation technology [2, 3].

With the fast advance of web applications, fault tolerance has become a fairly serious issue in cloud computing. On the one hand, more and more high performance computing (HPC) applications are being developed over cloud platforms. Evangelinos et al. [4], for example, confirmed the feasibility of running coupled atmosphere-ocean climate models on Amazon EC2 [5]. Nurmi et al. [6] made use of VMs to build a cloud platform (called EUCALYPTUS) allowing Grid users to run HPC applications. In general, large-scale systems always face more or less fault-tolerance problems. For example, the BlueGene/L system with 100k nodes at Lawrence Livermore National Laboratory (LLNL) experiences an L1 cache bit error every 8 hours [7], and a hard failure every 7-10 days. On the other hand, over-commitment of physical resources is very common in cloud systems, in order to achieve high resource utilization. According to a Google trace [8, 9] with 10k+ hosts, for example, Reiss. et al. [10] showed that the requested resource amounts are often greater than the total capacity of Google data centers. Such an over-commitment may cause exhaustion of physical resources and, eventually, may lead to kill or eviction events for low priority tasks [9].

In comparison to the traditional HPC/Grid platforms, fault tolerance issue in the context of cloud computing faces at least two new challenges. (1) Cloud jobs are much smaller than Grid jobs (as reported by Di's work [11] based on Google trace [9]), so that cloud job execution time is more sensitive to the impact of checkpointing/restart cost. (2) The probability of failure occurrences for cloud jobs may largely differ from that for HPC/Grid jobs. This is due to the fact that cloud resources are often allocated with more restrictions, like user payments and task priorities. For instance, Yi et al. [12] shows that the failure probability density function (PDF) of Amazon cloud spot instances is not only dependent on task length but is also related to user bids. Cirne et al. [13] showed that the failure probability of Google

jobs, observed through real-world production traces, follows a distribution with a saw-tooth curve. These observations contrast the failure probability distribution of HPC/Grid jobs [14, 15, 16], which is characterized as a rather simple independent and identical distribution (IID).

In this paper, we mainly answer the four following questions:

- Based on the characterization of checkpointing/restart cost and on the statistics of failure events of **cloud** tasks, how to optimize the number of checkpoints for each task? Since cloud task failure events may not simply depend on a particular probability distribution, we have to analyze the issue with no assumption on probability distribution. This contrasts the traditional analysis like Young's work [17] or Daly's work [14], which assumes that failure probability follows an Exponential distribution. In this paper, we derive a novel succinct formula to compute the optimal number of checkpointing intervals. We also prove that Young's formula [17] can be considered a particular case of our new formula.

- How to dynamically tune the optimal solution with the checkpoint/restart mechanism at runtime, in order to adapt to possible changes of the failure probability distribution? In the context of cloud computing, task failure probabilities may change over time. For instance, if a cloud user dynamically changes the bid on service instances in Amazon EC2 [5], or the priority on a Google cloud data center [9], the failure probability changes accordingly.

- How to optimize the tradeoff between checkpointing a task memory on local disks or on a shared disk? Checkpointing on shared disks leads to higher reliability and flexibility, as it avoids the impact of the local node failures and supports implicit process migration. However, this approach incurs heavier checkpointing costs and possible bottleneck problems. In contrast, storing memory states in a local disk reduces the checkpointing costs, but incurs heavier process migration costs. Based on Berkeley Lab Checkpoint/Restart (BLCR) [20], for example, before restarting a failed task on another host, one has to transfer the task memory from the (ram)disk of its last execution host to the disk of the current host, introducing extra disk read/write costs.

- What are the experimental results like, when applying our dynamic optimized fault-tolerant method on a real-cluster environment deployed with VMs? We evaluate this method on a real cluster environment deployed with XEN's hypervisor [18] and BLCR [20], unlike other traditional work that performs evaluation only by simulation or hypothetical cases. We reproduce the overall benchmark based on Google's one-month production trace [9], in which each job contains one or more tasks (such as bag-of-tasks like Mapreduce [19]). Experiments confirm that our fault-tolerant solution can effectively improve the workload processing ratio by 3-10 percent.

The rest of this paper is organized as follows. In Section 2, we provide an overview of our cloud model and introduce the fault-tolerance mechanism. We formulate our research problem in Section 3. In Section 4, we detail the derivation of the optimal number of checkpointing intervals in the context of cloud computing, as well as the adaptive solution with minimized execution cost. We present experimental results in Section 5. We discuss related work in Section 6. Finally, we provide concluding remarks and hints for future work in Section 7.

2. SYSTEM OVERVIEW

The system architecture of our fault-tolerant cloud platform is shown in Figure 1, which is basically consistent with most cloud models, e.g., Google task execution model [9]. A user request (a.k.a., a job) is made up of one or more tasks, each of which is an instance of a cloud service (or online application). Job scheduling layer is used to coordinate the job priorities, so that they can be treated in a fair way, or the overall system can work quite efficiently. Resource allocation layer is responsible for allocating resource fractions for cloud tasks based on specific demands, and for performing resource isolation by hypervisor on selected VMs if needed.

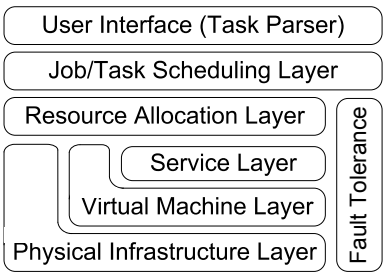


Figure 1: System Architecture of Composite Cloud Service System.

The overall cloud system is often organized in a loosely-coupled way. For example, Google web search system [21] covers 10k+ servers worldwide, each comprising hundreds of PC commodity machines. The fault tolerance on each server is autonomously organized for purpose of high reliability and performance. Accordingly, our research is based on a particular large-scale data center with many execution hosts.

In our cloud model, each job is processed according to the procedure illustrated in Figure 2. At the beginning, a job is submitted and analyzed by *job parser*, in order to predict the job workload based on its input parameters. Recently, many effective workload prediction methods were proposed, like the polynomial regression method [22]. As there are available resources, one unprocessed task will be selected and put in a pending queue, waiting for the scheduling notification with a selected qualified host and a VM instance running atop it. In our experiments, the physical host with the maximum available memory size will be selected. This scheduling policy is to account for the specular features of Google jobs, to be further discussed later. The corresponding hypervisor will perform the resource isolation for the selected VM to match the resource customization. The computing result will be cached in the VM, in case of data transmission to a succeeding task.

In our design, we use three specific threads to periodically check the liveness of each physical host, VM, and task run-

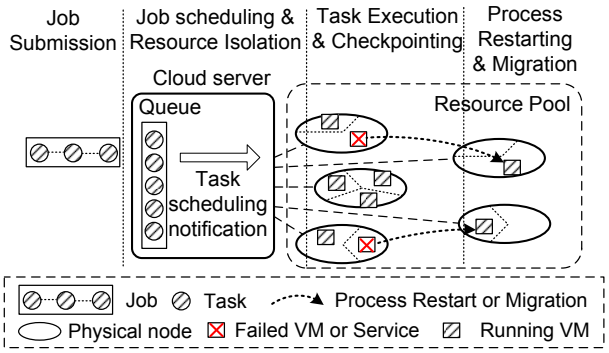


Figure 2: Cloud Job Processing Procedure.

ning process. For example, if a host is down, all the tasks running on the VMs of this host (as recorded in the scheduling queue) will be immediately restarted on other hosts from their most recent checkpoints. For any task restoration, a new thread (called restoring thread) will be launched to handle the restoration. We intentionally do not checkpoint an entire VM's state at runtime, but instead just the transient memory of running tasks, because of the heavy overhead in checkpointing VMs [23]. In particular, BLCR [20] is used to periodically store the memory of running tasks in local disks, or shared disks like Network File System (NFS). When an interrupted task is detected, its execution can be restarted on an idle VM, using its memory state stored in its most recent checkpoint.

3. PROBLEM FORMULATION

There are n jobs in the system, denoted by J_i for i=1,2,$\cdots$,n. Each job is made up of one or more tasks. We denote the tasks of J_i by $t_{i(1)}$, $t_{i(2)}$, $\cdots$, $t_{i(m_i)}$, where m_i is the number of tasks in job J_i. The execution time (a.k.a., productive time, excluding the time lost due to checkpoints and failures) of task $t_{i(j)}$ is denoted by $T_e(t_{i(j)})$.

Statistics of a Google trace with millions of tasks [8] indicate that cloud jobs are likely to encounter failure/interruption events. Thus, it is necessary to checkpoint cloud jobs from time to time. Thanks to outstanding checkpoint/restart tools like BLCR software [20], equidistant checkpointing is viable. That is, we can take checkpoints (storing any task memory) at any time in the course of its execution. We formulate the fault-tolerance research in the context of cloud computing as an optimization problem of equidistant checkpointing.

Equidistant checkpointing aims at determining the optimal number of same-length checkpointing intervals for a task, in terms of the probability distribution of failure events for that task. We suppose that the number of failure events in a task $t_{i(j)}$ follows a probability distribution $P_{t_{i(j)}}$. That is, the probability of task $t_{i(j)}$ encountering K failure/interruption events is denoted as $P_{t_{i(j)}}(Y = K)$. We use $T_h(t_{i(j)})$ to denote the date of task $t_{i(j)}$'s h-th failure event, and use $\Lambda(T_h(t_{i(j)}))$ to denote the checkpointing position which is before and closest to the date $T_h(t_{i(j)})$. According to our characterization, the *checkpointing cost* (defined as the increment of the task wall-clock time due to one checkpoint) is determined by the task memory size (to be discussed later). Thus, the checkpointing cost is relatively stable for any given task. Likewise, the cost for restarting a task (called *task restarting cost*) is also constant in most cases. As a consequence, we define the checkpointing cost and the task restarting cost as two constants with respect to a particular task, denoted by C and R respectively. Considering the total overhead due to task failure events and checkpointing/restart costs, the total wall-clock time (a.k.a., wall-clock length) of a task $t_{i(j)}$ that encounters K failure events can be represented as Formula (1), where x refers to the number of checkpointing intervals. Here $T_h(t_{i(j)}) - \Lambda(T_h(t_{i(j)}))$ refers to the time wasted on the rollback of the task execution to its closest checkpoint. This formula means that a task total wall-clock time is equal to its execution time (a.k.a., productive time) in processing its workload, plus the total overhead of taking checkpoints and the total time cost of the rollbacks of task execution upon failure events:

$$\begin{aligned} T_w(t_{i(j)}) =& T_e(t_{i(j)}) + C \cdot (x-1) \\ &+ \sum_{h=1}^{K} (T_h(t_{i(j)}) - \Lambda(T_h(t_{i(j)})) + R) \end{aligned} \tag{1}$$

Our objective is to compute the optimal number of checkpointing intervals for minimizing a task expected wall-clock time, when we set equidistant checkpoints. A task expected wall-clock time could be written as Formula (2), where we omit the notation $t_{i(j)}$ for simplicity of expression, e.g., T_e and T_h refer to $T_e(t_{i(j)})$ and $T_h(t_{i(j)})$ respectively.

$$\begin{aligned} &E(T_w(t_{i(j)})) \\ &= \sum_{K=0}^{\infty} \left(P_{t_{i(j)}}(Y=K) \cdot (T_e + C(x-1) + \sum_{h=1}^{K} (T_h - \Lambda(T_h) + R)) \right) \end{aligned} \tag{2}$$

We summarize key notations in Table 1.

Table 1: Summary of Key Notations.

Notation	Description
n	number of jobs
J_i	a user request that is made up of multiple tasks
$t_{i(j)}$	the jth task in the job J_i
$P_{t_{i(j)}}(Y=K)$	probability of K failure events striking $t_{i(j)}$
$T_h(t_{i(j)})$	date of the hth failure event of task $t_{i(j)}$
$\Lambda(T_h(t_{i(j)}))$	checkpointing position that is before & closest to $T_h(t_{i(j)})$
C	checkpointing cost (per checkpoint)
R	time cost when restarting a failed task
$T_e(t_{i(j)})$	execution time of $t_{i(j)}$ without failure events, also excluding checkpointing costs
$T_w(t_{i(j)})$	wall-clock time of $t_{i(j)}$, including all costs induced by failure events and fault tolerance

4. OPTIMIZING FAULT TOLERANCE FOR CLOUD TASKS

In this section, we compute the optimal number of checkpointing intervals based on the equidistant checkpointing model and introduce an adaptive algorithm to minimize execution cost regarding checkpointing/restart overhead.

4.1 Optimizing the Number of Checkpoints

For the sake of simplicity, we omit the notation $t_{i(j)}$ in the following text. For instance, we use T_w, $P(Y{=}K)$, and T_h, to represent respectively $T_w(t_{i(j)})$, $P_{t_{i(j)}}(Y{=}K)$, and $T_h(t_{i(j)})$.

THEOREM 1. *Based on the problem formulation in Section 3,* ***if*** *we set checkpoints evenly during a task execution,*

then *the optimal number (x^*) of checkpointing intervals is given by Equation (3), where $E(Y)$ denotes the expected number of failure events occurring during the execution of the task.*

$$x^* = \sqrt{\frac{T_e \cdot E(Y)}{2C}} \tag{3}$$

PROOF. $E(T_w)$

$$=\sum_{i=0}^{\infty}\left(P(Y=i)\cdot(T_e+C(x-1)+\sum_{j=1}^{i}(T_j-\Lambda(T_j)+R))\right)$$
$$=(T_e+C(x-1))\sum_{i=0}^{\infty}P(Y=i)$$
$$+\sum_{i=0}^{\infty}\Big(P(Y=i)\cdot(\sum_{j=1}^{i}(T_j-\Lambda(T_j))+iR)\Big)$$
$$=(T_e+C(x-1)+R\cdot E(Y))+\sum_{i=0}^{\infty}\left(P(Y=i)\cdot\sum_{j=1}^{i}(T_j-\Lambda(T_j))\right)$$

Let us analyze the expected value of $\sum_{j=1}^{i}(T_j-\Lambda(T_j))$. Checkpoints are set evenly during a task execution. Because T_e denotes the total execution length in the absence of fault-tolerance mechanisms and failures, a checkpoint is taken once the execution of the task has progressed for a duration $\frac{T_e}{x}$ without encountering any failure event. This is exemplified by Figure 3, where x=4 in this example.

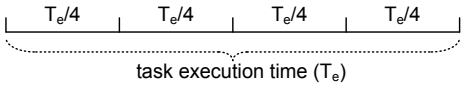


Figure 3: Segment Intervals Separated by Checkpoints.

Once a task is victim of a failure event, it is either restarted immediately using an idle VM, or it is restarted later when some resources become available. That is, a task may not be restarted immediately when it encountered a failure event due to the nature of cloud computing. In addition, task failure events are often attributed to multiple complex factors. Hence, checkpointing dates and failure events are independent. Therefore, the expected time loss of the rollback due to a failure event is $\frac{T_e}{2x}$, because a failure event must happen in an interval of length $\frac{T_e}{x}$. Hence, $E(T_w)$ can be rewritten as Equation (4).

$$\begin{aligned}&E(T_w)\\&=(T_e+C(x-1)+R\cdot E(Y))+\sum_{i=0}^{\infty}\Big(P(Y=i)\cdot\tfrac{i\cdot T_e}{2x}\Big)\\&=(T_e+C(x-1)+R\cdot E(Y))+\tfrac{T_e}{2x}\sum_{i=0}^{\infty}(i\cdot P(Y=i))\\&=(T_e+C(x-1)+R\cdot E(Y))+\tfrac{T_e}{2x}E(Y)\end{aligned} \tag{4}$$

Since $\frac{\partial^2 E(T_w)}{\partial x^2}=\frac{T_e E(Y)}{2x^3}>0$, $E(T_w)$ has a minimum extreme point when $\frac{\partial E(T_w)}{\partial x}=0$. Accordingly, we can compute the optimal value (x^*) by solving Equation (5).

$$\frac{\partial E(T_w)}{\partial x} = C - \frac{T_e E(Y)}{2x^2} = 0 \tag{5}$$

This leads to $x^*=\sqrt{\frac{T_e E(Y)}{2C}}$. □

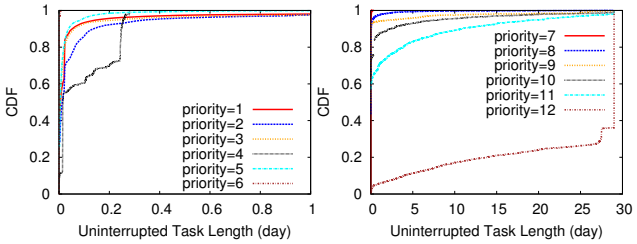
Remarks:

- We give an example to illustrate the theorem. Suppose that the task execution length is T_e=18 seconds and that checkpointing cost is C=2 seconds. If the number of failure events follows a Poisson distribution where $P(Y=k) = \frac{\lambda^k}{k!}e^{-\lambda}$ and the failure rate λ is equal to 2, then, $E(Y)=\lambda=2$. Hence, the optimal number of checkpointing intervals for the task is $\sqrt{\frac{18\times 2}{2\times 2}}=3$. That is, the optimal solution is to take a checkpoint every $\frac{18}{3}$=6 seconds during the execution of the task. In practice, the expected number of failures for a task can be estimated by the mean number of failures (MNOF) of the task, and MNOF can be estimated with the statistics computed based on history.

- Note that our theoretical conclusion does not depend on any probability distribution, unlike Young's formula [17] which needs to assume that failure intervals follow an exponential distribution. Young's formula is given by Equation (6), where T_c, C, and T_f refer to the optimal checkpointing interval, checkpointing cost, and the mean time between failures (MTBF) respectively.

$$T_c = \sqrt{2\ C\ T_f} \tag{6}$$

That is, Theorem 1 advances a more generic formula without any assumption on the probability distribution. This is significant because different types of cloud tasks are likely to be victim of different distributions of failure intervals. For example, Figure 4 presents, with respect to the different task priorities in Google systems, the cumulative distribution function (CDF) of the uninterrupted work intervals of a given task. One can observe that the distributions of uninterrupted intervals are quite different for the various task priorities (from 1 to 12). Tasks with higher priorities tend to have longer uninterrupted execution lengths, because low-priority tasks tend to be preempted by high-priority ones.



(a) w.r.t. low-priority tasks (b) w.r.t. high-priority tasks

Figure 4: Distribution of Google Task Failure Intervals According to Priorities.

- In fact, Corollary 1 proves that Young's formula is a particular case of Theorem 1.

COROLLARY 1. ***If*** *the failure intervals of a task follow an exponential distribution and if the checkpointing cost is small,* ***then*** *Young's formula can be derived from Formula (3).*

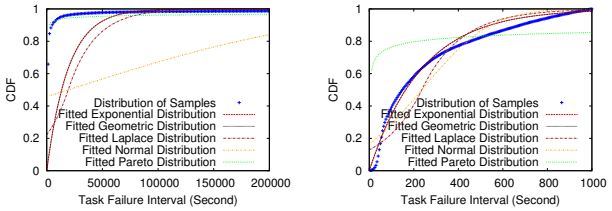
PROOF. We denote the Mean Time Between Failures (MTBF) of a task by T_f. Because the failure events striking a given task occur independently and because their consecutive intervals follow an exponential distribution, it can be proven that the number of failures must follow a Poisson process with the expected number of occurrences per time unit being equal to $\frac{1}{T_f}$. Then, the expected number of task failure events during its productive period can be approximated

as $E(Y) \approx T_e \times \frac{1}{T_f} = \frac{T_e}{T_f}$ if the checkpointing cost and the expected number of failures are small with respect to T_e. Then, we can get Young's formula based on the following derivation.

$$T_c = \frac{T_e}{x^*} = \frac{T_e}{\sqrt{\frac{1}{2}T_e E(Y)/C}} = \frac{T_e}{\sqrt{\frac{1}{2}T_e \cdot \frac{T_e}{T_f}/C}} = \sqrt{2CT_f} \quad \square$$

- Formula (3) is easier to apply than Young's formula in many situations. Note that Young's formula relies on the distribution of failure intervals. In general, it is non-trivial to record the accurate time stamps of failure events due to many factors like non-synchronous clocks across hosts (failed tasks may be restarted on a host using a different clock), inevitable influence of checkpointing cost, or significant delay of failure detection. In contrast, it is easy to record the number of failures striking a particular task. That is, Formula (3), which depends on the expected number of failure events hitting a task, is easier to apply.

- We give a practical example summarized from a one-month Google trace [8]. We derive the optimal checkpointing intervals for the execution of a Google task using both Theorem 1 and Corollary 1. Figure 5 presents the CDF of the uninterrupted work intervals of a task in the Google trace, as well as some well-known distribution curves fitted with maximum likelihood estimate (MLE).



(a) all failure intervals (b) short failure intervals

Figure 5: Overall Distribution of Google Task Failure Intervals and Distribution Fitting with MLE.

We find that a Pareto distribution fits the sample distribution best in general. However, since a large majority (over 63%) of task failure intervals last for less than 1000 seconds, according to our characterization, one should be more interested in small tasks. If we just consider failure intervals within 1000 seconds, the best-fit distribution is an exponential distribution with failure rate λ=0.00423445 (although not perfect, as shown by the discrepancies in Figure 5 (b)). Hence, in this situation, suppose the checkpointing cost is always 2 seconds. Then the optimal checkpointing interval for a relatively small cloud task (task length $\leq$1000 seconds) can be estimated as $\sqrt{2C\frac{1}{\lambda}} = \sqrt{2 \times 2 \times \frac{1}{0.00423445}} \approx$ 30.7 seconds, in terms of Corollary 1.

4.2 Adaptive Optimization of Fault Tolerance

So far, we have investigated the optimal number of checkpointing intervals for a cloud task, by setting checkpoints evenly during the task execution. However, the failure probability distribution a task is subject to depends on the task priority. Therefore, if the priority of a task changes during its execution, so does the failure probability distribution. Thus, we propose an adaptive solution that recomputes checkpointing dates in order to cope with potential changes at runtime in the failure probability distribution.

Algorithm 1: ADAPTIVE CHECKPOINTING ALGORITHM

Input: task $t_{i(j)}$, checkpointing cost C, task execution time T_e

```
Estimate the checkpointing/restart tradeoff between using shared-disk and local-disk  /* Section 4.2.2 */
Select the device used to store memory based on the optimal estimation
Compute X* based on Formula (3)
W0 ← Te/X*;        W ← W0
repeat
    if (W ≤ 0) then
        Take a checkpoint for task t_i(j) via a new thread
        Te ← Te − W0
        if (MNOF changed) then
            Compute a new X* based on Formula (3)
            W0 ← Te/X*
        W ← W0
    W ← W − Δt                                  /* countdown */
    Sleep a tiny period Δt
until task t_i(j) is completed
```

We also compute the most efficient checkpointing/restart approach when one can store checkpoints either in local ramdisks[1] or in shared disks. We present the pseudo-code in Algorithm 1.

At the beginning of Algorithm 1, we compute the first checkpoint with Formula (3) using the task predicted execution workload[2] and the corresponding mean number of failures (*MNOF*). During the task execution, the algorithm will periodically check whether it is time to take a checkpoint through a countdown mechanism. This step can also be implemented easily with a notify/wait mechanism, instead of a polling method.

In the following text, we mainly focus on two key issues: in what situation the checkpointing dates (i.e., optimal checkpoint positions) should be updated, and whether to use local ramdisks or shared disks to store checkpoints.

4.2.1 Dynamic Optimization of Checkpointing Positions

We prove in Theorem 2 that the next checkpoint position needs to be recomputed, if and only if the task MNOF (i.e., $E(Y)$ in Theorem 1) is changed during the previously last checkpoint interval (probably due to the credits or priorities tuned by users). That is, although the remaining workload of a task decreases over time, the optimal checkpointing dates remain actually the same if the task MNOF remains unchanged. More specifically, if the factor (e.g., task priority) that may influence the task MNOF remains unchanged, the optimal checkpointing positions would be unchanged for its remaining execution.

[1] RAMDisk creates a virtual RAM drive, or block of memory, which the computation host treats as if it were a disk drive.

[2] A task workload can be predicted by prediction methods like polynomial regression [22] or the estimation based on history [25].

THEOREM 2. *Next optimal checkpointing position will be changed for the task remaining workload (i.e., remaining execution length),* ***if and only if*** *the task MNOF is changed in the previously last checkpointing interval.*

PROOF. Without loss of generality, suppose that the current checkpointing position is the $(k+1)$st checkpoint and its preceding one is the kth checkpoint, and their remaining execution lengths are denoted as $T_r(k+1)$ and $T_r(k)$ respectively. We denote the optimal number of checkpointing intervals computed at the kth and $(k+1)$st checkpoints as X^* and $X^{(*)}$ respectively. We illustrate the notations in Figure 6. In the following text, we will prove $X^{(*)}=X^*-1$.

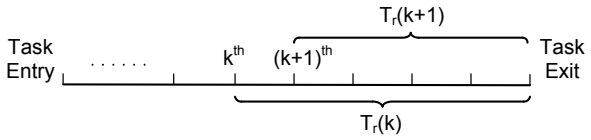


Figure 6: Illustration of Notations used in the Proof.

According to Theorem 1, we can represent the optimal number of checkpointing intervals at the two positions in Formula(7) and Formula(8) respectively, where $E_k(Y)$ denotes the expected number of failures in the task remaining execution time $T_r(k)$. Obviously, $E_0(Y)=E(Y)=$MNOF.

$$X^* = \sqrt{\frac{T_r(k)E_k(Y)}{2C}} \tag{7}$$

$$X^{(*)} = \sqrt{\frac{T_r(k+1)E_{k+1}(Y)}{2C}} \tag{8}$$

Since each computation of checkpoints is based on equidistant checkpointing model, we can get $T_r(k+1)=T_r(k)\cdot\frac{X^*-1}{X^*}$, which can also be derived from Figure 6.

Based on the definition of $E_k(Y)$, we can derive $E_k(Y)=\frac{T_r(k)}{T_r(0)}E_0(Y)=\frac{T_r(k)}{T_r(0)}$MNOF and $E_{k+1}(Y)=\frac{T_r(k+1)}{T_r(0)}E_0(Y)=\frac{T_r(k+1)}{T_r(0)}$MNOF. By combining the condition that MNOF is unchanged between the kth and $(k+1)$st checkpoints, we can further get $E_{k+1}(Y)=E_k(Y)\cdot\frac{T_r(k+1)}{T_r(k)}$. Then, we can derive $X^{(*)}$ as follows:

$$X^{(*)}=\sqrt{\frac{T_r(k+1)\cdot(E_k(Y)\frac{T_r(k+1)}{T_r(k)})}{2C}}=\sqrt{\frac{T_r(k)\cdot\frac{X^*-1}{X^*}\cdot(E_k(Y)\frac{X^*-1}{X^*})}{2C}}$$
$$=\frac{X^*-1}{X^*}\sqrt{\frac{T_r(k)E_k(Y)}{2C}}=X^*-1$$

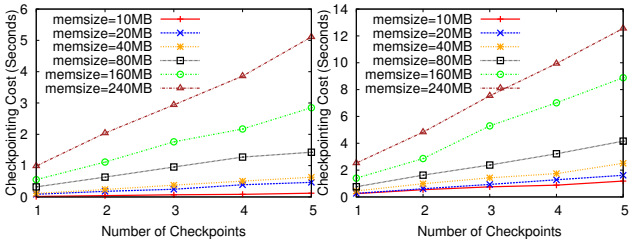
Hence, the next optimal checkpointing position will not be changed if the task MNOF is unchanged. In contrast, if the task MNOF is changed, we can get $X^{(*)}\neq X^*-1$ based on a similar derivation. □

4.2.2 Local Disk vs. Shared Disk Checkpointing

We want to determine what is the most efficient approach, storing checkpoints either in local disks or in shared disks. We denote the checkpointing cost over local disks and over shared disks as C_l and C_s respectively. Based on the two checkpointing approaches, we call their corresponding task migrations (a.k.a., process migration) as migration type A and migration type B. We denote the task restarting cost based on the two checkpointing approaches as R_l and R_s respectively. Then, according to Formula (4), in order to identify the most efficient approach we only need to compare their respective expected total costs, $C_l(X_l-1)+R_lE(Y)+\frac{T_e\cdot E(Y)}{2X_l}$ and $C_s(X_s-1)+R_sE(Y)+\frac{T_e\cdot E(Y)}{2X_s}$, where $E(Y)$ and X respectively refer to the MNOF and the specified number of checkpointing intervals. That is, it is better to store checkpoints on local disks if $C_l(X_l-1)+R_lE(Y)+\frac{T_e\cdot E(Y)}{2X_l}<C_s(X_s-1)+R_sE(Y)+\frac{T_e\cdot E(Y)}{2X_s}$, and to select shared disks otherwise.

How to choose a suitable migration type depends on the system setting. If each VM instance owns a local *ramdisk* with relatively large space, migration type A is likely to be faster than migration type B in that it does not access disks. However, the local disk space and memory size of a VM instance are often both limited, and our benchmark environment belongs to this case. This means that, with BLCR, upon a task failure, we have to move the memory from local ramdisk to shared disks before restarting it on another host, introducing some extra cost. Note that it may be more efficient for a task to precopy its checkpointed memory from local ramdisk to the shared-disk beforehand in case of task failures, yet this may induce seriously heavy load on the network, or even network congestion, since checkpoints are usually much more frequent than task failures. For instance, if a task length, checkpointing cost and expected number of failures are 441 seconds, 1 second, and **2** respectively, then, the number of optimal checkpoints is $\sqrt{\frac{441\times2}{2\times1}}-1=$ **20**.

Obviously, it is necessary to carefully characterize the checkpointing cost and task restarting overhead, based on the above discussion. We evaluate them based on a cluster [26] deployed with BLCR, where each evaluated case is performed 25 times. We find that the task total checkpointing cost increases linearly with its consumed memory size and with the number of checkpoints, as shown in Figure 7. As observed, for the memory size being in [10,240] MB, the checkpointing cost is [0.016,0.99] seconds when using local ramdisk, while it ranges in [0.25,2.52] seconds when adopting NFS.



(a) over local ramdisk (b) over NFS

Figure 7: Checkpointing Cost based on BLCR.

We also investigate the issue about checkpointing conflict, i.e., the situation when checkpointing multiple tasks simultaneously on the same hardware. In our characterization, when two or more tasks are simultaneously checkpointed, the checkpointing cost over local ramdisk will not be impacted significantly, while the cost with NFS will be largely ncreaseed, as presented in Table 2 (memory size=160MB). The table shows the checkpointing cost is quite stable when simultaneously checkpointing tasks and storing memories in the same local ramdisk, while it increases over NFS with increasing number of simultaneous checkpoints. The increased checkpointing cost over NFS is due to the network conges-

tion on NFS servers, or to NFS synchronization mechanism.

Table 2: Cost of Simultaneous Checkpointing Tasks on Local Ramdisk and NFS (seconds).

type \ parallel degree		X=1	X=2	X=3	X=4	X=5
Local ramdisk	min	0.613	0.71	0.51	0.53	0.55
	avg	0.632	0.81	0.74	0.59	0.58
	max	0.667	0.91	0.93	0.69	0.64
NFS	min	1.4	2.66	4.66	5.96	8.36
	avg	1.67	2.665	5.38	6.25	8.95
	max	1.78	2.67	6.05	6.35	9.18

In order to control the simultaneous checkpointing cost over the shared-disk, we design a distributively-managed NFS (DM-NFS) by alleviating the bottleneck problem. We let every physical host in the system serve as an individual NFS server, and make each VM instance mount each of NFS server to a different mount point. As it is required to make a checkpoint for a running task in a VM instance using shared-disk, one of NFS servers will be randomly selected for storing its memory. In Table 3, we present the checkpointing cost when simultaneously checkpointing one or more tasks over the DM-NFS. By comparing it to Table 2, we find that our design is fairly effective in controlling the mutual impact of simultaneous checkpointing. The checkpointing cost is always limited within 2 seconds even with simultaneous checkpointing, which means a much higher scalability.

Table 3: Cost of Simultaneously Checkpointing Tasks on DM-NFS.

type \ parallel degree		X=1	X=2	X=3	X=4	X=5
DM-NFS	min	1.4	1.4	1.54	1.61	1.48
	avg	1.67	1.49	1.63	1.75	1.74
	max	1.78	1.58	1.66	1.89	1.97

Based on BLCR, we find that the operation time cost in making a checkpoint on a running task is determined by its memory size. Each checkpointing operation (over shared-disk) takes 0.33-6.83 seconds when the memory size of a task is 10-240MB, as shown in Table 4. Hence, the checkpointing operation should be performed in a new thread in order to unblock the countdown to the next checkpointing position, as shown in Algorithm 1 (line 7).

Table 4: Time Cost of a Checkpoint.

memory size	operation time	memory size	operation time	memory size	operation time
10.3 MB	0.33 sec	82.4 MB	1.46 sec	162 MB	3.68 sec
22.3 MB	0.42 sec	86.4 MB	1.75 sec	174 MB	4.95 sec
42.3 MB	0.60 sec	90.4 MB	2.09 sec	212 MB	5.47 sec
46.3 MB	0.66 sec	94.4 MB	2.34 sec	240 MB	6.83 sec

In addition, we also characterize task restarting cost based on our system setting, as shown in Table 5 (measurement unit: seconds). It is observed that task restarting cost with migration type A is much higher than that with migration type B, due to the extra cost in accessing shared-disk under migration type A. This is consistent with our analysis above.

Finally, we give an example based on the above characterization to illustrate how to determine a suitable migration type regarding the checkpointing/restarting cost. Suppose that a task execution length T_e is 200 seconds, its memory size is 160MB, and there are 2 failures (expected number) during its execution. With respect to the two migration types, the optimal numbers of checkpointing intervals can be computed as $\sqrt{\frac{200\times 2}{2\times 0.632}} = 17.79$ and $\sqrt{\frac{200\times 2}{2\times 1.67}}$=10.94 respectively. Then, the total costs are $0.632\times(17.79-1) + 3.22\times 2 + \frac{200\times 2}{2\times 17.79} = 28.29$ and $1.67\times(10.94-1)+1.45\times 2+\frac{200\times 2}{2\times 10.94} = 37.78$ respectively. This means it is better to select local-ramdisk as the memory storage device, because it leads to a lower total cost.

Table 5: Task Restarting Cost based on BLCR over VM Ramdisk (Seconds).

memory size (MB)	10	20	40	80	160	240
migration type A	0.71	0.84	1.23	1.87	3.22	5.69
migration type B	0.37	0.49	0.54	0.86	1.45	2.4

5. PERFORMANCE EVALUATION

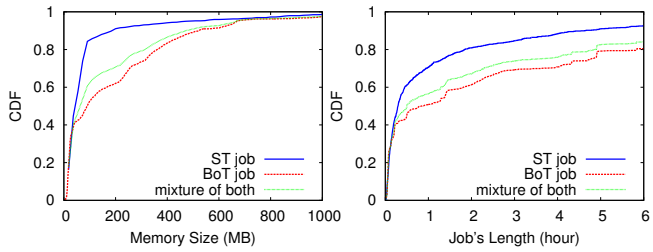
5.1 Experimental Setting

We evaluate both the checkpointing/restart method and the dynamic solution, through a set of comprehensive experiments. We reproduce Google jobs based on a large-scale one-month Google trace [9]. Each job/task's execution is exactly consistent with its arrival time stamp, execution lengths, and task events (e.g., evict, kill or finish events) recorded in the trace. The valid workload processed between checkpoints will be recorded over time. MNOF and MTBF are estimated based on historical task events in the trace, and the details are to be discussed later.

In particular, we perform the experiments on a powerful supercomputer at HongKong (namely Gideon-II [26]). We are assigned 32 physical hosts, each of which has 2 quad-core Xeon CPU E5540 (i.e., 8 cores per host) and memory of 16GB. Our experiment maintains 224 VM-images (centos 5.2) over DM-NFS (7 VMs per host). Each VM is set with 1GB memory size and 1GB ramdisk size. XEN 4.0 [18] serves as the hypervisor on each host and dynamically allocates customized CPU rates to VMs via credit scheduler. Although there are only 32 physical hosts, such an environment can serve up to 600 Google jobs simultaneously, since the processing parallelism is determined by the available memory. The VM selection policy adopts a greedy algorithm that selects the VM instance with the maximum available memory size for load balancing.

In the Google trace, there are two types of job structures, either sequential tasks (ST) or bag-of-tasks (BOT). The structure of each job emulated in our experiment is exactly based on a sample job randomly selected according to the trace. In order to focus on the effectiveness of different algorithms in front of failure events, only jobs half of whose tasks (at least) suffer from a failure event, are selected as sample jobs. Each task memory size is the same as the value recorded in the trace. In Figure 8, we present the CDF of the memory size and execution length of the Google jobs used in our experiment. We can observe that job memory sizes and lengths differ significantly according to job structures; however, most jobs are short jobs with small memory sizes.

Any running task would be killed by "kill -9" command



(a) job memory size (b) job execution length

Figure 8: Distribution of Google Jobs: Memory Size and Execution Length.

from time to time based on the kill/evict/failure events recorded in the trace [9]. Each task is checkpointed using BLCR based on the optimized checkpointing positions. Interrupted/killed tasks will be detected by our polling thread and restarted on another host. If there were network connections opened before the task failure, they will be reopened for retransmitting data upon the task restoration, which is consistent with the design of BLCR.

In our experiment, the key indicator used to evaluate the efficiency of a job execution is called *Workload-Processing Ratio* (WPR), as defined in Formula (9). J_i's workload processed refers to the valid execution length saved by checkpoints, excluding the rollback overhead caused by task failures. The *real wall-clock length* (i.e., T_w) indicates the duration from its submission moment to its final completion moment, including any extra costs caused by task scheduling, task interruption/failure, checkpointing, restarting and data communication.

$$WPR(J_i) = \frac{J_i's\ workload\ processed}{J_i's\ real\ wall\text{-}clock\ length} \tag{9}$$

5.2 Experimental Results

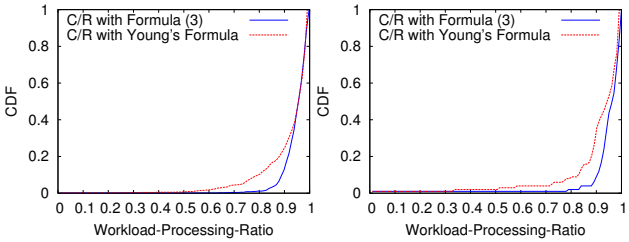
First of all, we analyze the checkpointing effect based on all of 300k Google jobs, using our optimized formula (Formula (3)) versus Young's formula respectively. We observe that if MNOF and MTBF can always be predicted correctly, the checkpointing effects with the two formulas are very close, as shown in Table 6. That is, with exact values, both approaches almost coincide as expected. In practice, however, the prediction of the number of failures and failure intervals may be inaccurate, inevitably leading to the degraded checkpointing effect. Hence, we also analyze the checkpointing effect with possible inaccurate prediction of MNOF or MTBF.

Table 6: Checkpointing Effect with Precise Prediction.

	Formula (3)		Young's formula	
	avg WPR	**lowest WPR**	**avg WPR**	**lowest WPR**
BoT	0.960	0.742	0.954	0.735
ST	0.937	0.742	0.938	0.633
Mix	0.949	0.742	0.939	0.633

The following evaluation shows that the checkpointing effect with our Formula (3) is much better than the one with Young's formula, when we estimate MNOF and MTBF based on priorities. That is, we first categorize all sample jobs into 12 groups based on 12 priorities and compute MNOF and MTBF for each group. We perform the checkpointing with our Formula (3) and Young's formula via the priority-based MNOF and MTBF respectively.

In Figure 9, we present the cumulative distribution function (CDF) of WPR in the situation with ST jobs and BoT jobs respectively. It is observed that the checkpointing/restart method with Formula (3) significantly outperforms the one with Young's formula with high probability. In absolute terms, for sequential-task jobs, the average WPRs of the two solutions with different formulas are 0.945 and 0.916 respectively. The average WPRs of bag-of-task jobs under the two solutions are 0.955 and 0.915 respectively. Moreover, with Formula (3), only 7% of ST jobs' WPRs are lower than 0.88, while with Young's formula, the corresponding ratio is about 20%. With Formula (3), 56.6% of BoT jobs' WPRs are higher than 0.95, while with Young's formula, the ratio is only 46.5%.



(a) sequential-task (ST) job (b) bag-of-task (BOT) job

Figure 9: CDF of WPR with Different Checkpoint-Restart Formulas over Google Trace.

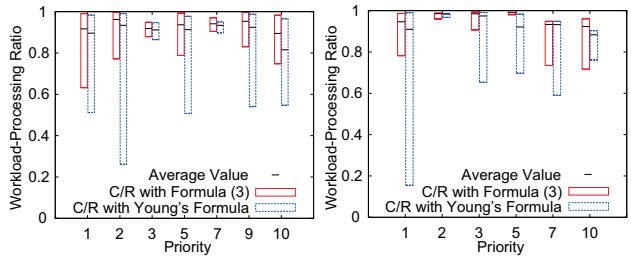
The main reason why our Formula (3) exhibits much better checkpointing effect than Young's formula, is that our formula is based on the MNOF estimated while Young's formula relies on the estimated MTBF. The MNOF and MTBF estimated based on priorities are shown in Table 7 (unit: second). We can observe that MTBF based on each priority is usually very large, due to the typical Pareto distribution of failure intervals as presented in Section 4.1. That is, a majority of failure intervals are short while a minority are extremely long, leading to the large MTBF on average thus large prediction errors for most of tasks. Young's formula is not proper for computing the optimal checkpointing interval due to its assumption with exponential distribution of failure intervals. In contrast, the checkpointing effect with our Formula (3) (shown in Figure 9) is pretty close to the effect with precise prediction of the number of failure events (shown in Table 6). There are two key factors for that. (1) Our formula does not depend on particular distribution like exponential distribution, as discussed in Section 4.1. (2) The mean number of failures (MNOF) estimated based on priority in Google trace would not change a lot with task lengths, rather than MTBF, as shown in Table 7. For instance, MNOF and MTBF of the tasks with priority=2 and lengths$\leq$1000 seconds are 1.06 and 179 respectively, while for all tasks with priority=2 and no limitations on task lengths, MNOF and MTBF are 1.21 and 4199 seconds respectively. That is, if we set MTBF to 4199 seconds, the prediction will definitely lead to large errors for short tasks.

We also present the minimum/average/maximum values of WPR in Figure 10, for the ST jobs and BoT jobs, with

Table 7: MNOF & MTBF w.r.t. Job Priority in Google Trace.

limit (sec)	Pr	Seq-Task MNOF	Seq-Task MTBF	Bag-of-Task MNOF	Bag-of-Task MTBF	Mixture MNOF	Mixture MTBF
task length ≤ 1000	1	0.24	130	1.12	126	0.77	127
	2	1.0	3377	1.06	179	**1.06**	**179**
	7	0.4	186	1.0	80.5	0.15	180
	10	12.0	37.1	4.5	37.4	11.9	37.1
task length ≤ 3600	1	0.52	184	1.18	180	0.72	183
	2	1.0	3377	1.08	396	**1.08**	**396**
	7	0.57	303	1.0	198	0.58	300
	10	13.0	37.9	3.5	93	11.8	37.6
task length ≤+∞	1	3.33	6005	3.46	1074	3.36	5106
	2	0.5	3258	1.27	4274	**1.21**	**4199**
	7	0.57	303	1.0	198	0.58	300.3
	10	9.5	51	3.14	571	9.34	55.5

respect to various priorities. The bottom-edge, middle black line, and upper-edge refer to the minimum, mean, and maximum values respectively. The results at some priorities (such as priority 4, 8, 11 and 12) are missing due to no job failure events or no jobs normally completed according to the Google trace. Through Figure 10, we observe that for almost all priorities, the checkpointing method with Formula (3) significantly outperforms that with Young's formula, by 3-10% on average.



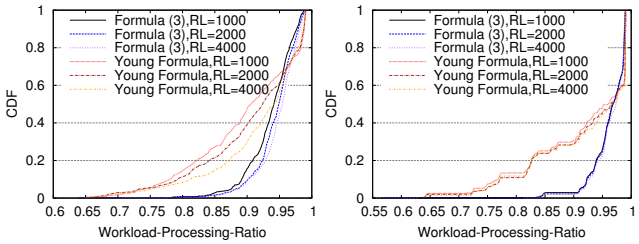
(a) Sequential-Task Jobs (b) Bag-of-Task Jobs

Figure 10: Min/Avg/Max WPR with respect to Different Priorities.

We also investigate the execution performance especially for relatively short jobs with a certain restricted length (RL), with respect to ST jobs and BoT jobs respectively, using a one-day period experiment with totally about 10k jobs. MTBF (as well as MNOF) are estimated using corresponding short tasks based on priorities, in order to estimate MTBF with as small errors as possible for Young's formula. We show the distribution of Workload-Processing Ratio in Figure 11, for the two types of jobs. Under the approach with our Formula (3), 98% of jobs' WPR is greater than 0.9, while Young's formula leads to up to 40% of jobs' WPR being lower than 0.9.

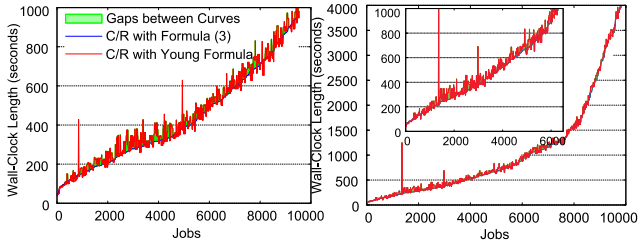
In Figure 12, we present the real wall-clock lengths of the jobs in our experiment, where task lengths are limited within 1000 seconds and 4000 seconds respectively. It is observed that majority of jobs' wall-clock lengths are incremented by 50-100 seconds under Young's formula compared to our Formula (3). Such a difference is actually quite large due to the fact that majority of jobs in Google data centers are quite short (200-1000 seconds) [11].

Through Figure 13, we observe that not all jobs exhibit shorter wall-clock lengths when using our Formula (3) than when using Young's formula, but a large majority of jobs are



(a) Sequential-Task Jobs (b) Bag-of-Task Jobs

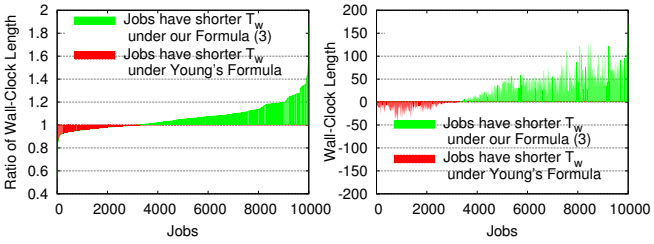
Figure 11: Distribution of WPR in the Test over One-day Google Trace.



(a) RL=1000 seconds (b) RL=4000 seconds

Figure 12: Wall-Clock Length in Experiment with One-day Google Trace.

finished much faster under our solution. In Figure 13 (a), a job's *ratio of wall-clock length* is defined as the ratio of the wall-clock length under our solution with Formula (3) to the one with Young's formula. Comparing our Formula (3) to Young's formula, about 70% of jobs' wall-clock lengths are reduced by about 15% on average, while only 30% of jobs'wall-clock lengths are increased by 5% on average.

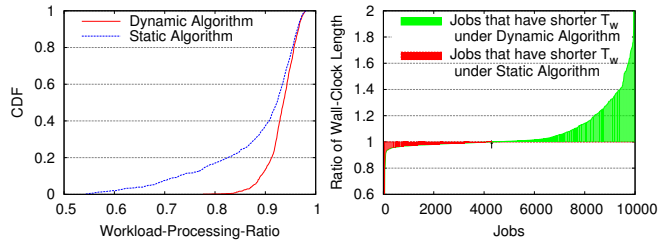


(a) Ratio of Wall-clock Length (b) Wall-clock Length

Figure 13: Portions of Jobs using Different Solutions (RL=1000 seconds).

Finally, we evaluate the effectiveness of our dynamic design with varied MNOF (i.e., line 8-11 in Algorithm 1) as opposed to the static approach with fixed MNOF, in Figure 14 (with one-day trace). In the experiment, each job priority is changed once in the middle of its execution. Upon the change of a task priority, MNOF will change accordingly in our dynamic algorithm, while it will stay fixed in the static algorithm. Via Figure 14 (a), it is observed that the dynamic algorithm significantly outperforms the static one. In absolute terms, the worst WPR under dynamic solution stays about 0.8 while that under static approach is about 0.5. Figure 14 (b) shows that 67% of jobs' wall-clock lengths are similar under the two different solutions, while

over 21% of jobs run faster in the dynamic one than static one by 10%. The key reason why the static algorithm suffers low WPR is that the checkpointing effect would be degraded with skewed MNOF.



(a) Distribution of WPR (b) Ratio of Wall-Clock Length

Figure 14: Comparison between Dynamic Solution and Static Solution.

6. RELATED WORK

The research most related to equidistant checkpointing is Young [17] and Daly [14]. In 1974, Young [17] proposed a mathematical checkpoint/restart model and derived the optimal checkpointing interval with first order approximation. In 2006, Daly [14] extended this work to a higher order approximation, and also took into account the task restarting overhead. Their models both aim at deriving the optimal checkpointing interval for a task, which is similar to our equidistant checkpointing model with Theorem 1. However, they have three significant limitations, and their results are not suited to cloud environments (contrarily to our approach). First, their analysis does not take into account task execution lengths (in fact, they assume very long or infinite jobs), but only analyzes the relation between optimal checkpointing interval and the probability of failure interval. Second, their results depend on the assumption that task failure intervals always follow an Exponential distribution, while ours does not have such a limitation. Finally, their results hold approximately if and only if the checkpointing/restart overhead is far smaller than MTBF. In fact, cloud tasks are usually small as reported by [11], which means that their results would suffer from large errors (to a certain extent) in this context. In contrast, the derivation of Theorem 1 does not suffer from the limitation of checkpointing/restart overhead, being more suitable for cloud frameworks.

More theoretical research on checkpoint/restart-based fault tolerance focuses on stochastic models, which can be found in [27, 28, 29]. Leung and Choo [27] analyzed in-depth job wall-clock lengths by taking into account the probability distribution of failure intervals, restarting overhead, and job execution length. Their model does not depend on a particular distribution of failure events. However, their work suffers from two limitations: (1) It ignored the checkpointing cost. (2) It just derived an expression of task wall-clock lengths in presence of failures, but did not give the optimal number of checkpointing intervals. Walter summarized many stochastic models for checkpointing at program level in his book chapter [28], including equidistant checkpointing, random checkpointing, forked checkpointing, and so on. However, all of the theoretical results depend on the Poisson process of failure events (or Exponential distribution of failure intervals). Nakagawa [29] introduced a bunch of reliability models for optimizing the retrial numbers in presence of failure events, including standard model, checkpoint model, Markov Renewal Process, Bayesian model, and automatic-repeat-request (ARQ) model. They cannot be directly used in clouds because of its common limitation which is ignoring the shortness of cloud job lengths.

Fault-tolerance issue in the context of cloud computing has been extensively studied recently [23]. Nicolae and Cappello [23] proposed a novel checkpoint-restart mechanism (namely BlobCR) especially for high-performance computing applications on Infrastructure-as-a-Service (IaaS) clouds at system level. They summarized four key principles, aiming to improve the robustness of running virtual machines using virtual disk image snapshots. In comparison to their work, this paper focuses on the theoretical optimization of the cloud task execution, and corresponding implementation issues, at the application level. Tchana et al. [30] also focused on the fault-tolerance at system level, on cloud platforms deployed with virtual machines. They tried to improve the tradeoff in the collaboration between provider and consumer, but did not optimize task execution performance based on the failure characterization. Amazon Web Service (AWS) [31] built a queuing model, called Amazon Simple Queue Service (SQS), for tolerating the inevitable failures of message processing. Such a design cannot checkpoint and restart tasks during their execution, significantly restricting the fault-tolerance granularity. In summary, the significant contribution of our work is that we not only optimize the cloud task execution based on our in-depth theoretical analysis with the characterization of real-production traces, but also we propose an adaptive solution by taking into account varied failure probability distributions and the checkpointing tradeoff between using local disks and shared disks. We evaluate our method using a real cluster environment deployed with BLCR and XEN, and the experiments are performed in accordance with Google trace [9].

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel approach to checkpoint/restart task execution in order to improve the execution performance with failure events in the context of cloud computing. Our fault-tolerance model aims at optimizing the number of checkpoints and their positions for running tasks, with minimized checkpointing/restarting cost. Unlike the traditional analysis like Young's work, our theoretical results do not depend upon assuming a particular failure distribution. We also designed a dynamic algorithm based on the characterization of checkpointing cost, to adapt to the task varied remaining workload to process and to possible changing failure probability. We evaluate our optimized fault-tolerance solution with Google trace, which was produced with 10k+ machines and millions of jobs. Some key findings are listed below:

- Our designed DM-NFS can effectively mitigate checkpointing cost when simultaneously checkpointing tasks. The checkpointing cost is always limited within 2 seconds even with simultaneous checkpointings, which means a high scalability.
- For all Google jobs, our approach significantly outperforms Young's solution by 3-10 percent. The average WPRs under our new formula and under Young's formula are about 0.95 and 0.915 respectively.

- Most job wall-clock lengths are reduced by 50-100 seconds under our formula compared to Young's formula.
- On average, our formula leads to about 70% of jobs running faster by 15% than Young's formula, and only 30% of jobs running slower by 5%.
- The dynamic solution with adaptive MNOF based on priority significantly outperforms the static one for a majority of jobs. The worst WPR under dynamic solution stays about 0.8 while that under static approach is about 0.5. 67% of job wall-clock lengths exhibit similar under the two different algorithms, while over 21% of jobs run faster in the dynamic one by 10%.

In the future, we plan to improve our method to better suit high performance computing applications like MPI programs with extremely large scales.

Acknowledgments

This work is supported by the projects ANR RESCUE 5323, *Illinois-INRIA-ANL* Joint Laboratory on Petascale Computing, and also in part by a Hong Kong RGC Grant HKU-716712E.

8. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the clouds: A berkeley view of cloud computing*. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[2] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms For Systems And Processes*. Morgan Kaufmann, 2005.

[3] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. in *Proceedings of 7th ACM/IFIP/USENIX Int'l Conf. on Middleware (Middleware'06)*, pages 342-362, 2006.

[4] C Evangelinos and C.N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. in *Computability and Complexity in Analysis (CAA'08)*, 2008.

[5] Amazon elastic compute cloud: on line at http://aws.amazon.com/ec2/.

[6] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov. Eucalyptus: an open-source cloud computing infrastructure. in *Journal of Physics: Conference Series*, 180(1):1-14, 2009.

[7] J.N. Glosli, K.J. Caspersen, J.A. Gunnels, D.F. Richards, R.E. Rudd, and F.H. Streitz. Extending Stability Beyond CPU Millennium: A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability. in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'07)*, pages 58:1-58:11, 2007.

[8] J. Wilkes. *More Google cluster data*. Google research blog, Nov. 2011, posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[9] C. Reiss, J. Wilkes, and J. L. Hellerstein. *Google cluster-usage traces: format + schema*. Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2012.03.20.

[10] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. *Towards understanding heterogeneous clouds at scale: Google trace analysis*. Intel science and technology center for cloud computing, Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. ISTC-CC-TR-12-101, Apr. 2012.

[11] S. Di, D. Kondo, and W. Cirne. Characterization and comparison of cloud versus grid workloads. in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'12)*, pages 230-238, 2012.

[12] S. Yi, A. Andrzejak and D. Kondo. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. in *IEEE Trans. on Services Computing*, 5(4):512-524, 2012.

[13] W. Cirne, G. Chaudhry, and S. Johnson. *Managing Descheduling Risk in the Google Cloud*. posted at http://cloud.berkeley.edu/data/managing-descheduling-risk-in-the-google-cloud-berkeley.pdf.

[14] J.T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. in *Future Generation Computer Systems*, 22(3):303-312, 2006.

[15] R. Subramaniyan, E. Grobelny, S. Studham, A. George. Optimization of checkpointing-related I/O for high-performance parallel and distributed computing. in *Journal of Supercomputing*, 46(2):150-180, 2008.

[16] M.S. Bouguerra, T. Gautier, D. Trystram, J.M. Vincent. A flexible checkpoint/restart model in distributed systems. in *Proceedings of the 8th international conference on Parallel processing and applied mathematics (PPAM'10)*, pages 206-215, 2010.

[17] J.W. Young. A first order approximation to the optimum checkpoint interval. in *Communications ACM*, 17(9):530-531, 1974.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. in *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03)*. pages 164-177, New York, NY, USA: ACM, 2003.

[19] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. in *Commun. ACM*, 51(1):107-113, 2008.

[20] P.H. Hargrove and J.C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. in *Journal of Physics: Conference Series*, 46(1):494, 2006.

[21] L.A. Barroso, J. Dean and U. Holzle. Web search for a planet: The Google cluster architecture. in *Journal of Micro*, 23(2):22-28, 2003.

[22] L. Huang, J. Jia, B. Yu, B.G. Chun, P. Maniatis, and M. Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. in *Proceedings of 24th International Conference on Neural Information Processing Systems (NIPS'10)*, pages 1-9, 2010.

[23] B. Nicolae and F. Cappello. BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots. in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 34:1-34:12, 2011.

[24] S. Boyd and L. Vandenbergheąč *Convex Optimization*. Cambridge University Press, 2009.

[25] S. Di and C.L.Wang. Error-Tolerant Resource Allocation and Payment Minimization for Cloud System. in *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 24(6):1097-1106, 2013.

[26] Gideon-II Cluster: http://i.cs.hku.hk/∼clwang/Gideon-II.

[27] C.H.C. Leung and Q.H. Choo. On the Execution of Large Batch Programs in Unreliable Computing Systems. in *IEEE Trans. on Software Engineering*, 10(4):444-450, 1984.

[28] K. Wolter, Stochastic models for checkpointing. in *Stochastic Models for Fault Tolerance*, pages 177-236, Springer Berlin Heidelberg, 2010.

[29] T. Nakagawa. Optimum retrial number of reliability models. in *Advanced Reliability Models and Maintenance Policies*, pages 101-122, ser. Springer Series in Reliability Engineering. Springer London, 2008.

[30] A. Tchana, L Broto, and D. Hagimont. Fault Tolerant Approaches in Cloud Computing Infrastructures. in *Proceedings of the 8th International Conference on Autonomic and Autonomous Systems (ICAS'12)*, pages 42-48, 2012.

[31] J. Barr, A. Narin, and J. Varia. *Building Fault-Tolerant Applications on AWS*. Tech. Rep., Oct 2011.