GloudSim: Google Trace based Cloud Simulator with Virtual Machines

Sheng Di[1][2], Franck Cappello[2]
[1]INRIA, Saclay, France, [2]Argonne National Laboratory, USA
sheng.di@inria.fr, cappello@mcs.anl.gov

Abstract—**In 2011, Google released a one-month production trace that was produced with hundreds of thousands of jobs running across over 12000 heterogeneous hosts. In order to perform in-depth research based on the trace, it is necessary to construct a close-to-practice simulation system. In this paper, we devise a distributed cloud simulator (or toolkit) based on virtual machines, with three important features. (1) The dynamic changing resource amounts (such as CPU rate and memory size) consumed by the reproduced jobs can be emulated as closely as possible to the real values in the trace. (2) Various types of events (e.g., kill/evict event) can be emulated precisely based on the trace. (3) Our simulation toolkit is able to emulate more complex and useful cases beyond the original trace to adapt to various research demands. We evaluate the system on a real cluster environment with 16×8=128 cores and 112 virtual machines (VMs) constructed by XEN hypervisor. Experiments show that our simulation system could effectively reproduce the real checkpointing/restart events based on Google trace, by leveraging Berkeley Lab Checkpoint/Restart tool. It can simultaneously process up to 1200 emulated Google jobs over the 112 VMs. Such a simulation toolkit has been released as a GNU GPL v3 software for free downloading, and it has been successfully applied to the fundamental research on the optimization of checkpoint intervals for Google tasks.**

I. INTRODUCTION

With increasing demands on computing resources over the Internet and fast development of virtual resource isolation technology [1], cloud computing has become a compelling paradigm that can provision elastic services and on-demand resources for various users. An ease-of-use and close-to-practice cloud simulator is very helpful for the in-depth research on cloud computing. In comparison to traditional simulation systems like Grid simulators [2], [3] and P2P/network simulators [4], [5], there are many new requirements (or challenges) for cloud simulation systems.

- The execution environment in cloud computing has fairly high demands on resource isolation. For instance, any cloud user is supposed to be able to customize one or more particular execution environments in the form of virtual machine (VM) instances [6], each of which is allocated with multiple types of on-demand resources, e.g., CPU rate and memory sizes.
- There are much more restrictions for a cloud task execution than Grid task execution, introducing more difficulties to the cloud simulation work. In general, different types of resource amounts consumed by a cloud task may be related to many factors like user bids, payment budgets and job priorities. For example, the resource allocation in Amason EC2 [7] is dependent on user's bids and payment model; quite a few optimization strategies [8], [9], [10] on cloud resource allocation need users to provide preferred payment budget in advance; a task's scheduling priority in a Google data center [11], [12] is mainly determined by its execution priority[1] assigned by its user or administrator.
- Unlike high performance computing (HPC) and Grid applications, cloud application's execution may not exhibit regular rules in cloud computing environment. In general, HPC/Grid applications are usually computation-intensive programs, requiring much heavier workload on computation than other types of resources. However, there are more various types of applications in cloud environment. Mapreduce [13], for example, is a kind of I/O-intensive distributed program. Google trace [11] shows that most of Google applications' demand on memory size is higher than that on CPU rate. Moreover, we can also observe through Google trace [12] that task's interruptions like kill or evict events are hard to characterize/predict in that they do not exhibit regular rules with task's features like execution length or scheduling class[2].

In November 2011, Google released a one-month production trace that was produced with hundreds of thousands of jobs running across over 12,000 heterogeneous hosts. This trace involves about 4000 types of applications like Mapreduce programs and other data mining programs. So far, there have been quite a few existing research [14], [15], [16] on the in-depth characterization and analysis in the context of Google data centers, including the characteristics of hostload, taskload, task length, resource usage/constraints, various task events like failure and eviction. However, there are no off-the-shelf simulation toolkits to help reproducing such a Google cloud environment based on the Google trace.

In this paper, our objective is to explore a fairly effective method that can build a close-to-practice cloud simulation system based on Google trace, providing huge convenience for the further research on cloud computing. In particular, we propose quite a few novel insights in effectively sim-

[1]Google job priority has twelve levels, denoted 1-12. Bigger priority value indicates higher execution priority.

[2]In Google trace, bigger value of scheduling class (0-3) implies a more latency-sensitive task (e.g., serving revenue-generating user requests) while smaller value means a non-production task (e.g., development, non-business-critical analysis, etc.).

ulating Google jobs/tasks based on our experiments with 128 cores and 112 virtual machine (VM) instances. For instance, we find that Google task execution can be emulated precisely through a while-loop with alternating sleep/wakeup operations, by excluding the estimated time cost of setting sleep/wakeup clauses. The checkpointing overhead does not change clearly with the percentage of CPU usage consumed but would be closely related to the task's memory size and sleeping interval. Google task's memory consumed does not change noticeably during its execution, thus it can be emulated by loading a file at the beginning of its execution.

This simulation toolkit has been released as a GNU GPL v3 software [17] for free downloading. All of modules in the toolkit are loosely coupled and are implemented using Java programming language, such that the whole software has a high portability. It has been successfully applied to the fault-tolerance research on optimization of Google task checkpoint intervals, which was published in Supercomputing'13 conference [18]. We believe that our work is very helpful for researchers to customize a close-to-practice simulation system with their particular demands.

We build/evaluate the cloud simulation system using a real cluster with hundreds of VM instances and Berkeley Lab Checkpoint/Restart tool (BLCR) [19]. Its effectiveness is evaluated through the well-known checkpointing theory Young's formula [20], as well as well-known scheduling policies like First-Come-First-Service (FCFS). Experiments show that our simulator could simultaneously process up to 1200 jobs in parallel.

The remainder of the paper is organized as follows. We present the design overview of our simulation system in Section II. In this section, we first briefly introduce the concept of Google trace related to our simulation system and then describe the key modules. In Section III, we discuss some key technical skills in our implementation, including how to simulate Google jobs/tasks according to their traced lengths as accurately as possible, how to detect task failures in the system, and how to precisely compute experimental job/task's productive length and wall-clock length. In Section IV, we present the evaluation results about the peak parallel degree of processing tasks and task execution efficiency in our simulation system. Finally, we discuss the related works in Section V, and conclude with a vision of the future work in Section VI.

II. Design Overview

In this section, we briefly introduce Google trace [11], [12], and present a design overview in building a simulation environment based on the Google trace.

A. Overview of Google trace

The Google trace was produced on a Google data center for one month, with about 670,000 jobs running across over 12,000 hosts. All the hosts are connected via a high-speed network. Users submit their requests in the form of jobs, each of which contains one or more tasks to execute. There are totally 25 million tasks recorded in the whole trace. Any job has a particular priority assigned by its user or administrator, and there are 12 priorities in total. A central-controlled scheduler is deployed on a server for scheduling the jobs in order to coordinate their priorities. The tasks in one job may be connected in series (called sequential-tasks) or in parallel (called bag-of-tasks) or the mixture of both. For instance, a Mapreduce program will be handled as a job with multiple reducer tasks and mapper tasks concurrently. Each task is always generated with a set of user-customized requirements (such as the minimum memory size).

Based on Google trace guide [12], each task has four possible states, *unsubmitted*, *pending*, *running* and *dead*. Any newly submitted task will be first put in a pending queue, waiting for the resource allocation. As long as a qualified host meets the task's constraints, the task will be executed on it. Users are allowed to tune their tasks' constraints at runtime. Any task could be evicted, killed, failed, or lost during its execution, possibly due to priority, resource contention or other factors. Upon a task is finished, it will enter into a dead state, and it can be resubmitted by users later on.

When releasing the trace, Google normalized almost all floating-point values by its theoretical maximum value, for purpose of confidentiality. These values were always transformed in a linear manner, for keeping the validity of the trace well. Specifically, the maximum host memory capacity from among totally 12,000 hosts is treated as 1. Any other memory related values (e.g., task's memory size consumed or other hosts' memory capacities) would be transformed into [0,1] through an affine transformation.

B. Overview of the Simulation System

As described above, the whole system architecture can be split into two parts based on the Google trace guide [12], a job scheduling server and a set of task execution hosts (or slave nodes). Each task will be executed in a VM instance, which is running on a physical execution host. We first devise the architecture for server end and execution end respectively, and then make them communicate mutually.

The design architecture of the server end is shown in Figure 1, including hardware layer and software layer.

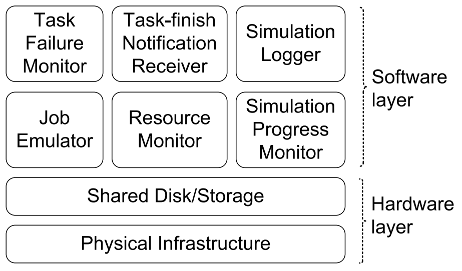


Figure 1. Design Architecture on the Server

The hardware layer contains a shared disk/storage device, a.k.a., General Parallel File System (GPFS), and other physical infrastructure like memory, CPU, and network. As for the GPFS, we designed a novel framework (called distributed-management NFS or DM-NFS for short) to effectively mitigate I/O congestion resulted from simultaneous checkpoint taking, significantly improving system scalability. We let every 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 GPFS, one of NFS servers will be randomly selected for storing its memory. The effectiveness of such a distributed design is confirmed by our experiment, to be shown in Section IV.

In the software layer, there are six key modules to perform the whole simulation work together. Their functions are described as follows.

- *Task Failure Monitor*: Task failure monitor is used to detect task failure events and make task-rescheduling decisions upon detecting the task failures.
- *Task-Finish Notification Receiver*: Task-finish notification receiver is used to listen through a separate thread to the notification messages sent by completed tasks from remote VM instances.
- *Job Emulator*: Job emulator is a critical module that aims to simulate various jobs based on Google trace. Its structure is illustrated in Figure 2. According to the figure, Google trace will be processed through a set of steps. (1) Google trace analyzer: we characterize each job recorded in the Google trace, with respect to the resource utilization of its tasks, task failure events, and task lengths. (2) Sample job generator: according to the trace, there are totally 670,000 jobs submitted, yet only 370,000 jobs (called *valid jobs* in the following text) are completed normally in the end. Based on the characterization work, we then generate a large set of sample jobs based on all of valid jobs. (3) Experimental job generator: various jobs used in experiments can be emulated by randomly selecting sample jobs iteratively. There are two ways in simulating job arrivals, generating jobs with an upper bound of parallelism or exactly based on job arrival timestamps recorded in the Google trace. The former is used to evaluate the maximum parallelism of the system throughput and the latter aims to make the simulation approach the original Google trace. (4) Job/task scheduler: any generated job needs to be scheduled through a scheduler before its execution.
- *Resource Monitor*: Resource monitor is used to monitor the states of resources, e.g., the aliveness of physical hosts and VM instances. It contains three submodules, host alive state monitor, VM alive state monitor and VM memory state monitor. They are performed concurrently in separate threads, by periodically communicating with sensors deployed on the hosts/VM instances, to be discussed later.

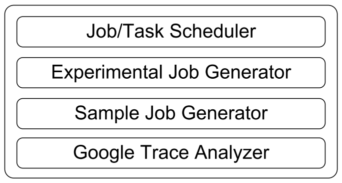


Figure 2. Architecture of Job Emulator

- *Simulation Progress Monitor*: Simulation progress monitor is used to check if all of the jobs submitted are completed. The whole simulation will stop if and only if all of jobs already enter into dead states.
- *Simulation Logger*: Simulation logger serves as an observer to record the simulation state over time on the server end, e.g., the number of jobs/tasks in the queue, the number of finished tasks, and so on. It can also help generating analytical results based on the observed data.

The design architecture of the execution end (i.e., salve node) is shown in Figure 3. On top of the physical hardware is virtual machine monitor (VMM) and the storage device used to store tasks' checkpointed memory. Above the two modules, it is necessary to build a virtual machine (VM) instance layer, in that virtual resources customized on demand is a critical feature of cloud computing environment.

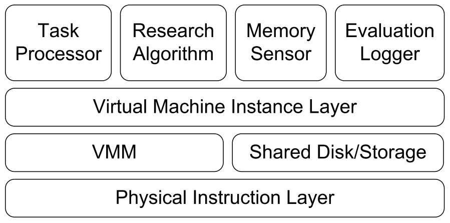


Figure 3. Design Architecture on Execution Hosts

On the execution end, there are four key modules to perform task execution and task failure simulation, and each of them is deployed in each VM instance. Each module is implemented via a separate thread. We briefly describe their functions below, and more details will be discussed later.

- *Task Processor*: Task processor is to receive task (re)scheduling notifications from the cloud server and perform task execution, checkpointing, and simulate failure events. Its framework can be split into four parts: *Task execution launcher*, *Task failure simulator*, *Task checkpointing monitor*, and *Task restarting listener*. (1) *Task execution launcher* is used to start a specific program with a specified productive length (i.e., the execution time without taking into account time cost due to task failures). (2) *Task failure simulator*'s key function is to kill running tasks (i.e., running processes) based on the failure events recorded in the Google trace [12]. (3) *Task checkpointing monitor* is used to checkpoint the running tasks by checkpointing tools like BLCR from time to time. The checkpointing intervals are determined by the research algorithm module shown in Figure 3. (4) *Task restarting listener* is used to receive task restarting notification messages from the server, and restart the corresponding failed tasks from their latest checkpoints.

- *Research Algorithm*: Research algorithm module contains all the algorithms to be evaluated for the purpose of research, such as some novel algorithm about optimized checkpointing intervals. In the released toolkit, we help implement Young's formula [20] as a case.
- *Memory Sensor*: Memory sensor is used to collect the instant states of memory utilization on the VM instance, such as the free memory size.
- *Evaluation Logger*: Evaluation logger is a kind of observer to record the key indicators to be studied, such as the task's real wall-clock length and their checkpointing/restart cost.

III. Implementation of the Simulation System

In general, the whole simulation includes host simulation and job simulation. With VM resource isolation technology [1], any VM's capacity like CPU rate and memory size can be easily customized. In our experiment, we simulate hosts by running multiple VM instances with different capacities in a cluster environment. In addition to host simulation, job simulation is rather complex, in that different jobs/tasks may have quite various resource utilizations and failure events during their executions. In this section, we mainly study how to precisely reproduce the Google jobs and tasks based on the trace data, as well as effective approaches in detecting system states like aliveness of VM instances in time.

A. Effective Simulation of Jobs/Tasks based on Google trace

Through a thorough characterization of Google jobs/tasks (about job length, resource utilization, priority, etc.), we devise a Google trace analyzer. According to the Google trace, we build a sample job generator (as shown in Figure 2), based on which we can further selectively generate a set of sample jobs to suit specific research demands. For example, if a researcher just wants to focus on the jobs with frequent task failure events, the sample job generator could select the sample jobs each of which has at least one failure event during its execution (ignoring jobs with no failures).

Each Google job has one or more tasks connected in series or in parallel to execute, and the tasks should be reproduced exactly based on Google trace. A Google task is a defacto workload-execution carrier, which corresponds to one or more processes in Linux/Unix systems. Any simulated task attached to a particular experimental job will be consistent with a sample task belonging to the corresponding sample job. As follows, we first present the characterization of resource utilization and failure/kill events per task according to Google trace, and then describe how to precisely emulate the task execution features in a practical cluster environment.

1) Characterization of Google Resource Utilization and Failure Events: Before investigating how to precisely reproduce Google task execution features, it is necessary to characterize the Google task resource utilization and task events based on the trace data.

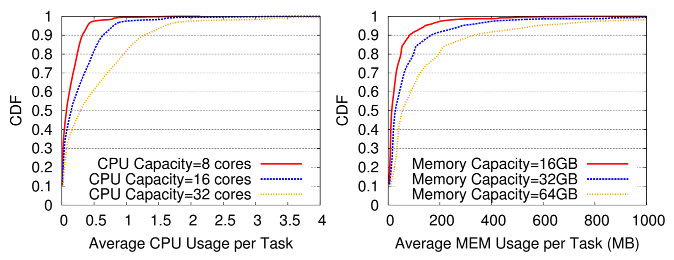
In order to ease users' simulation work, our toolkit includes a set of statistics about failures like failure intervals. The mean time between failures (MTBF) is the most important metric especially in that many checkpointing theories are based on its value, e.g., Young's formula [20] and Daly's work [21]. In Table I, we present the MTBF of Google tasks that have at least one failure event, based on different task priorities (from 1 to 12). For some high priorities like priority 8, 9, 11, 12, MTBF is unavailable because there are no task failure events or they have no completion record in the end of the trace. Through the table, we clearly observe that MTBF changes with different priorities, which can be leveraged to refine the estimate of MTBF. In our implementation, each task's statistical information is integrated in a Java object constructed by Job Emulator module in the server end (see Figure 1)).

Table I
MTBF of Google Tasks (Second)

Priority	MTBF	Priority	MTBF	Priority	MTBF
1	5106	2	4199	3	9672
4	23.4	5	1687	6	-
7	300.3	8	-	9	-
10	55.5	11	-	12	-

Our characterization shows that most of tasks consume very few amount of resources. We present the distribution of resource utilization per Google task in Figure 4. Since the trace provider deliberately hid the capacity information (such as maximum number of cores per host and maximum memory size per host) for confidentiality reasons, we can only get from the trace an affine-transformed usage values that are compared to the maximum capacity. Hence, one has to speculate the maximum resource capacity per host for simulation before investigating the real resource utilization per task. In Figure 4, we present the cumulative distribution function (CDF) of the CPU utilization (core/second) and memory size consumed (MB), based on different conjectures of possible capacities. From Figure 4 (a), we observe that if a host in the data center has at most 8 cores, about 97% of tasks consume less than 0.5 core per second on average. Even though the maximum number of cores per host is 32, there are still 80% of tasks each consuming less than 1 core per second. From Figure 4 (b), it is also observed that majority of tasks consume very small memory size per task at runtime.

2) Precise Simulation of Task CPU Utilization: We devise an approach that can reproduce Google tasks running in the system by using Java processes, such that the percentage of CPU usage and execution length are close to the specified



(a) CPU Usage (Cores/sec) (b) Memory Usage (MB)

Figure 4. Distribution of CPU and Memory Usage per Task

values in the trace. Each Java program used to simulate a task is coded using a while-loop that repeatedly performs *add* operations. There are three parameters for tuning the Java program's execution length and its CPU usage: the number of cycles, sleeping frequency, and sleeping interval. The values of the three parameters are dependent on particular system setting and hardware configuration. In our experiments, any one Java program always runs in a VM instance, whose CPU rate capacity is restricted as one core of the CPU processor (Xeon E5540) by credit scheduler [22]. Then, we can characterize the execution times of the Java program with different parameters, as shown in Table II. The task execution length of each case in the table is 10 seconds.

Table II
PARAMETERS USED IN THE SIMULATION OF CPU PERCENTAGE

CPU Percentage	number of cycles	sleeping frequency (10^3 cycles per sleep)	sleeping interval (second)
10%	89×10^6	74.4	0.002
20%	84.6×10^6	78.3	0.002
30%	108×10^6	108	0.002
40%	161×10^6	161	0.002
50%	178×10^6	178	0.002
60%	200×10^6	200	0.002
70%	220×10^6	220	0.002
80%	243×10^6	243	0.002
90%	257×10^6	257	0.002

In fact, it is inadvisable to always force task execution to stick to CPU usage values recorded in the trace. On one hand, forcing each task's CPU usage to be consistent with the trace will lead to the whole simulation system suffering a quite low scalability, because there is a CPU capacity limitation for each physical host. On the other, most of the cloud research is focused on fault-tolerance issue or I/O processing overhead instead of CPU utilization. Thereby, we will focus on how to precisely reproduce the following metrics, a task's memory utilization, task length, and checkpoint overheads.

3) Precise Simulation of Task Memory Utilization: A precise simulation of a task's memory consumption according to the trace is important because many task events and running features are relative to memory usage (shown later). Hence, in our simulation system, each task is to be started with a specific amount of memory size, according to its corresponding sample task in the trace. Based on the Google trace, we find that for a large majority of Google tasks, the memory sizes consumed change little during their execution. Specifically, the mean differences of the memory sizes between consecutive intervals in five minutes is only about 5% of the total memory sizes consumed. Consequently, we can emulate the memory size consumed by a running task by letting it load a data file from the disk at the beginning of its execution. In order to make a task's loaded memory size be consistent with a specified value, it is necessary to characterize the real memory size consumed by the task when loading data files in different sizes.

In Table III we present the relation between the memory size consumed by Java programs (i.e., the total memory size allocated to JVM) and the data file size loaded. From this table, we observe that the real memory size consumed by a task in the simulation can be controlled by loading a particular file in a much smaller size, which means a relatively high scalability in storing the memory-simulation files. For instance, to simulate a task loaded with memory size of about 80 MB, we just need to execute a Java program with a data file in size of 10 MB.

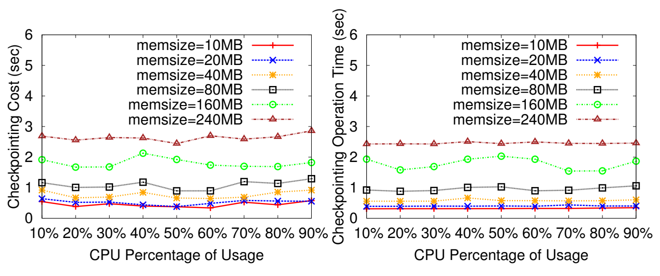
Table III
INVESTIGATION OF FILE SIZE VS. MEMORY SIZE CONSUMED (IN MB)

file-size	mem-size	file-size	mem-size	file-size	mem-size
2	10.27	4	22.34	6	42.34
10	82.37	16	94.4	20	166.4
24	174.46	28	182.48	32	190.5
40	208.4	50	228.3	60	248.33

4) Precise Simulation of Task Length and Checkpoint Overheads: In general, it is nontrivial to precisely emulate a particular task's execution length and cost based on the trace data. The execution time (or productive time, excluding cost on failure events) of the task is supposed to be equal to the length of the aliveness of its corresponding process. That is, the real length of the process should be controlled as closely as possible to the task length recorded in the trace. A straightforward idea is to simulate task length by launching a thread and making it sleep for a specific time until the task ends according to the trace. Such an approach can simulate the task execution length precisely but ignore the computational resource consumed (e.g., the CPU rate and memory size allocated). This will cause the checkpoint overhead to be largely skewed from the practical cases because it is closely related to resource consumption based on our characterization.

We also characterize the checkpoint overhead (including checkpoint cost and operation time) based on BLCR, with various percentage of CPU utilization and different memory sizes, as shown in Figure 5. Note that checkpointing cost is different from operation time. The former is the increment of the task's wall-clock time due to one checkpoint while the latter means the time cost in performing the checkpoint operation. In Figure 5 we see that the checkpoint overhead does not change noticeably with different CPU utilization but increases linearly with the memory size.

Based on the figure, if one just needs to focus on the

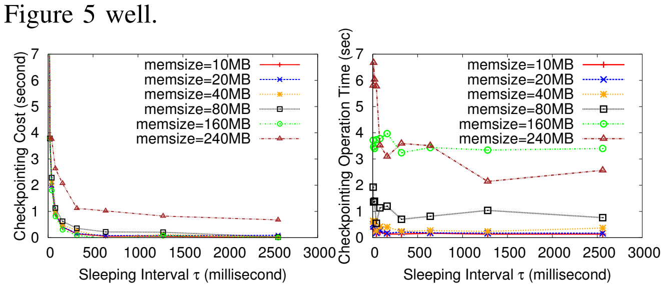


(a) Checkpointing (CP) Cost (b) Operation Time of CP

Figure 5. Checkpoint Overhead with Various CPU Usage and Memory

checkpoint overhead, it is not necessary to simulate CPU utilization precisely for each task. Instead, we just need to simulate task's memory size and task length as precisely as possible, based on the Google trace. Our characterization shows that running a while-loop program without *add* operations inside can still fit the real checkpoint overhead well, as long as the periodic sleeping interval (denoted by τ) is tuned properly. Specifically, the evaluated task's execution length and checkpoint overhead can be easily customized by letting the process sleep every τ milliseconds.

In Figure 6, we present the checkpoint overhead based on BLCR, by running a Java program with different sleeping intervals. In Figure 6, we see that the checkpoint overhead of a while-loop program with periodic sleeps but with no operations inside, depends just on the sleeping time interval τ and memory size. Specifically, the checkpoint overhead with fixed sleeping interval will increase linearly with memory size. If the memory size is fixed, the checkpoint overhead decreases exponentially with τ when $\tau \in [10,200]$ milliseconds and remains stable as τ is greater than 200 milliseconds. Based on this characterization, we find that the checkpoint overhead with τ=100 milliseconds (highly recommended setting) will fit the true values shown in Figure 5 well.



(a) Checkpointing (CP) Cost (b) Operation Time of CP

Figure 6. Checkpoint Overhead with Various τ

One can easily control a task's execution length using the periodic-sleep-based while-loop with null operations inside. Our characterization shows that the sleep operation costs little during the task execution. In particular, one sleep operation will increase a Java program's execution length by about 3%. For example, if a Java program sleeps every 100 milliseconds and there are 1,000 sleeps, then its real execution time can be estimated as $100\times1000\times1.03$ milliseconds = 103 seconds. In other words, if one wants to control the execution length to be close to 100 seconds, the number of sleeps should be equal to $1000\times\frac{1}{1.03}\approx971$.

5) Effective Simulation of Task Failure and Task Rescheduling: In Figure 7, we present the procedure in processing a task from its generation until its completion, regarding task failure events, task checkpointing, and restarting operations. At the beginning of the simulation, the master node will generate a set of threads corresponding to the software-layer modules in the server end (Figure 1). Then, a set of jobs will be generated based on job arrival time stamps recorded in the Google trace or other job arrival rules (e.g., a particular random process). Any job is a sample Google job in the trace, and each job contains one or more tasks in series or in parallel. Whenever a task is to be executed on a VM instance, the task execution launcher on the VM instance will receive a notification and start the task locally. The checkpointing monitor on this VM instance will start to checkpoint it from time to time based on some rule about checkpointing intervals.

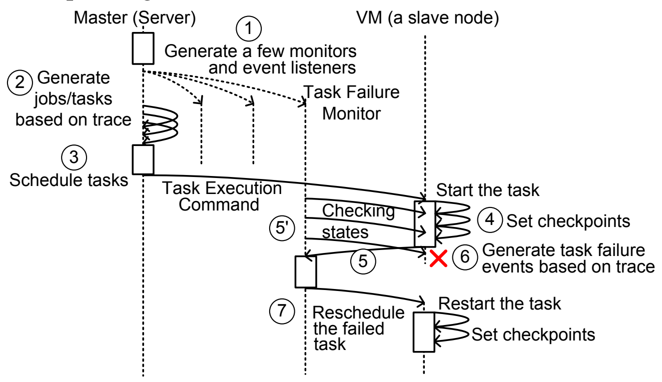


Figure 7. Procedure of Task Failure Detection

We try to make the simulation consistent with the practical situation as closely as possible in many details.

- In our implementation, each task is represented by a Java object that contains all the information about the corresponding sample Google task, such as the dates of task failure events, consumed memory size, and task execution length. All the generated tasks will be periodically scheduled based on some queuing policy (e.g., FCFS) and resource availability in the system. A task's failure events will be generated by the local task failure simulator based on the task failure events stored in the task object.
- Task failure events can be detected in two ways. On the one hand, any task may be killed or evicted because of its intrinsic problems or mutual competition on resources while VM nodes still stay alive. In this case, the task simulator in the VM instance could send a message to the task failure monitor in the server end as the notification of the failure events (Step (5) shown in the figure). On the other hand, any VM instance

on an execution host may be down (or disconnected) because of overconsumption of resources (e.g., memory exhaustion) and all the tasks that were running on the VM instance have to be restarted on another available VM instance. In this case, the task failure events can be detected only by a polling mechanism performed by the master node, shown as the Step (5') in Figure 7.

- As a task starts to be executed on a VM instance, the checkpointing monitor will checkpoint it by storing its instant memory into a peripheral device from time to time. The peripheral device could be a local disk or a shared disk such as a network file system. In our simulation system, we implement the checkpointing monitor by BLCR, which can checkpoint any process at any time.

B. Effective Detection of VM's Run-Time States

It is critical to precisely monitor the real-time states of each VM instance. The VM states to monitor in our system mainly include the aliveness of VM instances and their free memory sizes, in that they are key factors to impact the task running states and task scheduling decisions. Based on our experiments, we find that if a VM instance uses up its memory, the VM instance cannot be accessed any more, leading to a down state unexpectedly. For example, the delay in memory-state-message communication may cause inevitable detection errors. This may induce overallocation of memory size on a VM (i.e., memory exhaustion), leading to the inaccessible state of the VM instances.

It is relatively simple to monitor the aliveness of VM instances by repeatedly pinging VM instances concurrently, whereas it is nontrivial to maintain their precise free memory values since memory is dynamically consumed by various operations such as transferring data via the network or reading/writing data from/to disks. Moreover, the delay of communication between the server's monitor and execution node's sensor may also introduce errors in the estimation.

In our simulation system, we explore an approach that can keep the memory states about VM instances on the server as precisely as possible. In our method, we try to estimate any VM instance's free memory size conservatively.

Specifically, the available memory sizes of all VM instances are maintained as a list of values on the server node. The values in the list can be changed in two ways: via the memory state monitor or via the computation based on task failure events and task (re)scheduling. For the former way, a particular thread (the memory state monitor) on the server end that periodically detects each VM instance's instant free memory, by communicating with the memory sensors deployed in VM instances. For the latter, the initial memory size of each VM instance is set to its capacity, and its available memory size will be computed upon a task scheduling or notification of task failure events. At anytime, the free memory size of a VM instance (denoted by fm_{vm}) will be set to the smaller value retrieved/computed based on the above two ways, as shown in Formula (1).

$$fm_{vm} = \min(fm_s, fm_c),\ where\ fm_s\ and\ fm_c\ refer\ to\ the\ free\ memory\ values\ based\ on\ sensor\ and\ computation\ respectively. \tag{1}$$

Such a formula can significantly reduce the probability of memory exhaustion on VM instances, effectively avoiding the unexpected VM failures. For example, since a task is scheduled onto just one VM instance, the corresponding VM's free memory size will be immediately reduced by taking away the estimated memory size to be consumed by the task from the VM instance's original free memory size. This approach can effectively avoid the conflicts of memory allocation due to simultaneously scheduled tasks.

C. Assessment of Task Execution Efficiency

How to assess task execution efficiency is a key issue in the simulation system. This is related mainly to the simulation logger on the server end (shown in Figure 1) and the evaluation logger on the execution end (shown in Figure 3). The simulation logger is a key module to keep track of the running states of observed data on the server end, such as the number of jobs/tasks submitted, scheduled, failed or finished. The evaluation logger is used to record the detailed information about task execution occurring on VM instances, such as resource utilization, checkpointing storage device (whether using shared disk or local disk), and task events such as kill and eviction.

In general, a task's valid workload processing time (i.e., productive time) and real wall-clock time are two key indicators of most concern. In our simulation system, the productive time stamps of any task are recorded during its execution, based on interval lengths between checkpoints. The wall-clock time of any task is equal to the whole length from the task's submission to its completion, including task scheduling cost, the roll-back time due to failure events, checkpointing/restart cost, and data transmission cost.

A particular challenge involves estimating the productive times of the tasks that are not normally completed before the system simulation deadline. If a task is completed normally, its productive time must be equal to the task execution length recorded in the Google trace because all the workload has been finished in this situation. On the other hand, if a task is killed but never restarted again before the system simulation deadline, its productive time must be smaller than the original task length. In this case, the productive time has to be estimated based on its checkpointed file. However, every time a task is restarted, it is impossible for the task by itself to detect whether it should continue processing its remaining workload or terminate with an output of its workload already processed for analysis.

This issue can be solved by maintaining a *system-state mark* in an out-of-box configuration file. Our simulation system has two states: *simulation state* and *evaluation state*. The simulation state means that the system is running a simulation; the evaluation state indicates that the simulation is already finished and the system is collecting the log information for analysis. At the beginning of each task execution, it will start a separate thread listening to the system state. Whenever the task is restarted from a checkpoint by BLCR, its execution will immediately trigger an exception to check the current system state and decide whether it should continue running the remaining workload. It should continue with the simulation state, and should not otherwise. After the whole simulation test, each unfinished task will be restarted based on its last checkpoint for collecting its eventual workload processed, and then it will be immediately terminated. Such a method can guarantee a precise estimate of each task workload processed in the system.

In addition to these two indicators, we provide another one for the evaluation of task's execution efficiency, namely, the *workload processing ratio* (*WPR*). WPR is defined as the ratio of the productive time to wall-clock time. It can be used to evaluate the working efficiency of the checkpointing mechanism designed in the simulation.

IV. Performance Evaluation

We briefly describe the setup of our experiment before turning to a discussion of the results.

A. Experimental Setting

We evaluate our simulation system based on some use-cases with a powerful cluster called Gideon-II [23], which is located in Hong Kong. In our evaluation, there are 16 physical hosts, each of which has 8 cores and 16 GB in memory capacity. Each host is deployed with XEN 4.0 hypervisor [24] for managing the virtual resource isolation. We reserve 2 GB of memory for the XEN hypervisor usage on each host. There are 7 VM instances running per host, each instance customized with 1 CPU core and 2 GB of memory. Note that each VM instance just has 100 MB of local disk for user usage in order to control the image sizes; thus we treat a portion of memory (1 GB) as the local checkpointing storage device (i.e., diskless checkpointing). In our toolkit, we also provision a set of Java APIs to integrate the simulation with various BLCR operations such as task checkpointing and task restarting and a set of analytical tools such as computing probability distribution based on experimental data. Excessive tasks submitted to the system will be put in a queue and scheduled according to FCFS policy. The checkpoint/restart mechanism is implemented with an optimized checkpointing interval calculated by Young's formula [20].

We focus on four evaluation issues listed below.

- What is the peak processing ability of our simulation system (or the maximum number of tasks to process simultaneously)?
- What is the probability of WPR when Google tasks are submitted according to Google job arrival rates?
- How many tasks are running in the system, and how many tasks are finished over time?
- What is the scalability of the simultaneous checkpointing on our designed DM-NFS versus traditional NFS?

B. Evaluation Results

In Figure 8, we present the number of running tasks and the number of finished tasks over time in two simulation tests, where the lengths of sample tasks selected are limited within one hour (i.e., 3,600 seconds). There are 1,000 tasks and 2,000 tasks submitted at the beginning of the two tests, respectively, and they are submitted one by one every 0.5 second. The figure clearly shows that the simulation system in the two cases can simultaneously process about 800 tasks and 1,200 tasks, respectively. The main reason our system can process such a large number of tasks simultaneously is that the memory consumption per Google task is usually small, according to the trace, as shown in Figure 8(b). The number of running tasks cannot reach 1,000 and 2,000, respectively, because quite a few tasks are finished quickly (e.g., several minutes) due to short lengths.

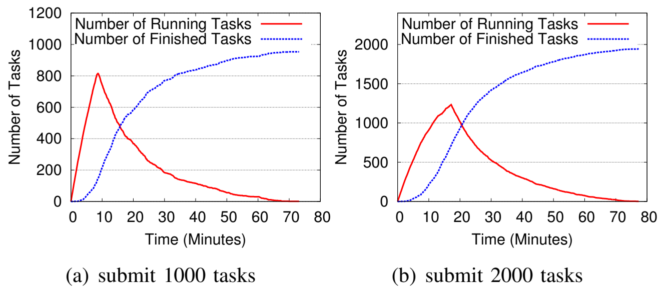
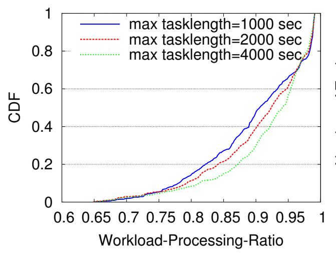


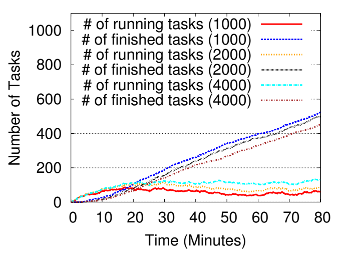
Figure 8. Peak Parallel Degree in Processing Tasks

We also evaluate the checkpointing effect in three cases with various maximum task lengths. In the three cases, the tasks in the test are randomly generated based on sample Google tasks, whose task lengths are limited in 1,000 seconds, 2,000 seconds, and 4,000 seconds, respectively. The tasks generated are submitted to the system, and their arrival intervals are based on the real job arrival dates recorded in the Google trace. We show the evaluation results in Figure 9, where the numbers in parentheses refer to maximum task lengths set in the experiments. We present in Figure 9(a) the probability (CDF) of WPR for different cases with different maximum task lengths. The figure shows that the checkpointing mechanism makes a majority of tasks gain a high workload processing ratio (≈90%). Figure 9(b) presents the number of running tasks and the number of finished tasks over time in the three cases. By comparing this figure with Figure 4, we find that the number of concurrently running tasks (about 100 tasks) in such an experiment with real job arrival rates is far less than the peak processing ability.

Table IV shows the differences of simultaneous check-



(a) CDF of WPR



(b) Running/Finished Tasks

Figure 9. Evaluation Based on Real Job Arrival Rate in Google Trace

pointing costs between traditional NFS and our DM-NFS, confirming the fairly high scalability of our design. The checkpoint cost increases linearly with the number of simultaneous checkpoints made on traditional NFS, while it is kept constant on our designed DM-NFS.

Table IV
COST OF SIMULTANEOUS CHECKPOINTING TASKS ON TRADITIONAL NFS AND DM-NFS (SECONDS).

type \ parallel degree		N=1	N=2	N=3	N=4	N=5
traditional 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
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

Our simulation system has been successfully applied to the fault-tolerance research on Google task processing, which was published in the Supercomputing 2013 conference [18]. More evaluation results about fault tolerance issues can be found in that paper.

V. Related Work

Traditional simulation research is focused on how to construct a simulated large-scale environment that can run various unchanged applications. For example, ModelNet [25] and MicroGrid [26] are two simulation platforms that can run MPI, C, C++, Perl, and/or Python programs. However, researchers still have to implement cloud applications themselves when using such simulation systems.

In recent years, many easy-to-use toolkits for simulating distributed-computing environments. GridSim [2] is a Grid simulation toolkit that can help researchers simulate a distributed environment with millions of resources and thousands of users based on varied requirements. It integrates several Grid features such as the simulation of network communication, usage of various resources, and virtual organizations. SimGrid [3] is another excellent Grid simulator that has been widely used in Grid/P2P research for years. It is able to simulate arbitrary network topologies and dynamic compute and network resource availabilities, as well as resource failures. Peersim [4] is an easy-to-use toolkit based on which peer-to-peer(P2P) researchers can easily evaluate their P2P protocols or algorithms with various network delays and bandwiths. NS-2 [5] is another outstanding tooklit for simulating network structure. In comparison with these works, our GloudSim aims at constructing a close-to-practice execution environment in the context of cloud data center based on a real production trace provided by Google. Although Google trace [11], [12] does not provide any intra- or inter-host network information, our simulation system can still reproduce a convincing execution environment relative to resource usage and availability, various user constraints, checkpointing cost, and task events.

Recently, many Grid simulation toolkits have integrated the VM concept to support the simulation in the context of cloud computing. The latest version of SimGrid [3] has been able to support the simulation of VM technology such as VM resource customization and seamless VM migration. The CloudSim [27] toolkit is another cloud simulator designed with a number of cloud features such as the simulation of VM technology and autonomic/cloud economy. These two toolkits are easy to use because they can be performed even on a desktop computer. However, such simulation works based on hypothetical objects such as theoretically controlled resources and tasks may not represent some practical cases. In fact, the cloud environment is so complex that the task features and resource dynamics may not fit any theoretical models. For instance, the failure probability density function of Amazon cloud spot instances [28] is not only dependent on task length but relative to user bids. The distinct feature of our simulation system is that we devise and implement a toolkit that can reproduce the execution environment based on Google trace as precisely as possible, by combining the real VM deployment, checkpointing tool and real-world trace data into a real cluster testbed. Our simulation toolkit will likely be more attractive to many researchers who want to study cloud data centers with practical deployment instead of just numerical emulation.

VI. Conclusion and Future Work

In this paper, we devise a close-to-practice cloud simulation system, GloudSim, based on a one-month Google trace that was produced with thousands of applications and millions of jobs/tasks running across over 12,000 heterogeneous hosts. To the best of our knowledge, this is the first attempt to construct an easy-to-use toolkit based on a real-world production cloud trace, which we believe is interesting to cloud computing researchers. Researchers can download our toolkit for free under a GNU GPL v3 license. In this paper, we present an overview of our design and then discuss many technical details of the implementation. Through experiments, we find that a Google task's CPU usage can be characterized by running a Java program with a while-loop based on three tunable parameters. The task checkpoint overhead can be precisely emulated by using the while-loop with periodic sleeps inside but with null operations. Various memory sizes consumed by tasks can be emulated by preloading data files in different sizes. We also propose a novel approach for lowering the probability of VM memory exhaustion. Experiments show that our simulation system can simultaneously process up to 1200 jobs in parallel, and users can clearly observe runtime states like the number of running tasks and finished tasks over time. This simulation system has been successfully used to perform the experiments for fundamental fault-tolerance research on optimization of Google task checkpoint intervals. We plan to extend our simulation system to fit more scenarios in addition to fault-tolerance.

Acknowledgments

This work is supported by the projects ANR RESCUE 5323, *Illinois-INRIA-ANL* Joint Laboratory on Petascale Computing, Hong Kong RGC Grant HKU-716712E, and also supported in part by the U.S Department of Energy, Office of Science, under contract DE-AC02-06CH1135.

References

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

[2] R. Buyya and M. Murshed, "GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing," *Journal of Concurrency and Computation: Practice and Experience (CCPE)*, vol. 14, no. 13, pp. 1175–1220, 2002.

[3] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: A generic framework for rarge-scale distributed experiments," in *Proceedings of the 10th International Conference on Computer Modeling and Simulation (UKSIM'08)*, 2008, pp. 126–131.

[4] Alberto Montresor and Márk Jelasity., "PeerSim: A scalable P2P simulator," in *Proceedings of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pp. 99–100, Seattle, Sept., 2009.

[5] S. Mccanne, S. Floyd, and K. Fall, "The Network Simulator NS-2", http://www.isi.edu/nsnam/ns/.

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

[7] Amazon EC2: on line at http://aws.amazon.com/ec2/.

[8] S. Chaisiri, B-S. Lee and D. Niyato, "Optimal virtual machine placement across multiple cloud providers," in *Asia-Pacific Services Computing Conference*, 2009, pp. 103–110.

[9] M. Buyya, S.Y. Chee, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities," in *10th IEEE International Conference on High Performance Computing and Communications (HPCC'08)*, 2008, pp. 5–13.

[10] S. Di and C.-L. Wang, "Dynamic optimization of multi-attribute resource allocation in self-organizing clouds," in *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 2012.

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

[12] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, Technical Report, Nov. 2011, revised 2012.03.20. Posted at URL http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004, pp. 137–150.

[14] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in Google compute clusters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, New York, ACM, 2011, pp. 3:1–3:14.

[15] 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)*, 2012, pp. 230–238.

[16] 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, Technical Report ISTC-CC-TR-12-101, April 2012.

[17] *Gloudsim v1.0*: online at https://code.google.com/p/gloudsim

[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 *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013, pp. 1–11.

[19] P.H. Hargrove and J.C. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, pp. 494, 2006.

[20] J.W. Young, "A first order approximation to the optimum checkpoint interval," *Communications ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[21] J.T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computater Systems*, vol. 22, no. 3, pp. 303–312, 2006.

[22] Xen-credit-scheduler: on line at http://wiki.xensource.com/xen wiki/creditscheduler.

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

[24] 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)*, New York: ACM, 2003, pp. 164–177.

[25] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Bechker, "Saclability and accuracy in a large-scale network emulator, " in *Proceedings of the 5th symposium on Operating Systems Design and Implmentation (OSDI'02)*, pp. 271–284, New York, 2002.

[26] H. Xia, H. Dail, H. Casanova, and A.A. Chien, "The MicroGrid: Using online simulation to predict application performance in diverse Grid network environments," in *2nd international workshop on Challenges of Large Applications in Distrubted Environments (CLADE'04)*, CA, 2004.

[27] R. Calheiros, R. Ranjan, A. Beloglazov, C.A.F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience (SPE)*, vol 41, no. 1, pp. 23–50, ISSN: 0038-0644, Wiley Press, New York, January 2011.

[28] S. Yi, A. Andrzejak, and D. Kondo, "Monetary cost-aware checkpointing and migration on Amazon cloud spot instances," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 512–524, 2012.