

GloudSim: Google Trace based Cloud Simulator with Virtual Machines

Sheng Di¹, Franck Cappello^{2*}

¹MCS, Argonne National Laboratory, IL, USA, sdi1@anl.gov

²MCS, Argonne National Laboratory, IL, USA; University of Illinois at Urbana-Champaign, Champaign, IL, USA, cappello@mcs.anl.gov

SUMMARY

In 2011, Google released a one-month production trace with hundreds of thousands of jobs running across over 12,000 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 $16 \times 8 = 128$ cores and 112 virtual machines (VMs) constructed by XEN hypervisor. To the best of our knowledge, this is the first work to reproduce Google cloud environment with real experimental system setting and real-world large scale production trace. Experiments show that our simulation system could effectively reproduce the real checkpointing/restart events based on Google trace, by leveraging Berkeley Lab Checkpoint/Restart (BLCR) 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. Copyright © 2013 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Cloud Computing; Simulation System; Google Cluster; Google Trace

1. INTRODUCTION

With increasing demands on computing resources over the Internet and fast development of virtual resource isolation technology [1], cloud computing [2] has become a compelling paradigm that can provision elastic services and on-demand resources for various users. An easy-to-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 [3, 4] and P2P/network simulators [5, 6], 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, many cloud systems allow users to customize their own particular execution environments, e.g., in the form of virtual machine (VM) instances [7], each of which is allocated with multiple types of on-demand resources such as CPU rate and memory sizes.

*Correspondence to: Sheng Di, MCS Division, Building 240, Argonne National Laboratory, 9700 S. Cass Avenue, Lemont IL 60439, USA, sdi1@anl.gov

- 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 Amazon EC2 [8] is dependent on user bids and payment model; quite a few optimization strategies [9, 10, 11] on cloud resource allocation require users to provide preferred payment budget in advance; a task's scheduling priority in a Google data center [12, 13] is mainly determined by its execution priority* 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 [14], for example, is a kind of I/O-intensive distributed program. Google trace [12] 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 [13] 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†.

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 a lot of existing research [15, 16, 17] 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 reproduce 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. We summarize the key contributions below.

- GloudSim toolkit is able to reproduce the task behaviors according to a real cloud trace (Google trace), for the purpose of further research. The researchers of fault tolerance can reproduce the Google tasks such that their resource consumptions are very close to the values in the trace, as well as various types of events (e.g., kill and evict). The researchers who are interested in the scheduling problem can implement/customize their own designed schedulers and run the simulation only based on the task/machine constraints and job priorities presented in trace. In this situation, the resource consumption (e.g., memory footprint and CPU rate) does not strictly conform to the Google trace, but follows the practical system resource usages which are dependent on some scheduling policy. The researchers who are interested in large scale or long-term simulation can make use of the statistical toolkit in GloudSim to run the simulation on a larger number of machines and for a longer period, instead of being restricted to 12,000 machines and one-month period presented in the trace. All in all, which part of the trace is to be reproduced is determined by user's demand. Users can selectively make use of the GloudSim toolkit to construct their own research environment.
- GloudSim toolkit provides two flexible simulation modes – numerical simulation and real-system-setting simulation – in order to suit different research demands. For the numerical simulation, the simulator can be executed on a desktop computer, simulating the Google jobs/tasks according to the trace, in terms of the event-driven model. For the real-system-setting simulation, the simulator can be run on a real cluster environment

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

†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.).

with virtual machines (VM), using real-world resource isolation technologies and system threads/processes.

- We provide novel insight on how to effectively emulating Google jobs/tasks' behaviors and resource consumptions for the real-system-setting simulation. 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 checkpoint 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.
- We build/evaluate the simulation effect using a real cluster with hundreds of VM instances and Berkeley Lab Checkpoint/Restart tool (BLCR) [20]. Its effectiveness is evaluated through the well-known checkpointing theory Young's formula [21], as well as well-known scheduling policies like First-Come-First-Serve (FCFS). In addition to FCFS, the GloudSim users can also implement their own schedulers, such as Priority-based Scheduling, Progress-based Scheduling, Workload-based Scheduling, etc. As for the experimental results, we present the checkpoint overhead suffered by Google tasks with BLCR [20], the checkpoint file size versus memory size consumed, the operation time of setting checkpoints, the scheduling states of the system when submitting different numbers of tasks, etc. All of the experimental results are supposed to be very valuable to the further cloud research.

This simulation toolkit has been released as a GNU GPL v3 software [18] for free downloading. All of modules in the toolkit are loosely coupled. For instance, the characterization of Google trace is separate from the trace. If the user wants to perform the simulation based on another cloud trace, he/she just needs to modify/improve the trace analyzer module in our software package to fit their flavor. Moreover, the entire software is 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 SC'13 conference [19]. We believe that our work is very helpful for researchers to customize a close-to-practice simulation system with their particular demands.

The remainder of the paper is organized as follows. We present the design overview of our simulation system in Section 2. 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 3, we discuss some key technics 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 estimate/compute experimental job/task's productive length and wall-clock length. In Section 4, 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 5, and conclude with a vision of the future work in Section 6.

2. DESIGN OVERVIEW

In this section, we briefly introduce Google trace [12, 13], and present a design overview in building a simulation environment based on the Google trace.

2.1. 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 centrally-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, an 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).

In the Google trace [13], 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 (such as minimal memory size required and the minimal CPU rate requested), 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 being finished, it will enter into a dead state, and it can be resubmitted by users later on.

It is worth noting that crash-loop phenomenon is very common in the Google task execution. Based on the previous report about Google trace [22], about 71% (=10M/14M) of task failures recorded in the trace are in three crash looping jobs. That is, for any task with one failure in the trace, it will likely experience multiple failures in its whole execution, being restarted automatically in case of failures on different hosts to continue the workload processing.

The applications targeted by our simulator are the 4,000 applications used in the Google trace. For the purpose of confidentiality, the detailed application names are not disclosed but presented as hash codes. Although we have no ways to know what applications they are, we can still characterize their features, including whether they are parallel applications or not, how much resource they consume, what kinds of constraint they are subject to, the execution scales, the application lengths, and so on. More details can be found in our previous characterization work [16, 23]. As for the service applications such as web server, the current release of the GloudSim is unable to provide an appropriate simulation, in that web services are often deployed on web servers such that they cannot be formulated as particular jobs with limited lengths. Whereas, for any other applications which can be used to submitted in the form of jobs, the users can leverage the GloudSim to perform the simulation and leverage the information we provided in the toolkit, including task length, workload, priority, resource usage, etc.

Indeed, the current GloudSim toolkit still has a few limits. In addition to application types, some other information are also hidden in the Google trace. For example, all of floating-point values are normalized by its theoretical maximum value. That is, they are always transformed in a linear manner, for keeping the validity of the trace well. For example, 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.

In addition, there is no information about network and electricity power cost in the Google trace. However, GloudSim users can still emulate such information in terms of the real cluster environment to use or some cost consumption model, because the resource consumption information (such as CPU rate and memory size consumed by each task) can be reproduced by GloudSim based on the trace. For example, when reproducing the Google tasks, GloudSim simulator tries best to guarantee the CPU rate and memory size consumed by the task are consistent with the trace. So, the GloudSim users can also use some real equipments/detectors to measure the energy cost based on the constructed environment on a real cluster. If the GloudSim users are interested in bandwidth information, they could also modify the source code of simulated task in our GloudSim to transmit different types of messages among different tasks.

2.2. Overview of the GloudSim Simulation System

GloudSim has two simulation modes, numerical event-driven simulation and real-system-setting simulation. As for the former, the design principle is similar to contemporary simulation toolkits such as PeerSim [5]. During the simulation, various types of events like task evict and task termination will be numerically generated and handled by the simulator, which will change the corresponding attributes of the "running" job/task objects. In addition to the numerical simulation, we also explore how to reproduce the task behaviors on a real cluster environment, so as to provide

a more convincing testbed for cloud researchers. In the following text, we mainly focus on the real-system-setting simulation mode.

The whole system architecture of the real-system-setting simulation can be split into two parts based on the Google trace [13], a job scheduling server and a set of task execution hosts (or slave nodes). We need to design and implement a scheduler which can run different scheduling policies to automatically process emulated Google jobs/tasks. 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.

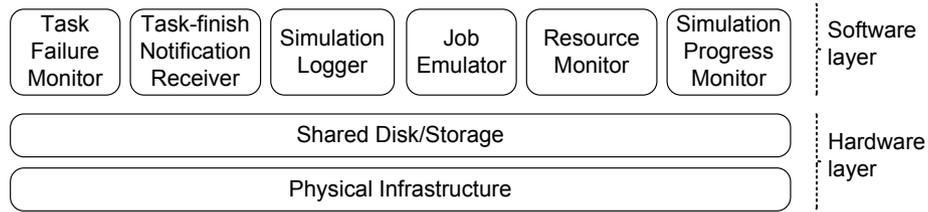


Figure 1. Design Architecture on the Server

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 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. Whenever it is required to set a checkpoint for a running task in a VM instance using GPFS, one of NFS servers will be randomly selected for storing its checkpoint file. In fact, our software can also be used conveniently with other distributed file systems like Parallel Virtual File System (PVFS) [24] or Hadoop Distributed File System (HDFS) [25]. What is most attractive in our DM-NFS framework is its ease-of-use and high effectiveness of dispersing bottleneck issue to save checkpoint files. The effectiveness of the DM-NFS is confirmed by our experiment, to be shown in Section 4.

In the software layer, there are six key modules to perform the whole simulation work together. The six modules have no particular layer relationship. They are just separate modules, each of which can communicate with each other as well as the hardware layer. 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: In order to generate a job/task, we have to know its length (or workload to process) based on the trace. The trace provided, however, just lasts one-month period, such that some tasks were not finished in the end. In order to generate the jobs/tasks with correct lengths, we have to ignore these “unfinished” jobs/tasks. 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. GloudSim generates the 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 can be used to make the simulation be consistent with the original Google trace.
4. Job/task scheduler: any generated job needs to be scheduled through a scheduler before its execution. More specifically, the jobs/tasks are generated based on the arrival times presented in the trace, and they will be executed on some idle/available VM instances. If there are not enough resources (e.g., the available remaining resource amounts are less than task constraints because of over-task-submit) in the simulation environment, the tasks have to be queued in the server and to be scheduled based on some policy later on (which tasks will be allowed to run first). Google trace provider hides the scheduling strategies in the trace. However, with the GloudSim simulator, the users can easily implement their own particular task scheduler. Our toolkit provides an example policy, First-Come-First-Serve (FCFS), for users' reference. Under FCFS, the jobs/tasks will be scheduled based on the arrival times when the overall system is running in a busy state. The resources assigned to scheduled tasks are still according to Google trace.

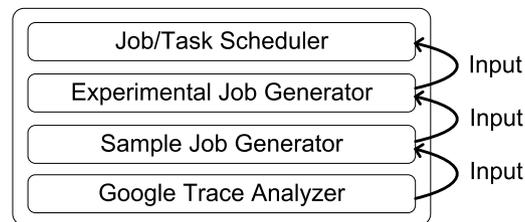


Figure 2. Architecture of Job Emulator

- *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.
- *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 generate analytical results based on the observed data.

The design architecture of the execution end (i.e., slave 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. GloudSim toolkit provides easy-to-use interfaces and scripts to start, stop, migrate VMs in a batch way. It is also able to help split the physical resources and memory sizes based on Google task usage shown in the trace. When some hosts or VMs are down, the "failed" tasks will be restarted on some other available VMs and our toolkit leverages BLCR to perform task checkpointing/restarting. That is, GloudSim will not migrate VMs among physical nodes inside the cluster, because our objective is to reproduce task execution and failures and VM migration may inevitably impact the task simulation and resource consumption. If users want to perform VM migration based on specific needs, XEN4.0 already provides simple interfaces to do that, and our toolkit provides simple interfaces in GloudSim to call any of XEN's commands.

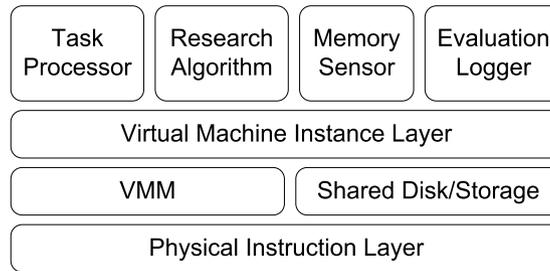


Figure 3. Design Architecture on Execution Hosts

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 [13]. (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 and some new scheduling policy. In the released toolkit, we provide the implementation of Young's formula [21] as a checkpoint case, and also implement a FCFS scheduling policy.
- *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.

3. KEY TECHNOLOGIES IN GLOUDSIM TOOLKIT

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.

3.1. 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.

3.1.1. Characterization of Google Task Failure Events and Resource Utilization

Before investigating how to precisely reproduce Google task execution features, it is necessary to characterize the Google task events and resource utilization based on the trace data.

In order to reduce user's burden, our toolkit includes a set of statistics about failures like failure intervals. The mean time between failures (MTBF) is the most important metric in that many checkpointing theories are based on its value, e.g., Young's formula [21] and Daly's work [26]. MTBF here includes both software failures and hardware failures. This is different from high performance computing (HPC) systems like Cluster and Grid computing. For HPC, MTBF is usually computed based on the frequencies of the hardware crashes. For Google cloud, as we observed based on the trace, task failures may occur very often, which means that the major reason for Google task failures is not hardware but other factors like priority, resource consumption, software issues, etc. That is, for Google cloud tasks, we need to take into account any interruption events to compute the MTBF.

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.

Table I. MTBF of Google Tasks (in Seconds)

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	-

We also present in Figure 4 the distribution (cumulative distribution function (CDF)) of the uninterrupted work intervals of a given task. The uninterrupted failure interval refers to the consecutive productive time without interruptions or failures. If a task has no failures, then the uninterrupted failure interval is equal to its execution length. 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.

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)). The important metrics (such as MTBF) used to compute the optimal checkpoint intervals for a particular task are estimated based on the task's characteristic and statistics. For instance, the MTBF of a particular task will be estimated based on the task's priority and productive time (execution length). Specifically, we first categorize all sample jobs in the trace into 12 groups based on 12 priorities and into 8 groups based on their execution lengths. Then, a task's MTBF is set to the mean value of the MTBF computed based on its corresponding group (with the same priority and the range of execution length). In addition

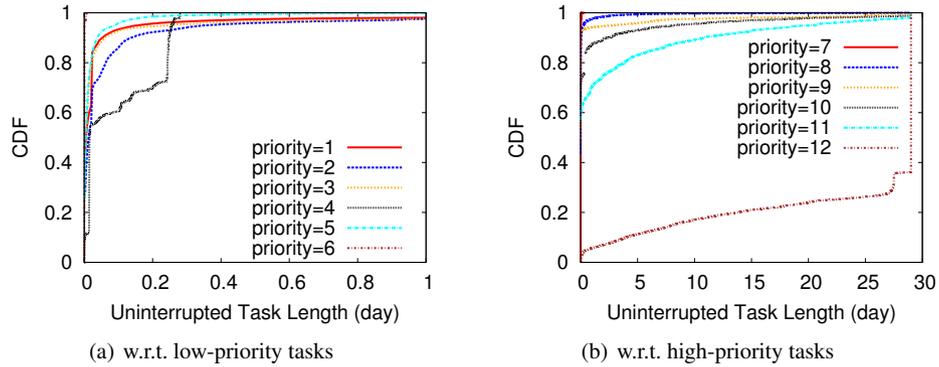


Figure 4. Distribution of Google Task Failure Intervals According to Priorities.

to priority and execution length, the CloudSim users can also extend the toolkit to compute more MTBF values based on other metrics, like scheduling class and task constraints.

In addition to failure events, 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 5. 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 5, 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 5 (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 5 (b), it is also observed that majority of tasks consume very small memory size per task at runtime.

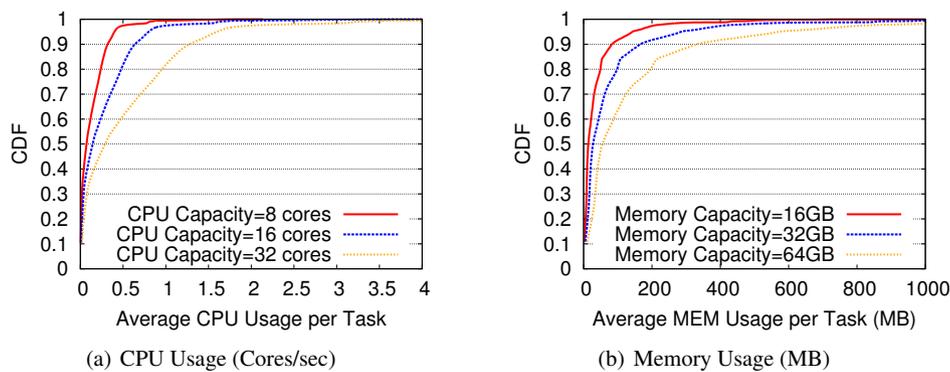


Figure 5. Distribution of CPU and Memory Usage per Task

3.1.2. 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 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 [27]. 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.

3.1.3. 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

3.1.4. Simulation of Task Length and Checkpoint Overheads

In general, it is nontrivial to precisely emulate a particular task’s execution length and checkpoint cost/overhead based on the trace data. The execution length (or productive time, which excludes the cost about failure events) of the task is supposed to be equal to the summed time of the aliveness of its corresponding running process. That is, the real working time of the process should be controlled as closely as possible to the task’s total productive time recorded in the trace. A straightforward idea is to simulate each task by launching a thread/process and letting it simply sleep for a specific time until the task ends according to the trace. Such an approach can simulate the task execution length precisely, but it ignores the computational resource consumed (e.g., the CPU rate and memory size allocated).

In our simulation system, we also need to reproduce the resource amounts consumed by each task because the checkpoint/restart overheads of storing/reading checkpoint files into/from disks are closely related to them, e.g., memory size. That is, if we ignore the resource usage of tasks, then checkpoint overhead in our simulation will be largely skewed from the practical cases. Hence, we need to characterize the relationship between checkpoint overheads and resource consumption in practice.

We characterize the checkpoint overhead and operation time based on BLCR, with various percentage of CPU utilization and different memory sizes, as shown in Figure 6. 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 6 we see that the checkpoint overhead does not change noticeably with different CPU utilization but increases linearly with the memory size.

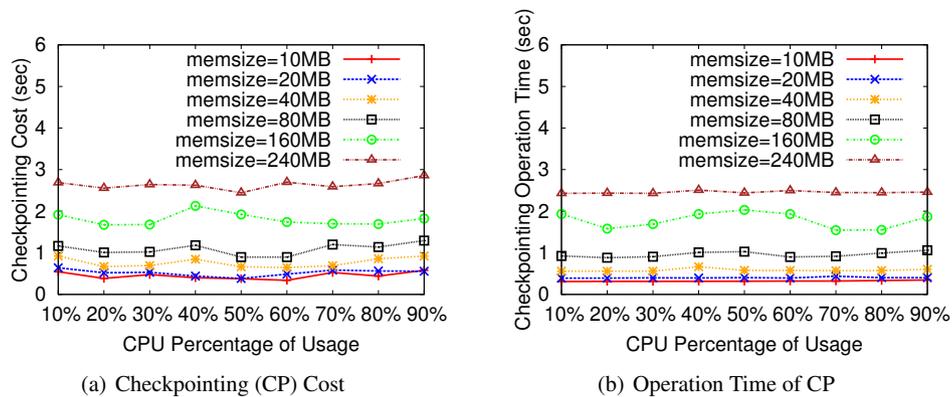
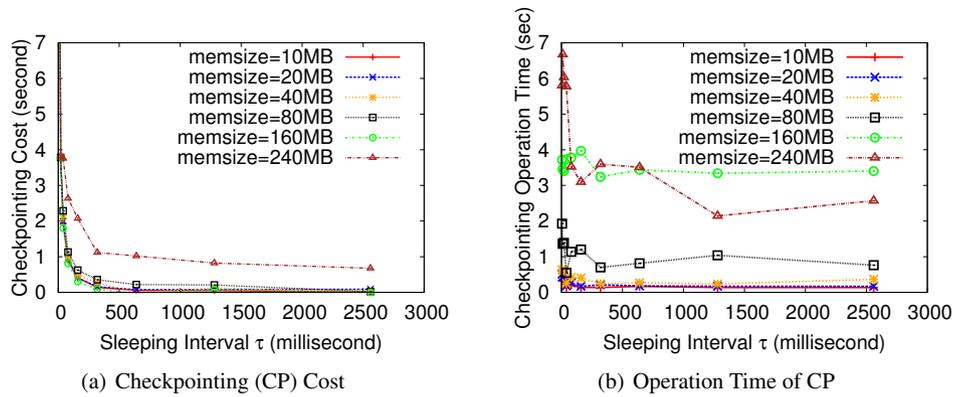


Figure 6. Checkpoint Overhead with Various CPU Usage and Memory

Based on the figure, if one just needs to focus on the 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 single-instruction operations (single instruction such as *plus 1*) 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 can be easily customized by letting the process sleep every τ milliseconds.

In Figure 7, we present the checkpoint overhead based on BLCR, by running a Java program with different sleeping intervals. In Figure 7, 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 $\tau=100$ milliseconds (highly recommended setting) will fit the true values shown in Figure 6 well.

Figure 7. 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 $100 \times 1000 \times 1.03$ 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 $1000 \times \frac{1}{1.03} \approx 971$.

3.1.5. Effective Simulation of Task Failure and Task Rescheduling

In Figure 8, 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.

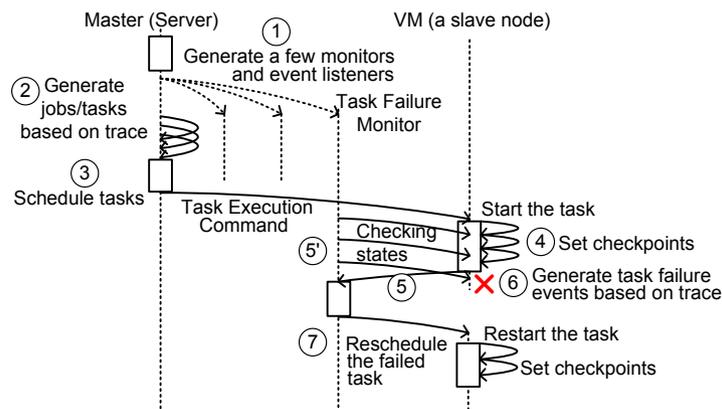


Figure 8. Procedure of Task Failure Detection

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 8.
- 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.

3.2. 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 periodically detects each VM instance's instant free memory, by communicating 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).

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

Such a formula can significantly reduce the probability of memory exhaustion on VM instances, effectively avoiding the unexpected VM failures. For example, as 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.

3.3. Assessment of Task Execution Efficiency

How to assess task execution efficiency is a key issue in the simulation system. This is mainly related 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 (or real productive time), and then it will be immediately terminated. Such a method can guarantee a precise estimate of each task's real productive time in the system.

We provide a particular indicator to evaluate task's execution efficiency, namely, the *workload processing ratio (WPR)*. WPR is defined as the ratio of the productive time to wall-clock time, as shown in Formula (2). It can be used to evaluate the working efficiency of the checkpointing mechanism designed in the simulation.

$$WPR(J_i) = \frac{J_i's \text{ workload processed}}{J_i's \text{ real wall-clock length}} \quad (2)$$

4. PERFORMANCE EVALUATION

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

4.1. Experimental Setting

We evaluate our simulation system based on some use-cases with a powerful cluster called Gideon-II [28], which is located in Hong Kong. In general, researchers prefer to using a medium-sized cluster to perform close-to-practice simulation based on real cloud traces. In our evaluation, there are 16 physical hosts, each of which has 8 cores and 16 GB in memory capacity. Such a scale is already quite enough for most of the researchers to perform their own simulations. Each host is deployed with XEN 4.0 hypervisor [29] 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 [21]. Young's formula is given by Equation (3), where T_c , C , and T_f refer to the optimized checkpointing interval, checkpointing overhead (as shown in Figure 7), and the mean time between failures (MTBF) respectively.

$$T_c = \sqrt{2 C T_f} \quad (3)$$

Based on trace, we observe that there are three types of job structures, including sequential-tasks (ST) job, bag-of-tasks (BoT) job, and the job with mixture of them. As for the sequential-tasks job, we can observe that the job's tasks are never executed in parallel, but always executed in series. More specifically, based on the tracing of the tasks belonging to the same job based on their task IDs, we observe some job's multiple tasks are executed sequentially. It is likely that this job actually performs the same work, but the execution is interrupted/restarted very often. It runs on some node at the beginning for a while, and then disappears from that node but is restarted on another node, and so on so forth. We call such a job "sequential-tasks job", because this job has multiple tasks which are connected sequentially. In comparison to the sequential-tasks job, bag-of-tasks job also has many tasks, but we can observe that all of the tasks start simultaneously and also terminate at the same time. That is, the tasks of a bag-of-tasks job always run in parallel. In addition, there is another type of job which mixes the above two types - we call it mixture-of-both. For this kind of job, the tasks start simultaneously but end at different times, and there are also some task interruptions during each task execution.

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 9, 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.

Since we already discussed and presented some evaluation results in previous sections (such as the effect of emulating task events in Section 3.1.2 and the simulation results about task length and checkpoint overhead in Section 3.1.4), we will mainly focus in this section on the 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?

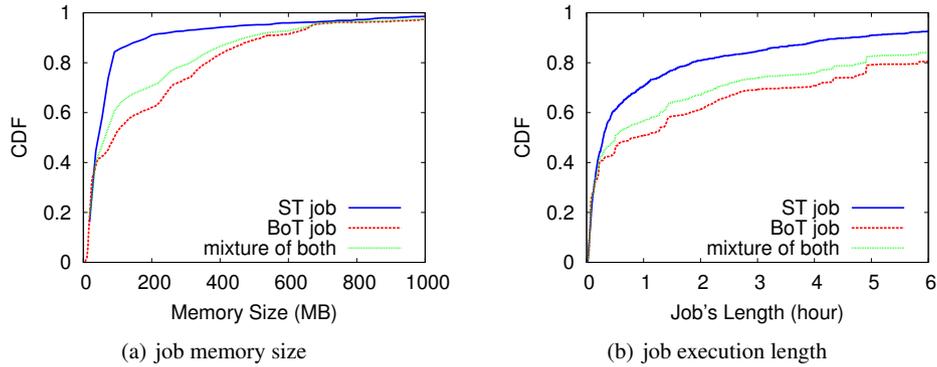


Figure 9. Distribution of Google Jobs: Memory Size and Execution Length.

4.2. Evaluation Results

In Figure 10, 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 10(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.

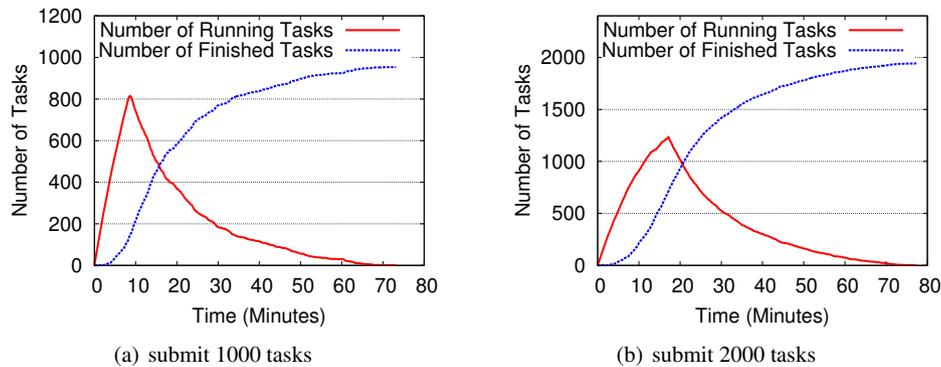


Figure 10. Peak Parallel Degree in Processing Tasks

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 11, where the numbers in parentheses refer to maximum task lengths set in the experiments. We present in Figure 11(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 ($\approx 90\%$). Figure 11(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 5, 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.

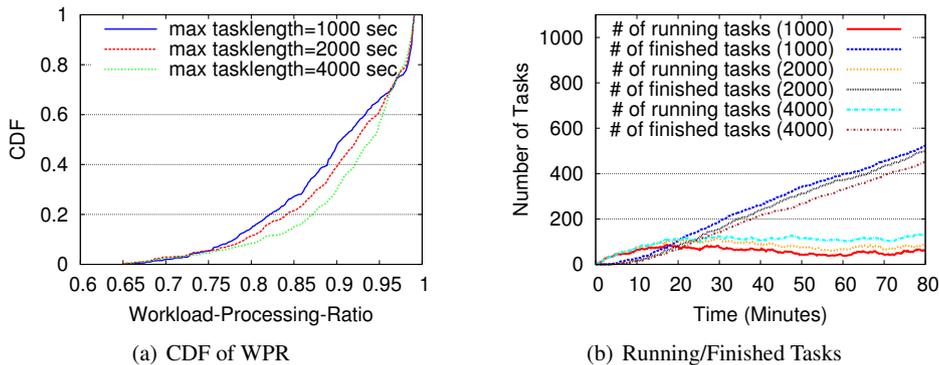


Figure 11. Evaluation Based on Real Job Arrival Rate in Google Trace

Table IV shows the differences of simultaneous checkpointing 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 (in 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 SC'13 conference [19]. More evaluation results about fault tolerance based on the GloudSim simulator can be found in that paper.

5. 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 [30] and MicroGrid [31] 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 are implemented for simulating distributed-computing environments. GridSim [3] 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 [4] 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 [5] 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 bandwidths. NS-2 [6] is another outstanding toolkit for simulating network structure. In comparison with these works, our GloudSim aims at constructing a close-to-practice execution testbed in the context of cloud data center based on a real production trace provided by Google. Although Google trace [12, 13]

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 [4] has been able to support the simulation of VM technology such as VM resource customization and seamless VM migration. The CloudSim [32] 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 and numerically controlled resources. Hence, some researchers combined real-world traces and some simulator to perform the experiments for their research, such as [33]. However, the cloud environment is still so complex that the task features, resource dynamics, energy consumption, and execution performance may not simply fit existing theoretical models or may be determined by multiple factors. In this situation, researchers may want to use a real cluster testbed to perform the emulation, in order to generate more convincing evaluation results.

There are some existing related works on constructing cloud environment with given traces, while they are mainly designed for particular applications like MapReduce without the concept of virtual machines. GridMix [34] is a benchmark for Hadoop clusters. It can generate Hadoop jobs and run them on a new cluster, based on some existing Hadoop trace. Chen et al. [35] built the case for evaluating MapReduce performance using workload suites. They proposed a framework to synthesize and execute representative workloads. Benchlab [36] is an open testbed that uses real Web browsers to measure the performance of Web applications.

With GloudSim, the users can quickly build an either numerical simulation environment or a real-system-setting testbed based on Google trace with thousands of applications, significantly reducing users' burden in performing experiments. *EMUSIM* [37] is a similar simulation system regarding virtual machines, which can automatically extract information from application behavior via emulation and uses this information to generate the corresponding simulation model. The distinct feature of our simulation system is three-fold:

- 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. Google tasks are good for in-depth research on cloud computing, because Google trace [13] provides up to 4,000 different types of applications.
- GloudSim is very flexible to be extended based on researchers' demands. We provide easy-to-use interfaces and scripts to easily extract the information from the Google trace (based on the trace format - CSV format). If there are other traces available to use and they are consistent with CSV format, GloudSim can extract the key information used for simulation, such as resource usage, task length, etc.
- GloudSim adopts VM technology to split various resources like CPU rate and memory sizes, which is consistent with well-known VM mechanisms used by Amazon EC2. The GloudSim users are allowed to provide their own implementations on some key components, such as their own resource allocation schemes like spot instances design, payment policies like pay-by-use, task scheduling methods like FCFS, and checkpoint policies based on their research purpose. Our toolkit already assists users to finish many key steps, like building the execution environment with Google job/tasks, reproducing resource usages and failure events, performing BLCR checkpoint mechanism and XEN VM facilities, etc.

All in all, the GloudSim toolkit is expected to be more attractive to many researchers who want to study cloud data centers with practical deployment instead of just numerical simulations.

6. CONCLUSION AND FUTURE WORK

In this paper, we devise an ease-of-use and 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 public attempt to construct an easy-to-use toolkit based on a real-world production cloud trace (Google trace), which we believe is fairly interesting and helpful 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 execution length 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. In the future, we plan to extend our simulation system to fit more scenarios in addition to fault-tolerance.

ACKNOWLEDGEMENT

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-AC02-06CH11357.

REFERENCES

1. Gupta D, Cherkasova L, Gardner R, and Vahdat A. Enforcing performance isolation across virtual machines in XEN. *Proceedings of the 7th ACM/IFIP/USENIX International Conference on Middleware (Middleware'06)*, 2006; 342–362.
2. Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M. Above the clouds: A Berkeley view of cloud computing. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, february 2009.
3. Buyya R, Murshed M. 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)*, 2002; **14**(13): 1175–1220.
4. Casanova H, Legrand A, Quinson M. SimGrid: A generic framework for range-scale distributed experiments. *Proceedings of the 10th International Conference on Computer Modeling and Simulation (UKSIM'08)*, 2008; 126–131.
5. Montresor A, Jelasity M. PeerSim: A scalable P2P simulator. *Proceedings of the 9th International Conference on Peer-to-Peer (P2P'09)*, September 2009; 99–100.
6. Mccanne S, Floyd S, Fall K. The Network Simulator NS-2. <http://www.isi.edu/nsnam/ns/>.
7. Smith J, Nair R. Virtual Machines: Versatile Platforms For Systems and Processes. Morgan Kaufmann, 2005.
8. Amazon EC2. <http://aws.amazon.com/ec2/>.
9. Chaisiri S, Lee B, Niyato D. Optimal virtual machine placement across multiple cloud providers. *Asia-Pacific Services Computing Conference*, 2009; 103–110.
10. Buyya M, Chee S, Venugopal S. Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. *10th IEEE International Conference on High Performance Computing and Communications (HPCC'08)*, 2008; 5–13.
11. Di S, Wang C. Dynamic optimization of multi-attribute resource allocation in self-organizing clouds. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, March 2013; **24**(3): 464–478.
12. Wilkes J. More Google cluster data. Google research blog. <http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html> [November 2011].
13. Reiss C, Wilkes J, Hellerstein J. Google cluster-usage traces: format + schema. Google Inc., Mountain View, CA, Technical Report, revised 2012.03.20. <http://code.google.com/p/googleclusterdata/wiki/TraceVersion2> [November 2011].
14. Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004; 137–150.
15. Sharma B, Chudnovsky V, Hellerstein J, Rifaat R, Das C. Modeling and synthesizing task placement constraints in Google compute clusters. *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, New York, ACM, 2011; 3:1–3:14.

16. Di S, Kondo D, Cirne W. Characterization and comparison of cloud versus grid workloads. *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'12)*, 2012; 230–238.
17. Reiss C, Tumanov A, Ganger G, Katz R, Kozuch M. 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.
18. *Gloudsim v1.0*: <https://code.google.com/p/gloudsim/> [November 2013].
19. Di S, Robert Y, Vivien F, Kondo D, Wang C, Cappello F. Optimization of cloud task processing with checkpoint-restart mechanism. *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013; 64:1–64:12.
20. Hargrove P, Duell J. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 2006; **46**(1):494.
21. Young J. A first order approximation to the optimum checkpoint interval. *Communications ACM*, 1974; **17**(9): 530–531.
22. Reiss C, Tumanov A, Ganger G, Katz R, Kozuch M. Towards understanding heterogeneous clouds at scale: Google trace analysis. Technical Report. *Intel science and technology center for cloud computing, Carnegie Mellon University*, ISTC-CC-TR-12-101, 2012.
23. Di S, Kondo D, Cappello F. Characterizing and modeling cloud applications/jobs on a google data center. *journal of supercomputing*, 2014; **69**: 139–160.
24. Carns P, Ligon W, Ross R, Thakur R. PVFS: A Parallel File System for Linux Clusters. *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000; 317–327.
25. Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop Distributed File System. *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010; 1–10.
26. Daly J. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 2006; **22**(3): 303–312.
27. Xen-credit-scheduler. <http://wiki.xensource.com/xen/wiki/creditscheduler>.
28. Gideon-II Cluster. <http://i.cs.hku.hk/~clwang/Gideon-II>.
29. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, New York: ACM, 2003; 164–177.
30. Vahdat A, Yocum K, Walsh K, Mahadevan P, Kostic D, Chase J, Bechker D. Scalability and accuracy in a large-scale network emulator. *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002; 271–284.
31. Xia H, Dail H, Casanova H, Chien A. The MicroGrid: Using online simulation to predict application performance in diverse Grid network environments. *2nd international workshop on Challenges of Large Applications in Distributed Environments (CLADE'04)*, CA, 2004.
32. Calheiros R, Ranjan R, Beloglazov A, De-Rose C, Buyya R. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience (SPE)*, **41**(1): 23–50, ISSN: 0038-0644, Wiley Press, New York, January 2011.
33. Beloglazov A, and Buyya R: Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. *Software: Practice and Experience (SPE)*, **24**(13): 1397–1420, September 2012. DOI: 10.1002/cpe.1867.
34. GridMix. <https://hadoop.apache.org/docs/r1.0.4/gridmix.pdf>
35. Chen Y, Ganapathi A, Griffith R, Katz R: The Case for Evaluating MapReduce Performance Using Workload Suites. *19th Annual IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'11)*, 2011; 390–399.
36. Cecchet E, Udayabhanu V, Wood T, Shenoy P: BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications. *2nd USENIX Conference on Web Application Development (WebApps'11)*, 2011; 37–48.
37. Calheiros R, Netto M, De-Rose C, Buyya R. EMUSIM: An Integrated Emulation and Simulation Environment for Modeling, Evaluation, and Validation of Performance of Cloud Computing Applications. *Software: Practice and Experience (SPE)*, **43**(5): 595–612, April 2012. DOI: 10.1002/spe.2124.

Government License Section (please add after the reference section): The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.