Xerxes: Distributed Load Generator for Cloud-scale Experimentation

Mukil Kesavan, Ada Gavrilovska, Karsten Schwan
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
mukil,ada,schwan@cc.gatech.edu

***Abstract*—With the growing acceptance of cloud computing as a viable computing paradigm, a number of research and real-life dynamic cloud-scale resource allocation and management systems have been developed over the last few years. An important problem facing system developers is the evaluation of such systems at scale. In this paper we present the design of a distributed load generation framework, *Xerxes*, that can generate appropriate resource load patterns across varying datacenter scales, thereby representing various cloud load scenarios. Toward this end, we first characterize the resource consumption of four distributed cloud applications that represent some of the most widely used classes of applications in the cloud. We then demonstrate how, using Xerxes, these patterns can be directly replayed at scale, potentially even beyond what is easily achievable through application reconfiguration. Furthermore, Xerxes allows for additional parameter manipulation and exploration of a wide range of load scenarios. Finally, we demonstrate the ability to use Xerxes with publicly available datacenter traces which can be replayed across datacenters with different configurations. Our experiments are conducted on a 700-node 2800-core private cloud datacenter, virtualized with the VMware vSphere virtualization stack. The benefits of such a microbenchmark for cloud-scale experimentation include: (i) decoupling load scaling from application logic, (ii) reslience to faults and failures, since applications tend to crash altogether when some components fail, particularly at scales, and (iii) ease of testing and the ability to understand system behavior in a variety of actual or anticipated scenarios.**

I. Introduction

Cloud computing has become a popular computing paradigm that allows end-users to dynamically scale up or down the resources they use to run their applications. Typically, users pay only for resources they actually use, resulting in large cost savings compared to self-hosting applications on dedicated hardware. This growing acceptance of cloud computing has fueled two major trends: (a) a large number and a wide variety of applications now run in the cloud [1] and, (b) researchers and corporations alike are building dynamic resource allocation systems to support these applications at all levels of the system stack [2], [3], [4], [5], [6]. Good application level performance depends on both the nimbleness and accuracy of the underlying resource allocation system, and, the ability of the particular application model to take advantage of scaling out in the cloud without bottlenecks.

An important problem facing system developers is the evaluation of these systems at large scale. The number of servers in current generation datacenters number in their tens of thousands [7], [8]. Not all applications scale to such levels easily without hitting performance bottlenecks due to application logic, framework or data access limitations [9]. Therefore benchmarking is restricted primarily to embarassingly parallel workloads. Even then, it is difficult to generate appropriate inputs to these parallel applications that represent realistic cloud scenarios.

An additional aspect that is implied in the selection of workloads to evaluate cloud systems at scale, is their need to be failure resilient. It is well known that failures are the norm in large scale commodity server datacenters. Clearly, applications that crash outright when one or more components fail are a poor fit for the cloud paradigm, let alone a benchmark. But here again it is important to note that even those applications that do have fault tolerance built into them have varying degrees of tolerance to failures. For example, the use of replicas, say 'n', in data storage systems allows for the tolerance of upto (n -1) replica failures and no more before data becomes inaccessible. At large scales, there is a higher probability of a large number of failures, above and beyond an application's tolerance limit. The net effect of all of these factors is that it is increasingly difficult to properly build and evaluate dynamic resource allocation systems at scale, potentially leading to narrow design assumptions, and optimizations that ultimately lead to poor application performance during deployment.

In this paper we present the design and use of a new distributed load generation framework, *Xerxes*, that decouples the generation of load at scale from any application logic. This allows the evaluation of the scalability of dynamic allocation systems to load patterns of applications that may not inherently scale themselves. In addition, Xerxes offers the ability to generate load patterns at both individual node levels, and collectively across a large number of machines. Various interesting load patterns, including large volume spikes [10] across a large number of machines, can be easily generated. Finally, resource usage traces from real-life deployments can also be adapted and replayed in datacenters of varying sizes.

Xerxes is composed as a collection of four independent load generators – one each for CPU, memory, storage and network resources – deployed on independent physical or virtual machines in the datacenter. The load generators in the individual datacenter nodes are designed to not require any coordination during the course of an experiment to generate an overall load pattern across many machines. This gives Xerxes

the ability to tolerate multiple node failures, thereby improving the robustness of the experimentation process beyond what is achievable with applications which are typically much less tolerant to failures. The accuracy of the generated load pattern at scale decreases linearly with the number of component node failures.

In summary, the technical contributions of this paper are the design and development of Xerxes – a microbenchmark for cloud-scale experimentation, and the demonstration of its utility for different load generation scenarios, based on load patterns extracted from realistic cloud applications or real-life cloud datacenter traces, on a 700-node 2800-core datacenter facility. The benefits of such a microbenchmark include: (i) simplified experimentation at scales due to decoupling of load scaling from application logic, (ii) improved reslience to faults and failures, since applications tend to crash altogether when some components fail, particularly at scales, and (iii) ease of testing and the ability to understand system behavior in a variety of actual or anticipated scenarios.

The remainder of the paper is structured as follows. First, we present four commonly used classes of cloud applications and a characterization of their resource consumption across multiple resource types. The architecture, design and implementation of the Xerxes framework are described in Section III. Section IV shows two example load generation scenarios: one that replays a publicly available datacenter resource usage trace and another that replays the extrapolated load patterns derived from the workload characterization discussed in Section II. Our experiments are conducted on a 700-node 2800-core private cloud datacenter, virtualized with the VMware vSphere virtualization stack.

II. Workload Characterization

A. Workloads

Our workload suite comprises of a set of applications representative of the most popular application classes in current cloud platforms [11], configured to run inside virtual machines (VMs) in our virtualized datacenter:

Data Analytics: The map-reduce framework and its open source implementation, Hadoop [12], have emerged as the de-facto standard for analyzing large datasets in parallel fashion in the cloud. Web search is an important example of this class of applications where data from the plethora of online websites (or local webpages, for private installations) are crawled and indexed on an on going basis in massive datacenters. We replicate this job in our datacenter using the Nutch [13] search engine that is used to crawl an internal mirrored deployment of the popular Wikipedia.org website containing millions of pages of articles. The local deployment allows us to avoid WAN traffic that would skew the workload characterization results.

Data Serving: Key-value stores allow users to store replicated data in a schema-less fashion with relaxed consistency models, and rely on efficient indexes to quickly locate data. As a result, some of the Internet's largest applications running in cloud deployments, such as Facebook Photo Store, Google App Engine, Amazon S3, etc. use these stores to serve vast amounts of data to millions of users. For our characterization we use the Voldemort [14] key-value store, known for its use at LinkedIn, and drive load to it using the Yahoo Cloud Serving Benchmark [15]. Our workload profile consists of 2 million operations (50% reads and 50% writes) with record request popularity following a zipfian distribution. Each record is 64KB in size.

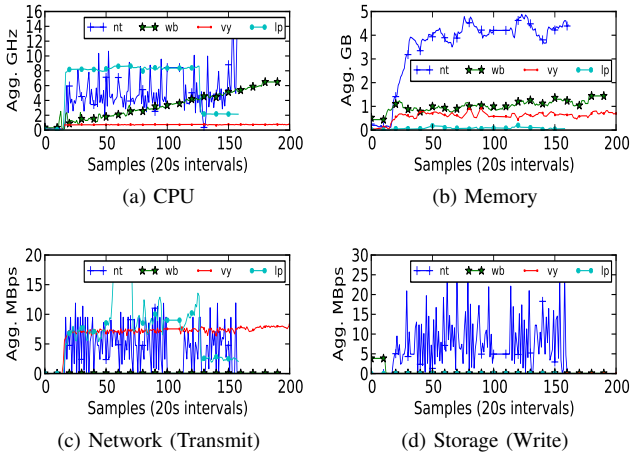
Web Services: N-tier web applications (usually n=3) form the basis for some of the largest online services. In the cloud, a typical web service is composed of a LAMP (Linux, Apache, MySQL, Php) stack, either on a single server or with components split across multiple servers, and a load balancer. The number of instances of any of the three tier nodes (web, application or database) can be scaled out or scaled down in response to varying load [9]. However, the use of a standard SQL database limits the scalability of the data tier mostly due to the overhead of providing ACID properties. For our characterization, we built a 3-tier, airline reservation benchmark that uses Apache Geronimo for the web server, a Tomcat query processing engine for the middle-tier and a HBase backend instead of a SQL-database, capable of achieving good horizontal scalability. We obtained airline fare data from one of our industry partners, Travelport Inc., as well as request traces numbering in their tens of thousands from a real-life deployment. We modified the httperf tool [16] to generate load by replaying these traces for around an hour using three threads with each thread's requests exponentially distributed with a mean inter-arrival time of 1 second.

High Performance Computing: Running large-scale high performance jobs used to be limited to the large national laboratories and big corporations in the past, due to the massive capital investment required to build an HPC cluster. But the availability of large amount of rent-by-the-hour disposable compute nodes provided by cloud computing has made it possible to provide the resources required by these types of applications to a richer user base [17]. Even though the majority of compute resources available in the cloud don't typically offer HPC performance, there are growing trends in both improving cloud infrastructures [18] and adapting HPC frameworks for the cloud [19]. We use programs from the LAPACK [20] package to solve a system of simultaneous linear equations as a representative workload in this space.

B. Resource Usage Characterization

Figure 1 shows the CPU, memory, network and storage usages of the four target applications. Each application is composed of 5 VMs and the utilization values reported are the aggregate usage of all of the VMs of a given application. We have omitted the network receive result, because the VMs only communicate with VMs from the same application (i.e., aggregate transmit = aggregate receive), and the storage reads, since all applications showed very little read activity.

It can be seen that the Nutch application is fairly resource intensive across all resource types, and that the usage pattern is bursty. In terms of dynamically allocating resources, this presents an interesting tradeoff between dedicating resources to handle the spikes (over provisioning) vs. allocating resources



(a) CPU (b) Memory

(c) Network (Transmit) (d) Storage (Write)

Fig. 1: Aggregate Resource Utilizations of Workloads. Key: nt = Nutch, wb = 3-tier web, vy = Voldemort, lp = LAPACK.

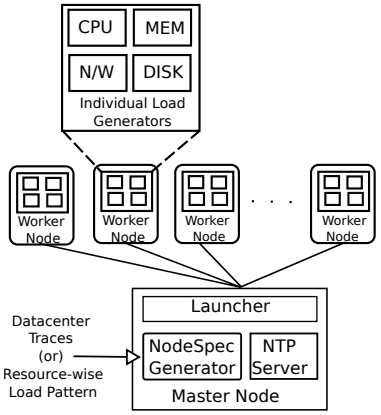


Fig. 2: Xerxes architecture.

to handle either the average load or a higher percentile (under provisioning). The 3-tier web benchmark has a higher CPU and memory requirement than communication or storage resources. This is because the network requests/responses are small in size and so is each HBase record on each VM's local disk. It can also be noted that the CPU and memory requirements steadily increase during the experiment as the application becomes more and more backlogged with requests. The LAPACK high performance computing benchmark has a steady, high CPU requirement and a moderate, somewhat bursty, network requirement. We assume that the spikes in aggregate network traffic correspond to collective operations. Finally, the Voldemort data serving benchmark appears to only require a small but fairly steady amount of memory resource for its operation. This is due to the fact that we configure the benchmark to store data in-memory and also due to the fact that our record size was only 64KB.

We will show in Section IV how this characterization information about the resource requirements of applications, and the different usage patterns, can be extrapolated and replayed across a larger set of machines.

III. Xerxes Design

We next present the design and implementation details of the Xerxes load generation framework. Figure 2 shows the overall framework architecture. Xerxes consists of a single master node and multiple worker nodes - one per server under evaluation. The master node takes load generation specifications, converts them to individual worker node specifications that when executed together produces a global, large scale aggregate load pattern. The individual node specifications are generated by the NodeSpec Generator Module shown in the figure, and their execution is orchestrated by the Launcher Module through the Linux cron facility. The load specifications to the master can be either in the form of (a) real-life datacenter traces specifying resource usages at various timestamps or (b) statiscal distributions, such as normal distribution with a specified mean and deviation values, for example. In addition, these specifications can be at a per-server level or per-logical-job level, that maps to multiple servers, to generate a global resource usage pattern. The base specifications are extrapolated (in the current implementation, only proportionally) when the number of target servers is greater than the number of specification objects in the benchmark input, or, combined via aggregation in the opposite case. Further, it is also possible to add usage volume spikes to the base load specifications by specifying the spike parameters as characterized by Bodik et. al. [10]: *time-spike-start, time-peak-start, time-peak-end, time-spike-end, spike-magnitude-multiplier*.

In order to orchestrate a global resource usage pattern, the master runs an NTP server that the worker nodes need to synchronize with periodically (typically in days on modern machines), so that their individual timeofday values are not hugely divergent. However, once the simulation starts, the worker nodes need no further coordination with the master and use local high precision timers to transition between multiple load phases.

The worker nodes are composed of four individual load generators: one each for CPU, memory, network and storage resources. The worker node gets individual load specifications (or none, as required) for each generator at the start of the simulation from the master, which it then executes in isolation utill completion. The worker load specifications for CPU and memory resources consist of many load phases, one per line, of the form:

$$< period - secs, load - percentage >$$
$$\vdots$$
$$< period - secs, load - percentage >$$

For each phase in the specification above, the CPU load generator generates $load - percentage$ by alternating between performing numerical computation over an integer array's elements and sleeping (using Linux nanosleep) at microsecond granularity, many times over until $period - secs$ seconds have elapsed. For example, using a 100 microsecond period and a desired load of 50% would mean that the generator simply computes for 50 microseconds and sleeps for the remaining 50. The CPU generator periodically calibrates itself to determine the number of computations required for a span of 1 microsecond in order to operate in a virtualized environment where the amount of available CPU resource varies with levels of consolidation. The generator can also be configured to do a

fixed amount of work instead. However, from our practical experience on virtualized systems this mode of operation results in decreased accuracy in global load pattern generation.

The memory load generator works similarly to the cpu generator in terms of transition between phases, but interprets the $load - percentage$ as a fraction of a large pre-configured memory size (could potentially be the total available worker node memory size) specified in megabytes. It allocates an integer array buffer of size corresponding to the fraction specified for a particular phase, and performs either random access or linear access of the elements of the array as required for $period - secs$ seconds.

In our current Xerxes prototype the master node components, except the NTP server, are implemented in Python. We use the NTP server available through the Ubuntu software repository. The CPU and memory load generators are written in C for Linux kernel versions 2.6.19 and above where the high-resolution timer API is available. We are currently in the process of developing the network and storage load generators. The network generator is written from scratch in C using the Linux sockets API allowing control over message sizes, data rates and multiple communication end points. As part of future work, we would like to explore different communication patterns representative of ones seen in real world applications (e.g., broadcast, collective communication, etc.). As for the storage load generator we plan to customize the open source fio [21] I/O tool to fit the overall Xerxes framework.

The following section presents several examples of how the Xerxes framework can be used to create a variety of CPU and memory resource usage patterns in the datacenter.

IV. Load Generation Scenarios

We run Xerxes on a 700 node private cloud constructed on a datacenter on campus using the VMware vSphere [22] virtualization platform. Each server has 2 dual core AMD Opteron 270 processors, a total memory of 4GB and two NICs capable of 1Gbps and 5Gbps respectively. The hosts are all connected to each other and a central storage array of 4.2TB total capacity via a Force 10 E1200 switch over a flat IP space. The worker nodes are Ubuntu Linux 9.10 virtual machines, each configured with 4 vcpus and 1GB of memory, stored on the central storage array. The master node is run on a separate physical server running Ubuntu Linux 9.10 as well. Our monitoring infrastructure is built using the VMware vSphere Java SDK [23] that allows us to fetch resource utilizations samples per-VM once every 20 seconds at small scales and once every 5 minutes at larger scales.

A. Replaying Datacenter Traces

Recently, Google Inc. released a large scale, anonymized production workload trace from one of its clusters [24], [25], containing data worth over 6 hours with samples taken once every five minutes. Their workload consists of 4 large jobs that each contain a multitude of sub tasks that map to their cluster machines in an unknown way[1]. Each row in the trace

[1]We use the trace version 1 where this information was unavailable. A subsequent update from Google provides more detailed information.

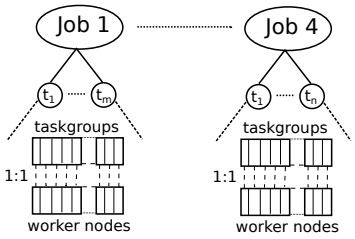
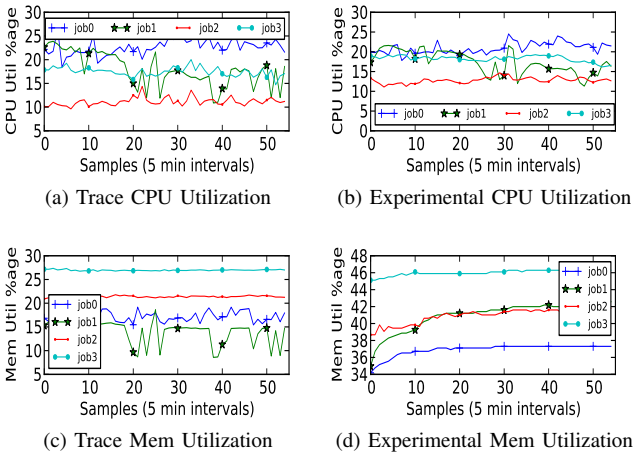


Fig. 3: Mapping Google Trace to Xerxes Model.



(a) Trace CPU Utilization (b) Experimental CPU Utilization

(c) Trace Mem Utilization (d) Experimental Mem Utilization

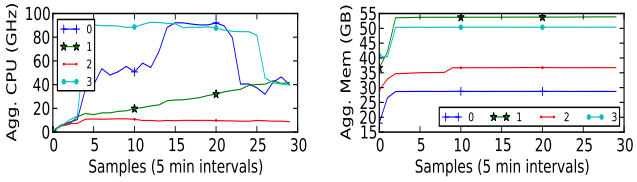
Fig. 4: Job-wise Resource Utilizations for Google Traces.

presents the resource usage of a single task, belonging to one of the four jobs, at a given timestamp.

The CPU usage is expressed as normalized value of the average number of cores used by the task and the memory usage is expressed as the normalized value of the average memory used by the task over the last 5 minute interval. This provides us sufficient information to replay the resource usage pattern of the four major jobs in the trace on 1600 VMs running on 512 of the 700 servers in our private cloud. For the sake of simplicity our global load specification evenly partitions the VMs into four sets of 400 VMs where each set is to replay the CPU and memory usage of a unique job.

The number of unique tasks for each job at each timestamp varies over time, with numbers in the tens of thousands, on average. The NodeSpec Generator at the master evenly partitions tasks of a job, at a timestamp, into 400 taskgroups, one taskgroup each for each worker VM as illustrated in Figure 3. We simply re-normalize the resource usage of each task group to a percentage value (assuming a base maximum value) and generate the entire single worker load specification as a series of utilizations to be generated every 5 minutes working up to around 5 hours. Thus each worker node has a different load pattern corresponding to its taskgroup but they together produce the overall global job patterns required.

Figures 4a and 4b show the CPU utilization per job computed from the trace and measured from an actual benchmark run, respectively. Given the scale of the experiment, it can be seen that the overall job load patterns are reproduced fairly accurately in the experimental run. Note that we did observe worker node failures during the course of the experiment and also that there are limits to the granularity of our monitoring setup measuring the utilization in the infrastructure.



(a) CPU Utilization (b) Memory Utilization

Fig. 5: Resource Utilizations for Each Application Profile. Key: 0 = Nutch, 1 = 3-tier Web, 2 = Voldemort, 3 = LAPACK.

Similarly, Figures 4c and 4d show the memory utilization per job computed from the trace and measured from an actual benchmark run, respectively. Our datacenter monitoring module measures the memory utilization of entire VMs: this includes the load generator's memory consumption and the VM guest OS utilization resulting in the reported experimental samples being higher than the ones computed from the trace. From our practical observations running the memory load generator, we observed that it is hard to generate accurate memory loads for VMs that have a small amount of configured memory due to the higher fraction of the base memory usage by the guest OS and any other services compared to VMs with large amount of memory (e.g., base = 400MB of 1GB vs. base = 400MB of 10GB).

B. Replaying Patterns Extracted from Workloads

In this scenario, our goal is to take each application's CPU and memory usage profile from Section II, and, extrapolate it such that it can run on 400VMs hosted on 128 machines. Recollect that each application profiled has resource usage information of each of its 5 VMs at a 20 second granularity. In our load specification, we simply map the resource usage of each single VM belonging to a given application to 20 VMs in our target simulation to come to a total of 400 VMs. Each application profile will be simulated by 100VMs now. Note that Xerxes has and can be extended to support more complex mappings as well. In addition, to demonstrate the ability of the Xerxes framework to simulate large spikes in the cloud, we added a volume spike to the CPU profile of the Nutch application in the specification. The spike lasts from 60 minutes into the simulation until 120 minutes, and approximately triples the overall workload volume.

Figure 5 shows the results of this scenario. It can be seen that even with the coarse monitoring granularity, the overall CPU usage of each application's VMs is similar to that characterized before. The Nutch profile shows the volume spike, across 100 VMs, as added by Xerxes. However, as with the previous scenario the memory monitoring data is less accurate for the same reasons as above. We are working on ways to collect and store at scale, the set of active pages being touched for each VM directly from the hypervisor.

V. Conclusions

In this paper we argued for the decoupling of scalable load generation from application logic in order to aid researchers/developers test their cloud systems at large scale, which ultimately prevents narrow system design assumptions that ignores the issues seen at scale. We demonstrated the use of a distributed load generation framework, on a 700 node private cloud virtualized with the VMware vSphere virtualization stack, that can: (i) replay real-life datacenter traces, (ii) extrapolate and replay workload characterization data and, (iii) simulate resource usage volume spikes across a large number of machines. In the future we plan on completing the network and storage load generation components, and, add new interesting features to extrapolate resource data. We also plan on using Xerxes to generate diverse load scenarios to understand the effectiveness of different load and power management methods at scale.

Acknowledgments

We would like to thank Hrishikesh Amur, a PhD student in our research group for giving us the initial version of the CPU and memory load generators, and our former student, Bala Chandar, for developing the airline reservation application used in this work.

References

[1] "Customer apps - amazon web services," http://aws.amazon.com/customerapps/.
[2] B. Hindman *et al.*, "Mesos: a platform for fine-grained resource sharing in the data center," in *NSDI '11*.
[3] Z. Shen *et al.*, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *SoCC '11*.
[4] "Rightscale: Cloud computing management platform," http://www.rightscale.com/.
[5] T. Wood *et al.*, "Black-box and gray-box strategies for virtual machine migration," in *NSDI '07*.
[6] "Elastic block store," http://aws.amazon.com/ebs/.
[7] L. A. Barroso *et al.*, "Web search for a planet: The google cluster architecture," *IEEE Micro*, Mar. 2003.
[8] A. Greenberg *et al.*, "The cost of a cloud: research problems in data center networks," *SIGCOMM Comput. Commun. Rev.*, Jan. 2009.
[9] S. Malkowski *et al.*, "Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks," in *IISWC '09*.
[10] P. Bodik *et al.*, "Characterizing, modeling, and generating workload spikes for stateful services," in *SoCC '10*.
[11] M. Ferdman *et al.*, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS '12*.
[12] "Apache hadoop," http://hadoop.apache.org/.
[13] "Apache nutch," http://nutch.apache.org/.
[14] "Project voldemort," http://project-voldemort.com/.
[15] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with ycsb," in *SoCC '10*.
[16] "httperf," http://www.hpl.hp.com/research/linux/httperf/.
[17] K. R. Jackson *et al.*, "Performance analysis of high performance computing applications on the amazon web services cloud," in *CLOUDCOM '10*.
[18] "High performance computing (hpc) on aws," http://aws.amazon.com/hpc-applications/.
[19] "icode: A framework for enabling supercomputing resources as hpc cloud," http://www.nsfcac.rutgers.edu/icode/.
[20] "Lapack - linear algebra package," http://www.netlib.org/lapack/.
[21] "fio," http://freecode.com/projects/fio/.
[22] "Vmware vsphere," http://www.vmware.com/products/vsphere/.
[23] "Vmware vi (vsphere) java api," http://vijava.sourceforge.net/.
[24] "googleclusterdata - traces of google tasks running in a production cluster," http://code.google.com/p/googleclusterdata/.
[25] Y. Chen *et al.*, "Analysis and lessons from a publicly available google cluster trace," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95, Jun 2010.