Web-scale Job Scheduling

Walfredo Cirne[1] and Eitan Frachtenberg[2]

[1] Google
walfredo@google.com
[2] Facebook
etc@fb.com

Abstract. Web datacenters and clusters can be larger than the world's largest supercomputers, and run workloads that are at least as heterogeneous and complex as their high-performance computing counterparts. And yet little is known about the unique job scheduling challenges of these environments. This article aims to ameliorate this situation. It discusses the challenges of running web infrastructure and describes several techniques to address them. It also presents some of the problems that remain open in the field.

1 Introduction

Datacenters have fundamentally changed the way we compute today. Almost anything we do with a computer nowadays—networking and communicating; modeling and scientific computation; data analysis; searching; creating and consuming media; shopping; gaming—increasingly relies on remote servers for its execution. The computation and storage tasks of these applications have largely shifted from personal computers to datacenters of service providers, such as Amazon, Facebook, Google, Microsoft, and many others. These providers can thus offer higher-quality and larger-scale services, such as the ability to search virtually the entire Internet in a fraction of a second.

This transition to very large scale datacenters presents an exciting new research frontier in many areas of computer science, including parallel and distributed computing. The scales of compute, memory, and I/O resources involved can be enormous, far larger than those of typical high-performance computing (HPC) centers [12]. In particular, running these services efficiently and economically is crucial to their viability. Resource management is therefore a first-class problem that receives ample attention in the industry. Unfortunately, little is understood about these unique problems outside of the companies that operate these large-scale datacenters. That is the gap we hope to address here, by discussing in detail some of the key differentiators of job scheduling at Web scale.

Web datacenters differ from traditional HPC installations in two major areas: workload and operational constraints. Few of the HPC workloads provide interactive services to human users. This is in stark contrast with datacenters that serve Web companies, where user-facing services are their *raison d'etre*. Moreover, a sizable part of the HPC workload consists of tightly coupled parallel jobs

[15] that make progress in unison (e.g., data-parallel MPI 1.0 applications). Web workloads, on the other hand, are less tightly coupled, as different tasks of a job typically serve different requests.

This difference in the amount of coupling of the applications affects the guarantees that the resource manager must deliver in each environment. Web applications typically do not require all nodes to be available simultaneously as a data-parallel MPI job does. In this sense, they are much closer to HPC high-throughput applications [7] and can better tolerate node unavailability. Unlike high-throughput applications, however, they do not finish when the input is processed. They run forever as their input is provided on-line by users.

Of course, not all workloads in Web clusters are user-facing. There are also batch jobs to enable the user-facing applications. Google web search, for example, can be thought of as three different jobs. The first crawls the web, the second indexes it, and the third serves the query (i.e., searches for it in the index). These jobs need different levels of Quality-of-Service (QoS) guarantees to run reliably, with the user-facing query-serving job having more stringent requirements. Identifying the QoS levels that different applications demand is key in the process of managing Web datacenters, as providing different Service Level Agreements (SLAs) allows for better utilization and gives flexibility to cluster management.

Moreover, the fact that a key fraction of the Web workload is user-facing has even greater significance than the QoS level: it also changes the operational requirements for the datacenter. As HPC jobs essentially run in batch mode, it is feasible for datacenter administrators to shut it down as needed, either entirely or partially. Jobs get delayed, but since maintenance is infrequent and scheduled, this is acceptable. Web companies, in contrast, never intentionally take down their services entirely. Consequently, a datacenter-wide maintenance of either hardware or software is out of question.

This paper discusses these differences in detail in Sections 2 (workload and SLAs) and 3 (operational requirements). Section 4 provides a case study of how these issues affect cluster management. It describes in detail a technique called *backup tasks* that is used by Google to assure a higher QoS for its user-facing application, running in a cluster shared with batch load. Section 5 discusses other issues in Web cluster management that, while not the focus of this paper, are relevant and important. Section 6 concludes the paper.

2 A Spectrum of SLAs

Traditional supercomputers, grids, and HPC centers can have a relatively rigid set of SLAs. Consider, for example, job reservation systems such as queue-based space-sharing schedulers [10]. Some of these schedulers can have varying and complex degrees of job priorities (e.g., Maui [14]). But the basic SLA remains the same: the system offers a firm commitment that all resource needed by a job will be available during its execution, usually ignoring the inevitable possibility of system failures and downtime. Ignoring the possibility of failure may be reasonable for some smaller clusters of commodity servers, or for some high-availability, high-end servers in supercomputers. But usually, more than by a hardware promise, it is driven by a software requirement: Many HPC jobs rely on the MPI-1 protocol for their communication (and other similar protocols) which assume that all tasks in a job remain available through the completion of a job. A failure typically means that the entire job has to restart (preferably from a checkpoint).

Schedulers for Web Clusters cannot make this assumption. The sheer scale of datacenter clusters, coupled with commoditized or mass-produced hardware, make failures a statistical certainty. But the applications are also designed for weaker guarantees. Instead of the tightly synchronous communication of parallel jobs, most Web jobs are distributed in nature, designed for fault-tolerance, and sometimes even completely stateless, so that the replacement of one faulty node with an identical node is virtually transparent to software. This leads to a primary distinction between traditional SLAs and Web SLAs: Accounting for non-deterministic availability of resources means that the resource manager makes probabilistic guarantees, as opposed to the absolute (albeit unrealistic) guarantees of supercomputers. An example of such a probabilistic guarantee can be: "this job will have a 99.99% probability of always receiving 1,000 nodes within a minute of requesting them."

This distinction, combined with the modal characteristics of Web workloads [12, 21], can also mean that Web jobs can be much more varied when it comes to QoS requirements, representing an entire spectrum of responsiveness (Fig. 1). The following breakdown enumerates some common types of SLA requirements for distributed Web jobs. Note that these are examples only, and not all workloads consist of all categories, or with the same prioritization.

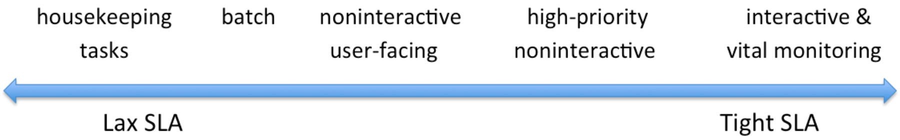
housekeeping tasks
batch
noninteractive user-facing
high-priority noninteractive
interactive & vital monitoring
Lax SLA
Tight SLA

Fig. 1. *Spectrum of SLA requirements at Web clusters.*

- **Monitoring jobs**. There are a small number of jobs whose sole purpose is to monitor the health of the cluster and the jobs running on it. These jobs demand very good SLAs, as their unavailability leaves systems administrators “flying in the dark”.
- **User-facing services**. These so-called “front-end” services are what the user interacts with and represent the organization’s highest priority, and therefore highest required QoS. Examples of front-ends are the search page for Google, the news feed for Facebook, and the store front for Amazon. The speed of interaction directly affects both user experience and revenue; so a high emphasis is placed on avoiding delays caused by lack of resources.

Guarantees for this QoS level may include a certain minimum number of dedicated machines, or a priority to override any other services when sharing resources.

- **High priority noninteractive services**. An example of such a service may be to sum up the number of clicks on an advertisement. Although not user-facing, this is a short, high-priority job, and may require guarantees such as the sum value in the database must be accurate to within the past few minutes of data. Managing these jobs, for example, may be done by servicing them on the same user-facing nodes that create these tasks, but deferring their processing past the point that the page has been presented to the user.
- **Noninteractive user facing services**. These are services that affect the user experience, but are either too slow to designate as interactive (in the order of many seconds to minutes) or not important enough to require immediate, overriding resources. One example of such a service builds the list of people recommendations on a social network ("People You May Know"). Whenever a user decides to add or remove a connection to their network, their list of recommended connections may require updates, which may take several seconds to compute. Although the list must be updated by the next time the user logs in, there is no need to require the user to wait for this update. Tasks for such jobs can be either scheduled on the same user-facing nodes at a lower local priority (“nice” value) [27], or on a different tier of dedicated servers, with separate queues for different SLAs.
- **Batch jobs**. Long jobs that do not directly affect the user experience, but are important for the business. These include MapReduce and Hadoop jobs, index building, various analytics services, research and experimentation, and business intelligence. Like batch jobs on traditional supercomputers, these jobs can vary in SLA from strong guarantees to no guarantees whatsoever. A special class of jobs may have recurring computation with hard completion deadline (but otherwise no set priority). An analogue for such jobs in traditional HPC is the daily weather forecasts at ECMWF [13].
- **Housekeeping tasks**. These can be various low-impact logging services, file-system cleanup, diagnostic metric collection, cron jobs, etc. These may be run with different variations on all active nodes in the system, at the lowest possible priority or at scheduled local downtimes. However, we must ensure they do not starve.

While the enumeration above does imply different levels of priority or SLA, the different categories overlap. For example, if disk is filling up very quickly, file-system cleanup will rise in priority and potentially take precedence over batch jobs or even user facing activities. In general, a given job is evaluated by its importance to the business and the minimal guarantees it needs, and an SLA-level is chosen.

Compute node sharing

Resource utilization is a very important economical and environmental concern, especially for large Web companies, due to the unprecedented scale of their clusters [12]. Maximizing utilization, in job scheduling, however, can be at odds with responsiveness and fairness [9, 11, 25]. In particular, Web schedulers have the choice of collocating jobs from different SLA tiers on the same compute nodes, or of allocating dedicated nodes for each desired tier.

Collocating different services on a single machine, even within virtual machines, can create hard-to-predict performance interactions that degrade QoS [19]. It can also complicate diagnosing performance and reliability issues, since there are more interacting processes and nondeterministic workload mixes. It cannot completely solve the performance problems of internal fragmentation either [12]. However, it allows for reclaiming unused resources that are allocated to guarantee we can deal with peak load, but that go unused most of the time.

Note that sharing compute nodes derives most of its appeal from the fact that different SLA tiers have varying requiremetns on resource availability. If a machine only runs high-SLA user-facing services, one cannot reclaim unused resources because they will only be needed at peak load. The existance of load that can be preempted (when user-facing services go through peak load) provides a way to use resources that would otherwise be wasted (when user-facing services are not at peak load).

Accounting

Another economic consideration for Web scheduling is accurate accounting and charging for resources while providing the agreed-upon SLAs [29]. This may not be an issue when all the services are internal to the same company with a single economic interest (such as Facebook and Google), but it is a critical issue for shared server farms, such as Amazon's EC2 cloud [1]. Part of the issue is the need to guarantee SLAs in the cloud in the presence of resource sharing and different VMs, as discussed in the previous paragraph. Trust is another aspect of the problem: How can the consumer verify if the resources delivered indeed have the correct SLA, if it is running within the provider datacenter (and thus in principle vulnerable to the provider's manipulation).

3 Operational Requirements

Cluster operation and job scheduling are two inseparable concepts for Web services. Whereas supercomputers can have scheduled full- or partial-downtime for maintenance (or even unscheduled interruptions), and then merely resume execution of the job queue from the last checkpoint, Web services strive to remain fully online at all times. Site-wide maintenance is not an option. And yet failing hardware must be repaired and software upgraded. To minimize service disruption, this process must be carefully planned as part of overall job scheduling.

One approach for scheduled upgrades is to handle the upgrade task the same way as a computational task. The downtime is estimated and treated as a job requirement, to be scheduled concurrently with existing jobs while preserving their SLAs. To protect against software failures, many maintenance tasks require a method of gradually changing the deployed service while maintaining the SLAs of currently running jobs. For example, Facebook pushes a new version of its main front-end server—a 1.5GB binary—to many thousands of servers every day [23]. The push is carried out in stages, starting from a few servers and, if no problems are detected, exponentially grows to all servers. In fact, the first stage deployment to internal servers is accessible only to employees. This is used for a final round of testing by the engineers who have contributed code to this push. The second stage of the push is to a few thousand machines, which serve a small fraction of the real users. If the new code does not cause any problems either in resource usage or in functionality, it is pushed to the last stage, which is full deployment on all the servers. Note that Facebook's service as a whole is not taken down while the code is updated, but each server in turn switches to the new version. The push is staggered, using a small amount of excess capacity to allow for a small number of machines to be down for the duration of the binary replacement. The files are propagated to all the servers using BitTorrent, configured to minimize global traffic and network interruptions by exploiting cluster and rack affinity. The time needed to propagate to all the servers is about 20 minutes.

Some applications accumulate a lot of state (such as large in-memory key-value stores) and require further coordination with the resource manager for graceful upgrades. For example, the application could be notified of an impending software upgrade, save all of its state and meta-data to shared memory (shmem), shut down, and reconnect with the data after the upgrade, assuming the version is backwards compatible.

Another operational requirement that affects scheduling is the need to add and remove server nodes dynamically without interrupting running services [12]. The implication here is that both applications and the scheduler that controls them can dynamically adjust to a varying number of servers, possibly of different hardware specifications. This heterogeneity poses an extra challenge to resource planning and management, compared to the typical homogeneous HPC cluster.

Yet another scheduling consideration that can affect both SLAs and resource utilization, especially in the cloud, is security isolation [24]. Often, jobs that are security sensitive, e.g., data with Personally Identifiable Information (PII) or financial information (credit card data, etc.) must be segregated from other jobs. Like SLAs, this is another dimension that is easier to manage with static resource allocation, as opposed to dynamically allocating entire machines to different jobs (due to issues of vulnerability explotation by previous jobs).

As mentioned in Sec. 2, one of the main philosophical differences between Web clusters and HPC clusters and supercomputers, is that Web schedulers always assume and prepare for the possibility of node failures [12]. In practice, this means that schedulers must keep track of job progress against available nodes, and automatically compensate for resources dropping off. One famous example of a fault-tolerant resource management system for Web services is Map-Reduce [8] and its open-source counterpart, Hadoop [28]. With these systems, tasks on failed machines are automatically respawned, and users have a strong expectation that their job will complete even in the face of failures, without their intervention.

An overriding principle in the operation of Web clusters and job scheduling is that it is almost always better not to start a new service than to bring down an existing one. In particular, if one wants to update a service and the new version is going to take a greater resource footprint than the current one, one must be able to answer in advance the question: is the update going to succeed? If not, admission control should reject the update.

The case study presented in the next section illustrates these points by describing one way in which resources can be scheduled dynamically for new services without interrupting existing SLAs.

4 Case Study: Guarantees for User-Facing Jobs

In the mix of jobs that run in the cloud, user-facing applications require the highest level of service. They require guarantees both at the machine level as well as the cluster level. Machine-level guarantees ensure how fast an application has access to resources in a machine it is already running, whereas cluster-level guarantees establish how fast an application has access to a new machine in case of need (e.g. raising load, or an existing machine failed).

Machine-level guarantees are needed because low latency adds business value. These applications must respond to user requests very quickly, typically within a fraction of a second. Therefore, they must have access to the CPU in very short order when they become ready to run. Likewise, we need to ensure appropriate performance isolation from other applications [19]. The easiest way to provide these guarantees is dedicating machines to user-facing jobs. Alas, this comes at the expense of lower utilization. There is therefore an incentive for finding ways to share machines between user-facing and less-stringent load while keeping strong QoS for user-facing jobs.

But machine-level guarantees are not enough per se. We must also ensure that these applications have enough resources available in spite of hardware failures and maintenance disruptions. As with machine-level guarantees, the easiest way to accomplish this is to dedicate a set of machines to the application at hand, over-provisioning to accommodate failures and disruptions. Keep in mind, however, that failures can be correlated (switch and power element nodes bring down a set of machines) and even with the dedicated assignment of machines, this is not a trivial problem.

Cluster-level Guarantees at Google

Google cluster management does share machines between user-facing and background jobs to promote utilization. A jobs is composes by a set of tasks. The cluster-level guarantee we provide is that we will (within a probabilistic threshold) restart a task if its machine (or other hardware element) fails. In order to do so, we need to ensure we have enough spare resources to withstand hardware failures. This is accomplished by doing admission control based on *backup tasks*. The basic idea is that we only admit a user-facing job that we are confident we have enough spare resources to keep it running in face of failures and disruptions. It is better not to start a new service, as it by definition has no users to affect, than to disrupt an existing service.

To provide the guarantee that we have enough resources we need to allocate spare (backup) resources that cannot be allocated to another service job. Explicit allocation of the spare resources is needed to prevent incoming jobs (which may schedule fine) compromising the backup resources that are providing guarantees to existing jobs. Background jobs can use the backup resources, but they will be preempted if the user-facing jobs need them.

Therefore, the issue becomes how to allocate (the minimum amount of) resources to ensure we can withstand a given failure/disruption rate. We do so by scheduling backup tasks. A backup task is just a placeholder. It does not execute any code. It just reserves resources to ensure we have a place to restart real tasks in case of failure. A key feature of a backup task is that it reserves enough resources for any real task it is protecting. For example, if a backup task **B** is protecting real tasks **X** (requiring RAM = 1GB, CPU = 1.0) and **Y** (requiring RAM = 100MB, CPU = 2.0), **B** must at least reserve RAM = 1GB, CPU = 2.0. Likewise, any constraint required by a protected real task (e.g., the machine needs a public IP) must also apply to its backup tasks.

We define a *task bag* as the basic abstraction to reason about which real tasks are being protected by which backup tasks. A task bag is a set of tasks with the following properties:

1. All backup tasks in the bag are identical.
2. All real tasks in the bag can be protected by any backup task in the bag (i.e., their requirements are no bigger than the backup tasks').

Note that task bag is an internal concept of the cluster management system, being completely invisible to the user. The user aggregates tasks into jobs to make it convenient to reason about and manage the tasks. The cluster management system aggregates tasks into bags to reason about failure/disruption protection. Fig. 2 illustrates the concept with an example.

As the backup tasks in a bag have enough resources to replace any real task in the bag, as long as we do not lose more tasks than the number of backups, we will always have resources for the real tasks in the bag. That's basically the observation we rely on when doing admission control. A job is admitted if we can place its tasks in existing bags (or create new ones) and the probability of any bag losing more than its number of backups is below a threshold T. That is, a bag containing r real tasks and b backup tasks is said to be T*-safe* if the probability os losing more than b tasks (real or backup) is below T.

When is a Task Bag safe?

We determine if a bag is T-safe by computing the discrete probability distribution function of task failure in the bag. Such a computation takes into account

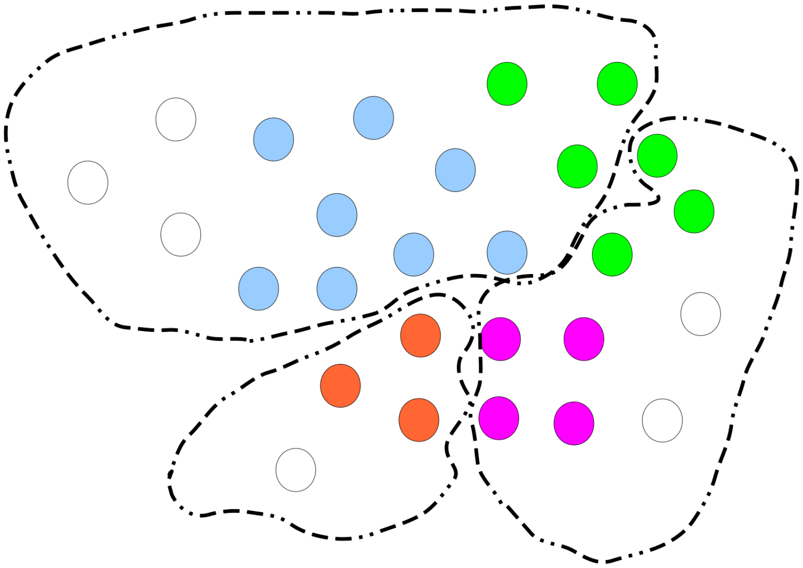

Fig. 2. *Four jobs (in different colors) protected in three bags (backup tasks are in white). Note that bags are formed for the convenience of the cluster management system, which can divide tasks in any way it deems useful.*

correlated failures, as long as the failure domains nest into a tree. If failures domains overlap, we approximate the actual failure graph by a tree. To the best of our knowledge, computing the pdf of task failure with overlapping failure domains is an open problem.

For the sake of explanation, we are going to assume that failures happen to machines and racks (which group a set of machines under a network switch). The generalization to more failure domains is straightforward, as we shall see. The input for our computation is the failure probabilities of each component that makes up the cluster, i.e., machine and racks in this explanation:

- $P(\mathbf{r})$ = the probability that rack **r** fails
- $P(\mathbf{m}|\neg\mathbf{r})$ = the probability the machine **m** fails but the rack **r** that holds **m** has not

Let's start with the case that all tasks in the bag are in the same rack **r** and that there is at most one task per machine. Say there are R tasks in the bag (i.e., the bag uses R distinct machines in rack **r**). We seek the probability distribution function $P_{\mathbf{r}}(f = x)$, where f is the number of tasks to fail, given $P(\mathbf{r})$, $P(\mathbf{m}|\neg\mathbf{r})$, and R. Fortunately, this computation is straightforward:

- $P_{\mathbf{r}}(f > R) = 0$, as we cannot lose more tasks than we have.
- $P_{\mathbf{r}}(f = R) = P(\mathbf{r}) + P(\neg\mathbf{r}) \times PI_{\mathbf{r}}(f = R)$, i.e. we either lose the entire rack or individually lose all machines we are using.
- $P_{\mathbf{r}}(f = x < R) = P(\neg\mathbf{r}) \times PI_{\mathbf{r}}(f = x)$, as if we lose less than R machines we could not have lost the rack.
- $PI_{\mathbf{r}}(f = x) = Binomial(x, R, P(\mathbf{m}|\neg\mathbf{r}))$, where $PI_{\mathbf{r}}(f = x)$ is the probability we independently lose x machines.

Note that the computation described above gives us a discrete pdf. Therefore, to cope with multiple racks, we compute $P_{\mathbf{r}}(f = x)$ for each rack **r** the bag uses and use convolution to add them together to obtain $P(f = x)$ over the entire cluster. This is possible because we assume racks fail independently. Likewise, as long as greater failure domains nest nicely (e.g., racks are grouped in power nodes), we can separately compute their failure pdf and add them together.

Dealing with more than one task per machine uses the same approach. We devide each rack into in the set of machines with i = 1, 2, 3, ... tasks. We then compute $PI_{\mathbf{r}i}(f = x)$ for each "sub-rack" $\mathbf{r}i$ to obtain the machine failure pdf. As each machine runs i tasks, we multiply it by i to obtain the task failure pdf. Adding these pdfs together produces the task failure pdf for the rack. Fig. 3 exemplifies the process by showing a rack in which 2 machines run 1 task of the bag ($i = 1$), 5 machines run 2 tasks each ($i = 2$), and 1 machine runs 3 tasks (i = 3). Note that the number of task failures is multiplied by i as there are i per machine (e.g. $PI_{\mathbf{r3}}(f = 0)$ and $PI_{\mathbf{r3}}(f = 3)$ for $i = 3$).

As an illustration of the failure pdf, Fig. 4 shows the task failure pdf for a real bag containing 1986 tasks. One may be curious about the saw-shaped pattern in the pdf. This was caused by 664 machines that had an even number of tasks (2 or 4 tasks per machine) whereas only 80 machines had an odd number of tasks

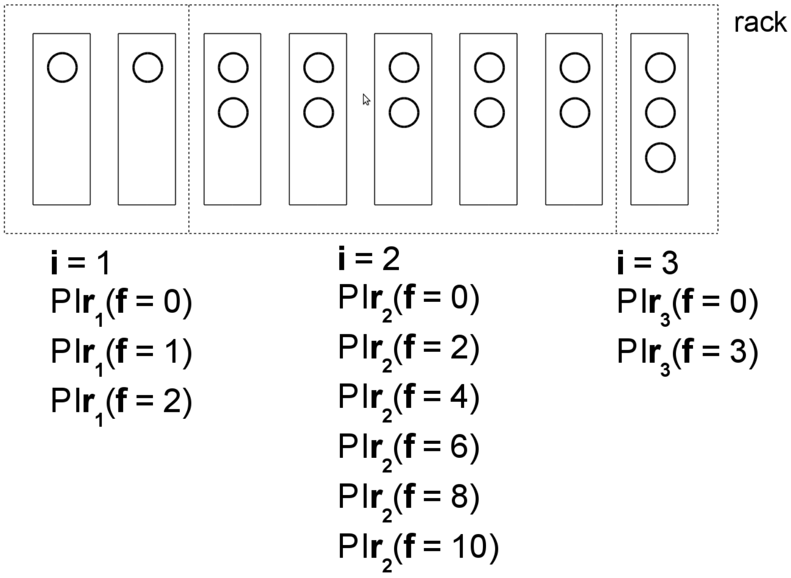

Fig. 3. *A rack in which 2 machines run 1 task of the bag, 5 machines run 2 tasks each, and 1 machine runs 3 tasks.*

(1 or 3 tasks per machine). This highlights another feature of the failure pdf: it is strongly influenced by where the tasks are placed.

The bagging process

Determining if a bag is T-safe runs in polynomial time. However, deciding how to group tasks into bags in an optimal way is NP-hard (optimality is defined as the grouping that minimizes the resources reserved by backup tasks.) In fact, this problem is akin the set cover problem [16]. We use an eager heuristic during the "bagging" process, used in admission control:

```
submission(job J):
   break J into a set of identical sub-jobs, each with identical tasks
   for each sub-job Ji:
      bag(Ji)
   if all bags are T-safe:
      commit the bagging of J
      accept J
   else:
      undo the bagging of J
      deny J
```

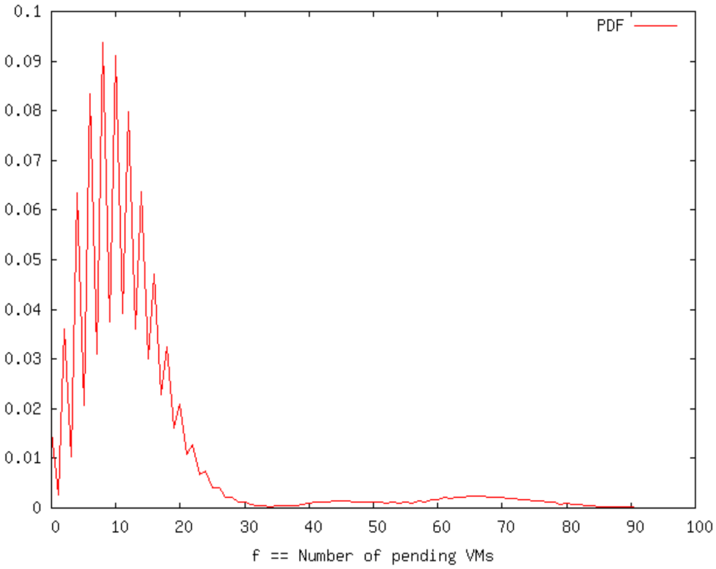

Fig. 4. *The task failure pdf of a real 1986-task bag.*

```
bag(Ji):
   for each existing bag B:
      if it is possible to find enough backups for B + Ji:
         original_cost = BackupCost(B)
         add Ji to B (possibly creating new backup tasks and/or
            increasing the resources reserved by the backup tasks)
         new_cost = BackupCost(B)
         cost[B] = new_cost - original_cost
         restore B to its original form
   if it is possible to find enough backups for Ji to be in a new bag:
      cost[ALONE] = Kpenalty * BackupCost(Ji)
   if at least one solution was found:
      select the choice that gives the smallest cost
```

BackupCost() is a multidimensional quantity over all resources we allocate (CPU, memory, disk capacity, etc). As different dimensions have different scales, we compare two costs is done by multiplying together all dimensions. That is, $c_1 < c_2$ iff c_1[cpu] × c_1[memory] × ... $<$ c_2[cpu] × c_2[memory] × ...

$K_{penalty}$ is a penalty factor for us to create a new bag. It was introduced because having fewer, larger bags in principle reduces the resource consumption of backup tasks. Larger bags tend to achieve T-safety with a smaller fraction of backups because we cannot allocate a fractional backup task. For a simplistic example, assume that racks do not fail, machines fail with 1% probability, and there is always one task per machine. In this scenario, both a 1-task and an 100-task bag require 1 backup.

The bagging algorithm considers the topology of the failure domains when placing a backup task. In order to get the best failure pdf with a new backup task, we prefer to select a machine in a rack that is not yet used by the bag. If there is no such rack, we prefer a machine that is not yet used by the bag. Finally, when there is no such machine, we collocate the new backup task with other tasks already in the bag.

5 Other Issues

This exposition is by no means exhaustive of the challenges of scheduling at Web scale. Besides SLA diversity and strict operational requirements, there are many other dimensions in which Web-scale Job Scheduling poses unique challenges.

One of the most important open problems in resource management in general, but especially at Web scale, is how to increase power efficiency through job scheduling [12]. The load at large Web services varies considerably with diurnal and other patterns [4], but capacity is typically provisioned for peak load. Hosts that remain idle during non-peak hours can spend disproportionate amounts of energy [5]. What is the best use for these idle or nearly-idle machines? Should they be put to sleep? This may not be possible if they are also storage servers. Should they run low-priority batch jobs? But then what about the performance interference on the high-priority load? Should they not be acquired in the first place? But then how do we deal with peak load? Each possible solution has its own business tradeoffs. There is significant research work in progress in industry and academia to address this problem, but many challenges still remain [3, 18, 20, 26].

Another interesting topic is the mixing of computational and storage resources in Web clusters. Supercomputers typically have their own storage infrastructure, often in the form of dedicated storage nodes with specialized hardware. The picture is less clear-cut for Web clusters that tend to use commodity hardware and often consume extremely large amounts of storage, in the order of many petabytes. For some services, such as Facebook's front-end Web servers, compute nodes are basically stateless, and all the persistent storage they require is served by specialized database and caching nodes. In other cases, such as Hadoop, the storage is collocated with the computation in the same nodes. The implications for scheduling are, how to assign jobs to nodes that would minimize storage-related bottlenecks [6, 17]. Scheduling matters even for servers without persistent storage, if they use distributed in-memory storage or caching [2].

In addition to performance considerations, storage considerations also complicate other scheduling issues, such as operations (Sec. 3). Collocating storage with computation reduces the latitude for scheduling and other cluster management operations (e.g., they cannot just be powered down if other servers depend on their storage).

Finally, it is interesting to note that there is also cross pollination of ideas back from Web scheduling to HPC scheduling. One such idea is the provision of probabilistic SLAs, as suggested in the work on probabilistic backfilling [22].

6 Conclusion

It is hard to escape the realization that computer science evolves in cycles. Centralized mainframes were the common computing platform for the 1960s and 1970s. We later saw great decentralization of computation in the 1980s and 1990s with powerful personal computers. Now, owing to the sheer size of the problems we tackle and the increased scaling in management, we are again centralizing much of our computing infrastructure, this time in very large datacenters.

As in previous shifts in computer architecture organization, this trend opens a score of new problems and research opportunities. It appears, however, that much of this activity lies behind closed doors, in the R&D facilities of large Web companies. This paper hopes to shed some light on this activity, by exposing some problems and challenges, as well as describing some working novel solutions that have not been openly discussed yet.

The key aspects discussed were (1) the need and opportunities created by multiple SLA tiers, and (2) the stringent operational requirements of modern Web-scale datacenters. The backup task technology described in section 4 complements the overall discussion by illustrating how the technologies designed to tackle these challenges differ from those of the better studied HPC datacenters.

References

1. Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing amazon ec2 spot instance pricing. In *CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011.
2. Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated memory caching for parallel jobs. In *Ninth USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, April 2012.
3. David G. Andersen, Jaosn Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, New York, NY, 2009. ACM. `portal.acm.org/citation.cfm?id=1629577`.
4. Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'12, pages 53–64, New York, NY, USA, June 2012. ACM.
5. Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007. `citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.5419`.

6. Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.
7. Allan Bricker, Michael Litzkow, and Miron Livny. Condor technical summary, version 4.1b. Technical Report CS-TR-92-1069, January 1992. `citeseer.ist.psu.edu/briker91condor.html`.
8. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
9. Dror G. Feitelson. Metrics for parallel job scheduling and their convergence. In Dror G. Feitelson and Larry Rudolph, editors, *Seventh Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 188–1205. Springer Verlag, 2001. `www.cs.huji.ac.il/~feit/parsched/`.
10. Dror G. Feitelson, Larry Rudolph, and Uwe Schwigelshohn. Parallel job scheduling – A status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2004. `www.cs.huji.ac.il/~feit/parsched/`.
11. Eitan Frachtenberg and Dror G. Feitelson. Pitfalls in parallel job scheduling evaluation. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshon, editors, *11th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3834 of *Lecture Notes in Computer Science*, pages 257–282. Springer-Verlag, Boston, MA, June 2005. `frachtenberg.org/eitan/pubs/`.
12. Eitan Frachtenberg and Uwe Schwiegelshohn. New challenges of parallel job scheduling. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–23. Springer Verlag, 2007. Lect. Notes Comput. Sci. vol. 4942.
13. Graham Holt. Time-critical scheduling on a well utilised HPC system at ECMWF using LoadLeveler with resource reservation. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 102–124. Springer-Verlag, 2004. `www.cs.huji.ac.il/~feit/parsched/`.
14. David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the Maui scheduler. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
15. Terry Jones, William Tuel, Larry Brenner, Jeff Fier, Patrick Caffrey, Shawn Dawson, Rob Neely, Robert Blackmore, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *15th IEEE/ACM Supercomputing*, Phoenix, AZ, November 2003. ACM Press and IEEE Computer Society Press. `www.sc-conference.org/sc2003/paperpdfs/pap136.pdf`.
16. Richard Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. 1972.
17. Rini T. Kaushik and Milind Bhandarkar. Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association.

18. Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, March 2010.
19. Jason Mars, Linjja Tang, Robert Hundt, Kevin Skadron, and Mary Lou Souffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011*, New York, NY, USA, 2011.
20. David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 205–216, New York, NY, USA, 2009. ACM.
21. Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *SIGMETRICS Performance Evaluation Review*, 37(4):34–41, March 2010.
22. Avi Nissimov and Dror G. Feitelson. Probabilistic backfilling. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 102–115. Springer Verlag, 2007. Lect. Notes Comput. Sci. vol. 4942.
23. Ryan Paul. A behind-the-scenes look at Facebook release engineering. `arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/`, April 2012.
24. Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 77–84, New York, NY, USA, 2009. ACM.
25. Gerald Sabin and P. Sadayappan. Unfairness metrics for space-sharing parallel job schedulers. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshon, editors, *11th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3834 of *Lecture Notes in Computer Science*, pages 238–256. Springer-Verlag, Boston, MA, June 2005. `www.cs.huji.ac.il/~feit/parsched/`.
26. Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, February 2006.
27. Angela C. Sodan. Adaptive scheduling for QoS virtual machines under different resource allocation—performance effects and predictability. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 259–279. Springer Verlag, 2009. Lect. Notes Comput. Sci. vol. 5798.
28. Tom White. *Hadoop: The Definitive Guide*. Yahoo! Press, USA, 2010.
29. Kaiqi Xiong and Sang Suh. Resource provisioning in SLA-based cluster computing. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–15. Springer Verlag, 2010. Lect. Notes Comput. Sci. vol. 6253.