Evaluating job packing in warehouse-scale computing

Abhishek Verma, Madhukar Korupolu, John Wilkes
Google Inc.

***Abstract*—One of the key factors in selecting a good scheduling algorithm is using an appropriate metric for comparing schedulers. But which metric should be used when evaluating schedulers for warehouse-scale (cloud) clusters, which have machines of different types and sizes, heterogeneous workloads with dependencies and constraints on task placement, and long-running services that consume a large fraction of the total resources? Traditional scheduler evaluations that focus on metrics such as queuing delay, makespan, and running time fail to capture important behaviors – and ones that rely on workload synthesis and scaling often ignore important factors such as constraints. This paper explains some of the complexities and issues in evaluating warehouse scale schedulers, focusing on what we find to be the single most important aspect in practice: how well they pack long-running services into a cluster. We describe and compare four metrics for evaluating the packing efficiency of schedulers in increasing order of sophistication: aggregate utilization, hole filling, workload inflation and cluster compaction.**

I. Introduction

At the very minimum, we wish that all articles about job schedulers, either real or paper design, make clear their assumptions about the workload, the permissible actions allowed by the system, and the metric that is being optimized.

– Dror Feitelson, 1996 [11]

There have been many studies of scheduling systems for compute clusters, but most evaluations of their effectiveness cannot be directly applied to the warehouse-scale environments that are behind cloud computing and high-performance web services [3]. A significant scheduling improvement here might produce a few percentage points of benefit. That may seem small in relative terms, but at this scale, we risk wasting millions of dollars without reliable, repeatable ways of quantifying the impact of those improvements. This paper explores techniques to do so. Its primary contributions are to explain why this is not a solved problem; describe and characterize four approaches (including a novel cluster compaction technique) to evaluating schedulers; and demonstrate their behavior by analysing some simple scheduler algorithms against real, production workloads that run on tens of thousands of machines.

A. Heterogeneity

Real-life warehouse-scale data centers exhibit heterogeneity that is often ignored on smaller scales, or in high-performance computing (HPC) studies. A month-long workload trace from a Google cluster [31] illustrates many of these aspects [25].

Machines are heterogeneous: A scheduler evaluation should not assume homogeneous machines. Even when a datacenter has just been constructed, its machines will have different numbers of disks, varying amounts of memory, differentiated network connections (e.g., some will have external IP connections), special peripherals (tape drives, flash memory), and processor types (e.g., GPUs or low-power processors). This variety only increases as new machines are brought in to replace broken ones, or the cluster is expanded [4], [25].

Keeping track of resources that have been allocated on a machine is not enough: the opportunity cost of placing a task on a machine needs to be considered too – e.g., the largest machines may need to be reserved for running the largest tasks, or else they won't fit. A similar issue arises for workloads that are picky, and can only run in a few places because of constraints [26], or because machines with the right attributes are scarce. Effective packing requires that other work is steered away from the scarce machines such picky work will require.

Heterogeneous machines will have different costs, too; because of issues like physical plant infrastructure and operational expenses, that cost may not always be a simple function of the quantity of resources (e.g., the total amount of RAM).

Workloads are heterogeneous: some work is structured as finite-duration batch jobs; some as long-running services. Some of those workloads are latency-sensitive (e.g., user-facing web functions); others not (e.g., long-running batch jobs). Some are small (e.g., a single task consuming a few MB of memory); some may be huge (e.g., TBs of memory, spread across thousands of machines). Availability, performance, and cost-effectiveness needs may vary widely, too [25], [33]. The union of the variety of properties and requirements mean that it's naive to assume homogeneous workloads in evaluations.

Evaluation criteria vary: scheduler evaluations serve different purposes. Capacity planners want to know how many machines they need to buy to support a planned workload; operators of a cluster want to know how many additional users they can add, or how much their existing users can grow their workloads. Scheduler designers are interested in both.

B. Other issues

Long-running jobs: Work in the Grid and HPC community has traditionally assumed that the workload is made up of batch jobs that complete and depart, and hence emphasized makespan or total time to completion, perhaps combined with user- and session-based modeling (e.g., [34]). But much of the workload for modern warehouse-scale data centers consists of long-running service jobs, so we need different evaluation criteria.

Constraints: placement constraints are critical for some applications to meet their performance and availability goals, but it is rare to find scheduler evaluations that take these into account. Similarly, machines with different attributes, as well as different amounts of resources, are inevitable in all but the smallest warehouse-scale installations, and these too are rarely discussed.

Workload modeling is surprisingly hard to do well: the presence of long-running jobs, constraints, preferences, disparate job shapes (the number of tasks, CPU-to-memory ratios, and the absolute sizes of those values) mean that varying workloads by naive techniques such as changing the inter-arrival rate of requests is unlikely to be appropriate. This is especially true for batch jobs, for which complications such as inter-job dependencies, inter-arrival rates and user response may be critical [11], [13]. Although it may be helpful to use simplified models while *developing* scheduling algorithms (i.e., ignoring much of the heterogeneity described above), adopting them when *evaluating* job packing at warehouse scale may lead to incorrect – and expensive – choices.

Incorrect metrics: Metrics that focus on time to completion for batch jobs miss the point. Many jobs in warehouse-scale clusters are long running services, and these are often more important than batch jobs: in our environment, long-running services can evict batch jobs to obtain resources if necessary. Although increased utilization is certainly a side-effect of better packing, simply measuring utilization (usage divided by capacity) is not enough – as it ignores the need to build in headroom for handling workload spikes, constraints and heterogeneity, and is heavily dependent on workload behaviors [11]. We need something better.

C. Appropriate metrics

We are not alone in observing the problem. Frachtenberg et. al [13] provide a summary of common pitfalls in scheduler evaluations for run-to-completion workloads, including job inter-arrival and running times, and open vs closed queueing models. And we echo the plea made nearly two decades ago that we quoted in the introduction [11].

A precise definition of scheduler "packing efficiency" is hard to pin down in the face of all the complications listed above. This paper offers four different metrics and approaches, and investigates their tradeoffs. Each evaluation metric defines packing efficiency somewhat differently.

We note that there are several other measures such as a scheduler's runtime performance, the amount of resources it needs to execute, fairness across users or jobs, power and energy consumption, predictability, repeatability, makespan minimization etc. All of these are important, but in our environment these are second-order considerations after packing efficiency, because we size our clusters around the needs of these long-running services. Other, lower-priority, work is fitted around their needs.

We focus here on the how well a scheduler can pack a workload into a cluster at a single moment in time. How well an online scheduler handles a stream of job requests is certainly relevant, too, but that is much harder to evaluate, requiring trace replaying, which can be quite cumbersome at warehouse scale. Replaying a four hour workload trace for a medium sized cluster at high fidelity takes us more than an hour. We'd ideally prefer to replay a multi-week trace (e.g., [31] perhaps combined with bootstrapping techniques [29] to provide variations), but this is not very practical at this scale. Indeed, almost all external users of the trace [31] use only a tiny fraction of it.

Fortunately, with long-lived service jobs, semi-static placement remains a key part of packing efficiency, and we focus our efforts on exploring how to carefully evaluate how good a scheduler is at this. As we will see, there are many details that have to be done right to do this evaluation well.

II. Background

We begin by introducing our system model, and then describe the experimental methodology we use for comparing our evaluation techniques.

A. System model

A warehouse-scale (cloud) workload is typically structured as a set of *jobs* that are composed of one or more *tasks*; each task is scheduled onto a machine or marked *pending*. A task has a set of *requirements* that specify how many resources (e.g., CPU, RAM) it needs; these may vary over time. A task's *usage* represents the amount of resources it actually consumes, which is usually smaller than the requirement. A task may have *constraints* that restrict where it may be placed, for example, by referring to *attributes* associated with machines (e.g., OS version, processor type). A job can also have bounds on the number of tasks that can be placed on the same machine or rack, typically implemented as special types of constraints.

Such workloads generally consist of a mix of service jobs, which can run for months, and batch jobs, which can complete in a few minutes, or last for many hours or even days. A job's requirements and constraints are known to the scheduler only upon its arrival and the duration of the tasks is typically not known before the job starts, so scheduling is an online process.

B. Experimental methodology

We use an offline simulator of our production scheduling system to compare the packing behavior of different schedulers. Each simulator run begins by feeding it a point-in-time snapshot (checkpoint) state of a real production Google cluster containing several thousand machines. This state includes data about all the machines, jobs and tasks, whether running or pending, as well as information about the placement and usage of running tasks. The data we use for this paper was obtained from checkpoints taken at 2013-08-01 14:00 PDT (21:00 UTC). A detailed analysis of a representative workload for a similar Google cluster is given in [25].

The simulator wraps our real-life scheduler production code, and emulates its interaction with the outside world (e.g., with the machines in the cluster). When we ask the simulated scheduler to pack the workload onto the cluster's machines, it uses the exact same algorithms as our production scheduler.

To simplify the exposition, we made a few changes: we eliminated all non-production-priority workloads from the cluster, so none of the data reported here indicates the utilizations we achieve in practice; we remove special and broken machines and all tasks assigned to them from the checkpoint. To assess the repeatability of the experiments, we run eleven trials for each data point and show error bars in all figures indicating the full range of the values across trials. For several

TABLE I. SUMMARY OF TECHNIQUES FOR EVALUATING PACKING EFFECTIVENESS.

	Aggregate utilization	**Hole filling**	**Workload inflation**	**Cluster compaction**
Accuracy	low	medium	high	high
Fragmentation/stranding	no	yes	yes	yes
Attributes/constraints	no	no	yes	yes
Time for computation	$<$ 1 min	$\sim$ 30 mins	$\sim$ 2 hours	$\sim$ 5 hours
Context where useful	quick approximation	fixed-size slot counts	cluster operators	capacity planners

figures the error bars appear as a small horizontal bar indicating that the variations are small.

We applied our techniques to eight clusters; space precludes us from providing data for more than two representative ones, which we refer to as Cluster 1 and Cluster 2. Each contains several thousand machines and is running a live production workload; the clusters and individual machines are shared by many users. We first demonstrate the different techniques using Cluster 1 and later use data from both clusters to measure the packing efficiency of job ordering policies.

We reiterate that the purpose of this paper is to compare *scheduler-evaluation techniques*, not schedulers. But to give the evaluation techniques something to evaluate, we concocted a few scheduler policies by tweaking the existing production schedulers in a specific way, described further in Section IV. Note that the full complexity of the production scheduler was retained – all we changed was one small aspect: the order in which tasks were examined while being scheduled.

III. EVALUATION TECHNIQUES

We compare four techniques for evaluating packing efficiency. In order of increasing sophistication, they are: *aggregate utilization, hole filling, workload inflation* and *cluster compaction*. Table I provides a quick comparison of the four.

A. Aggregate utilization

Probably the simplest, and most commonly used, efficiency metric is the *aggregate utilization*: the fraction of the cluster's resource capacity of each type (e.g., CPU, RAM) that has been allocated. For example, Cluster 1 has 68% CPU utilization and 48% memory utilization. This metric is simple to compute, but has some shortcomings: (1) It cannot distinguish between schedulers that place all tasks, since the utilizations are the same. (2) It hides fragmentation effects: the "holes" of unused resources may be too small to be useful. For example, even though aggregate CPU resources are available, they may be fragmented across several machines each of which may be too small to be useful for tasks. (3) It hides stranding effects since CPU and memory utilization are independently computed: for example, even though CPU resources may be available on a machine, tasks may not be able to schedule on it because there is no spare memory.

Because of these problems, we don't usually find this metric helpful, and won't pursue it further in this paper.

B. Hole filling

The shortcomings of the aggregate utilization metric can be partially addressed by looking at the unallocated capacity on each machine after task placement (the holes). This technique discounts fragmentation effects by counting how many appropriately-sized units of size U can fit into the holes, thereby discounting unallocated spaces that are too small to be useful. The size U can be picked in several different ways. It can be based on the overall capacity of the machines in the cluster (e.g., 30% of the median-sized machine), tailored to individual machines (e.g., 30% of a particular machine's size), based on the workload (e.g., 90%ile of the task request sizes), or picked by the execution environment (e.g., the memory slot size in Hadoop).

It is important that the units each have multiple resource dimensions (e.g., CPU and RAM) to account for both fragmentation and stranding. The algorithm is straightforward: simply count how many units will fit into the unallocated resources, one machine at a time, multiply by the unit size U, and add that to the utilization achieved before hole filling to indicate what might be possible, and thereby providing a measure of how much wastage the scheduler's packing has. We report the result as a percentage of the total cluster capacity.

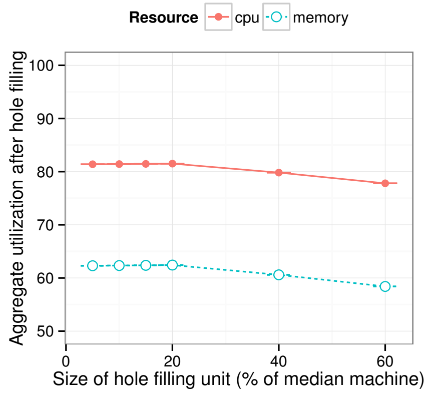

Fig. 1. Hole filling: aggregate CPU and memory utilization after hole filling as a function of the unit size U, expressed as a fraction of the median machine size. Both memory and CPU are scaled together. Data for Cluster 1.

Figure 1 shows the CPU and memory utilization achieved after hole-filling for cluster 1.[1] Without hole-filling, the aggregate utilization of the cluster is about 68% for CPU and 48% for memory. Using units whose sizes are 5% of the median machine size, hole filling can grow up to 80% of the CPU and 62% of the memory; the remaining capacity represents resources lost due to fragmentation and stranding. As expected, larger unit sizes result in more lost capacity, as it is harder to fit bigger units into the fragmented space that remains.

The hole-filling metric is almost as fast to calculate as

[1]Reminder: this is after removing all but "production" work, so is not an accurate reflection of the utilization we achieve in practice.

aggregate utilization, but it uses a single fixed unit U to determine what would fit into the holes, which is not a good match to the heterogeneous task sizes seen in real workloads. Fixing this by using multiple unit sizes would turn it into a bin-packing scheduling pass, losing the simplicity of the evaluation technique. And what mix of sizes would be appropriate? Hole-filling also ignores task constraints and machine attributes, thus providing an optimistic assessment of the remaining capacity and the efficacy of the scheduler's packing.

C. *Workload inflation*

Workload inflation addresses the limitations of hole filling by scaling up the original workload until it no longer fits. This gives a more accurate estimate of how much new workload could be added to a cluster. A few things have to be done correctly to make this work well. Workload heterogeneity needs to be maintained; this is done by scaling all jobs uniformly (using a scale factor) or by randomly sampling them using a Monte Carlo method. Similarly, the effects of constraints are retained by replicating existing constraints when jobs are grown or new copies are added.

Algorithm 1 gives an outline for workload inflation using a scale factor $f > 1$.

We distinguish two kinds of scaling. *Horizontal scaling* grows the total number of tasks by $(f-1)$, but retains the original task sizes. (This can be achieved by growing the number of tasks in the original job, which can cause difficulties if jobs are very large, or by cloning jobs; we do the latter.) *Vertical scaling* inflates the resource requirement of each task by the factor f, thus growing each task, while retaining the same number of tasks.

A task is considered *picky* if it is "too big" – larger than a threshold percentage of the median machine – or is constrained to "too few" machines – a threshold fraction of the cluster size. Both thresholds are set to 60% in our experiments. Inflating a non-picky (*conforming*) job is unlikely to cause problems, but inflating a picky job can quickly cause it to go pending, thereby ignoring the load it induces or stopping the inflation process prematurely. So one parameter to Algorithm 1 specifies whether to inflate all jobs or only the conforming ones.

With vertical inflation, there are a few more choices to be made. For example, what should be done with a task that is scaled to become larger than the biggest machine? We could let it go pending or clip it at the largest machine size. The former mimics what an automatically-scaled system might do; the latter represents how users would behave if they noticed that their jobs were going pending. Both options give a slightly different view of the workload growth that can be achieved; we picked the latter on the grounds that it is more realistic and gives larger scale factors.

Figure 2 shows what happens when workload inflation is applied to cluster 1. As expected, at scale factor $f = 1$ there are no pending tasks. As the scale factor increases the horizontal-all and vertical-all schemes immediately cause some tasks to go pending, probably because of picky tasks that are not able to schedule. The horizontal-conforming and vertical-conforming schemes do not suffer this problem, because they scale only conforming, non-picky tasks. After scale factor $f > 1.25$ the number of pending tasks ramps up quickly for all the inflation methods.

Algorithm 1 Workload inflation.

Input: cluster checkpoint with a list of *machines*
Output: CPU and memory utilization after inflation, number of pending tasks after inflation
Parameters: scaling factors, horizontal or vertical, conforming or all

```
 1. $\mathscr{J} \leftarrow$ set of all jobs
 2. **if** conforming **then**
 3.     $\mathscr{J} \leftarrow$ set of only conforming jobs
 4. **end if**
 5. **for each** $f$ **in** scaling factors **do**
 6.     **for each** $j$ **in** $\mathscr{J}$ **do**
 7.         **if** horizontal inflation **then**
 8.             $j' \leftarrow clone(j)$
 9.             $j'_{tasks} \leftarrow \lceil j_{tasks} \times (f-1) \rceil$
10.         **else if** vertical inflation **then**
11.             **for each** $t$ **in** tasks of $j$ **do**
12.                 $t_{cpu} \leftarrow t_{cpu} \times f$
13.                 $t_{mem} \leftarrow t_{mem} \times f$
14.             **end for**
15.         **end if**
16.     **end for**
17.     Schedule all tasks on *machines*
18.     Output number of pending tasks
19.     Output aggregate CPU and memory utilization after inflation
20. **end for**
```

The achieved utilization at scale factor of 1.25 is slightly under 80% for CPU and under 60% for memory and increases only a little with larger scale factors, although the number of pending tasks continues to grow.

Not surprisingly, inflating only conforming tasks results in larger scale factors before the cluster runs out of capacity, and the pending task graph for the conforming-only case is to the right of the all-tasks case.

One problem with inflation is determining when to stop: it is not always obvious what fraction of pending tasks should be permitted, because the number is a function of the particular workload applied to a cluster. In practice, we tend to "eyeball" the graphs, and look at the knee of the pending-tasks curves. This is not easy to automate.

An alternative that does better in this regard is to use a Monte-Carlo technique to inflate the workload (see Algorithm 2) [23]. At each step this picks a job at random from the existing workload and clones it, repeating the process until a chosen stopping criterion is reached (e.g., the number of pending tasks exceeds a chosen threshold, such as 1%). The output is the aggregate CPU and memory utilization achieved for that threshold. This procedure can be repeated many times to provide greater confidence in the result, and ensure that enough random samples have been taken to cover even the rare, picky jobs. Sensitivity analysis shows that the achieved utilization is fairly stable with respect to the stopping criterion: varying the pending-task threshold from 2% to 30% increases achieved utilization only by about 2%.

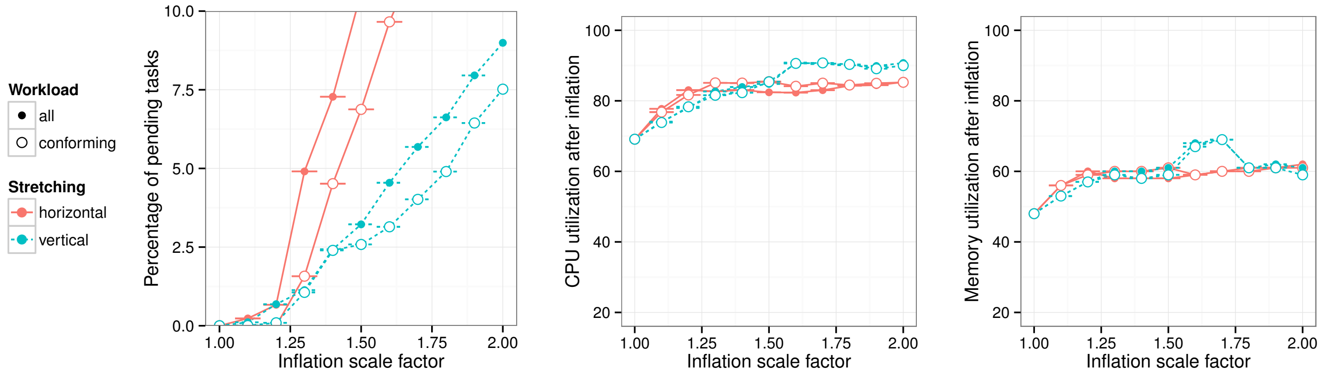

Fig. 2. Workload inflation: pending tasks, and CPU and memory utilization, as a function of the inflation scale factor f in Cluster 1. The lines show the effects of inflating all or just conforming jobs, using horizontal and vertical scaling. The effective inflation limit is the knee in the pending-tasks curve, at the point where just a "few" tasks have gone pending; beyond this point, utilization doesn't increase much.

Algorithm 2 Monte-Carlo workload inflation.

Input: cluster checkpoint with a list of *machines*
Output: CPU and memory utilization after inflation, number of pending tasks after inflation
Parameters: fraction of tasks allowed to go pending τ, conforming or all

```
1.  $\mathscr{J} \leftarrow$ set of all jobs
2.  **if** conforming **then**
3.      $\mathscr{J} \leftarrow$ set of only conforming jobs
4.  **end if**
5.  **while** fraction of pending tasks $< \tau$ **do**
6.      Pick $j$ uniformly at random from $\mathscr{J}$
7.      $j' \leftarrow clone(j)$
8.      Schedule all tasks on existing *machines*
9.  **end while**
10. Output number of pending tasks
11. Output aggregate CPU and memory utilization after inflation
```

D. Cluster compaction

Cluster compaction takes the opposite approach: treat the workload as fixed, and shrink the cluster until the workload no longer fits. This directly answers "how small a cluster could be used to run this workload?" and avoids the complexity of workload scaling. When machines are uniform without constraints and fragmentation, this would roughly correspond to the average utilization metric defined earlier. Presence of heterogeneity, constraints and fragmentation make this more challenging for warehouse scale datacenters.

The compaction process is described in Algorithm 3. The basic idea is to generate a random permutation of machines in the cluster, and then use binary search to determine the minimum prefix of the permutation required to run the workload. The end point of the binary search is returned as the *compacted cluster size* for this trial. Repeated trials are run to obtain a distribution of the compacted cluster sizes and to eliminate the effects of an unfortunate permutation (e.g., all big machines being at one end). Figure 3 shows the 90%ile of eleven trials and the error bars indicate the full range.

There are some key features, policy choices and details to note about the compaction process:

Algorithm 3 Cluster compaction algorithm.

Input: cluster checkpoint with a list of *machines*
Output: minimum number of machines needed
Parameters: fraction of tasks allowed to go pending τ, error bound for the number of machines needed μ

```
1.  **while** fraction of pending tasks $> \tau$ **do**
2.      $machines \leftarrow machines + clone(machines)$
3.      Schedule all tasks on *machines*
4.  **end while**
5.  Generate a random permutation $p$ of *machines*
6.  $min \leftarrow 0, max \leftarrow length(p) + 1$
7.  **while** $max - min \geq \mu$ **do**
8.      $mid \leftarrow \lfloor (min + max)/2 \rfloor$
9.      Disable all but the first *mid* machines in $p$
10.     Discard tasks (e.g., storage servers) that were tied to the disabled machines.
11.     Reschedule all tasks on the enabled machines
12.     **if** fraction of pending tasks $> \tau$ **then**
13.         $min \leftarrow mid$
14.     **else**
15.         $max \leftarrow mid$
16.     **end if**
17. **end while**
18. Output *mid* as the estimate for the minimum number of machines needed
```

Maintaining machine heterogeneity. The use of random permutation in step 5 helps to maintain the machine heterogeneity probabilistically. As machines are removed from one end of the permutation (step 9), the remaining machines on average have similar heterogeneity to the original set in terms of resource capacities and attributes. For clusters with rare combinations of heterogeneity and constraints (e.g., few rare machines out of n and a job being constrained to only those) this wouldn't work well, but our clusters do not have such adversarial combinations in practice.

Repeated trials and distribution: As mentioned above, we run repeated trials to obtain a distribution of the compacted-cluster-sizes and to eliminate the effects of an unfortunate permutation. A high percentile (e.g., 90%ile) of the repeated trials is used as the overall compacted cluster size, on the grounds that this is a conservative approximation to "how many machines would we need?" than just the mean or median. Error bars are shown in the plots to illustrate the variance involved.

Stopping criterion. The strictest one is to end compaction as soon as there are any pending tasks. This has the advantage of simplicity but is vulnerable to even a single extremely hard-to-schedule task. To overcome such picky jobs, we use the parameter τ (typically 0.6%) to allow a small fraction of tasks to go pending. Figure 3 shows the sensitivity of the compacted cluster size as the stopping criterion goes τ from 0 to 1%.

Constraints that tie tasks to particular machines. Some tasks are constrained to run on a small number of machines, or even specific machines, and go pending if these machines are removed from the cluster during compaction. Either the constraints must be relaxed or we simply allow the tasks to go pending, and count against the threshold τ in Algorithm 3. Such tasks are rare, so we choose the latter course.

Special jobs and preprocessing. Some jobs (e.g., file system servers) have tasks on all the machines in the cluster and have to be handled specially. The simplest way is to delete the associated file server task when the machine is removed. The resulting reduction in storage capacity can be compensated by moving the storage workload to other servers or by ignoring the issue on the grounds that storage capacity isn't scarce, which is what we do in the experiments reported below.

Cluster expansion. With some experiments, the cluster size might need to increase (e.g., if a scheduler policy that fosters better compaction is disabled, or if the workload is being inflated). This can be done by adding in clones of the original cluster (step 2), to preserve the existing machine heterogeneity, and then compacting the resulting combined cluster.

Running time: The number of binary search iterations in the compaction algorithm is proportional to $log(M/\mu)$, where M is the number of machines in the cluster and μ is the error bound (step 7). If desired, running time can be improved at the cost of accuracy by making μ larger and foregoing the last few iterations of the binary search.

Compaction has similarities to power-based right-sizing of data centers [19], [28] and virtualization enabled server-consolidation [30] in data centers, but their goals are different. While those studies aim to minimize the cost or number of servers needed to serve the given workload, our goal is to use the compacted cluster size as a metric to compare the given scheduling or packing algorithms.

If *no* tasks are allowed to go pending then the compacted cluster size is around 92% of the original (Figure 3). As we allow a few more to do so, the compacted cluster size decreases to 82% with up to 0.4% pending tasks, and eventually stabilizes around 80% with 1% pending tasks. Constraints and picky jobs are the cause: they make it hard to remove certain machines, which causes the compaction algorithm to stop.

We would expect the compacted cluster size to be about $1/f$ where f is the scale factor achieved by inflation, and indeed this is roughly what we see (compaction produces a little more than 80%, while horizontal conformal inflation sees $f = 1.25$). The two techniques roughly agree in this case. However, the presence of constraints, workload peculiarities and other factors can cause this to vary. Similarly, note that the aggregate utilization would suggest that the compacted cluster size should be about 68% – but that doesn't take fragmentation, heterogeneity and constraints into account.

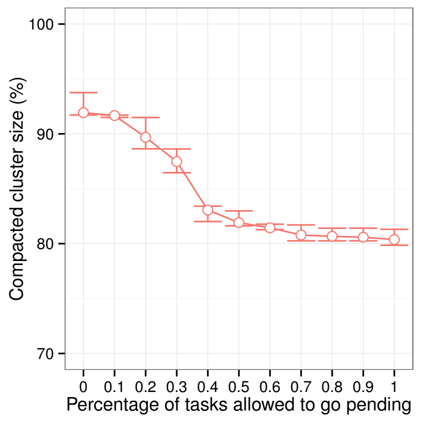

Fig. 3. Cluster compaction: number of compacted machines as a function of the fraction of tasks that are permitted to go pending. The plotted point is the 90%ile of 11 runs, error bars show the full range. Data for Cluster 1.

IV. Using the techniques: evaluating different job orders

As we mention above, this paper isn't about scheduling algorithms, but about ways to evaluate them – but to test these approaches, we needed some scheduling algorithms to compare. Here's what we did.

Our production scheduling algorithms are all online: they do not know about the future arrival of tasks and can rarely migrate tasks between machines. Could we achieve better packing of tasks if we looked at all the jobs that are running now, and repack them? If so, in what order should the jobs be looked at? We picked three intentionally-simple variations: decreasing cpu, decreasing memory, and decreasing normalized sum of cpu and memory (where normalization is done by dividing cpu and memory requirements by the size of the median machine in that cluster). The baseline is arrival, which reflects the sequence in which jobs arrived. We also fixed the order of the machines and used first fit for scheduling these incoming jobs – thus effectively emulating a First Fit Decreasing bin packing [2] where "decreasing" is either by cpu, memory or normalized sum.

Figure 4 shows the analyses of these different policies on data from Cluster 1 using horizontal inflation, vertical inflation, hole filling and compaction:

- Horizontal scaling reports that the scheduler can support a workload that grows by a scale factor f of about 1.4 for arrival and sum, and by 1.3 for cpu and memory.
- Vertical inflation produces slightly bigger scale factors: 1.5 for arrival, 1.4 for cpu, 1.3 for sum and

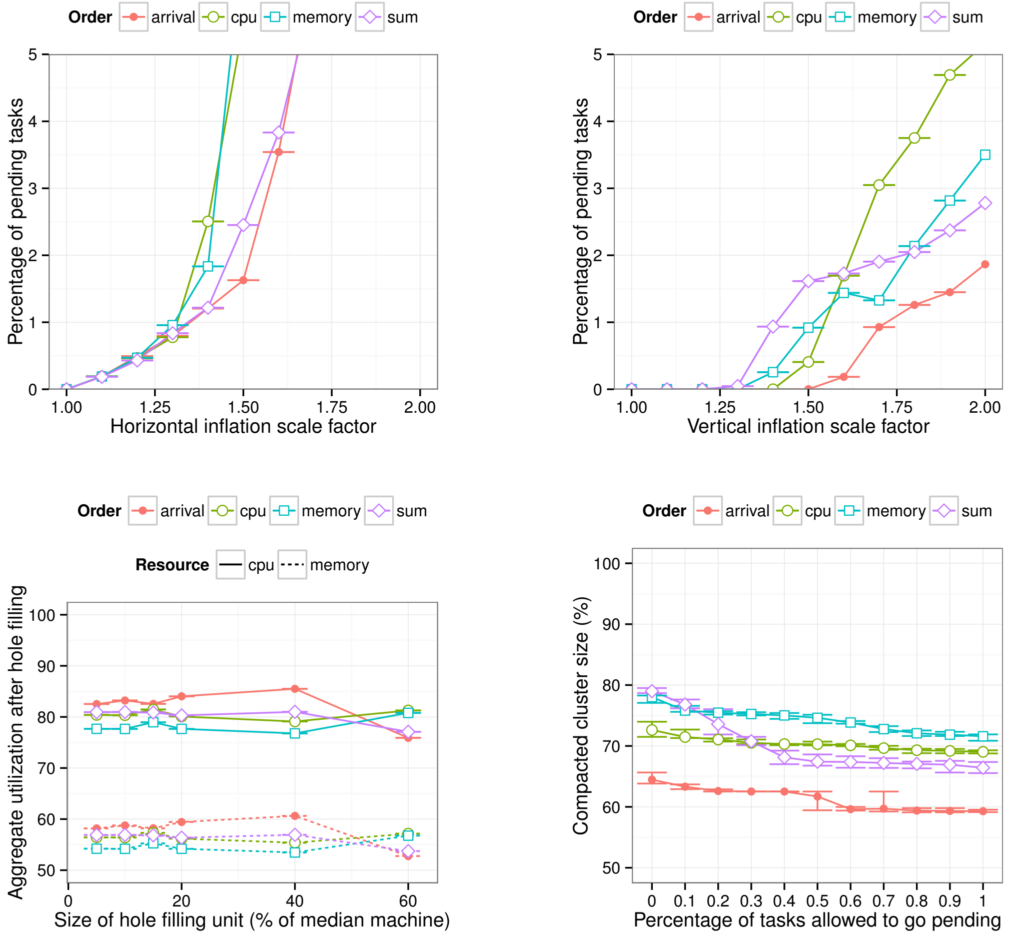

Fig. 4. Four different scheduling algorithms (represented by different lines), evaluated using horizontal inflation, vertical inflation, hole filling and cluster compaction (the different plots). Data from Cluster 1.

memory orders. However they have different slopes after the inflection point is reached.

- Hole filling shows that for most unit sizes, arrival order can achieve the highest utilization, cpu and sum orders are barely distinguishable, and are followed by memory. But the order changes with units above about 50% of the median machine size.
- Workload compaction with a pending tasks threshold of 0.4%, suggests that we can compact the cluster down to 60% of its original size using arrival, 68% using sum, 70% using cpu and 75% using memory.

All four techniques agree on arrival being the best for this cluster, followed by cpu or sum, and finally memory.

Ordering jobs by their cpu or memory requirements causes resource stranding: the scheduler does a good job of bin-packing machines in the preferred dimension, but often leaves other resources idle. sum does better, by encouraging a more balanced approach to the two resource dimensions. But arrival outperforms the others – probably because it (effectively) randomizes the order of jobs and this leads to less stranding.

Figure 5 shows the same data as in Figure 4, but for a different cluster. Here:

- Horizontal scaling can achieve a workload scaling factor of 2 using sum or arrival, and by 1.9 using cpu or memory.
- As before, vertical inflation can achieve an even higher scaling factor: 2.6 using sum or arrival, and 2.5 using cpu or memory.
- Hole filling consistently shows that arrival and sum order are better than cpu and memory.
- Compaction can shrink the cluster size down to 43% using sum, 55% using arrival, and 60% using cpu or memory.

All four techniques agree that sum and arrival are better than cpu and memory, with compaction biased towards sum. Based on this analysis, we would recommend using sum if the goal is to use fewer machines.

We performed similar experiments with six other clusters and found that workload inflation and cluster compaction techniques agree on the ranking order in the majority of the

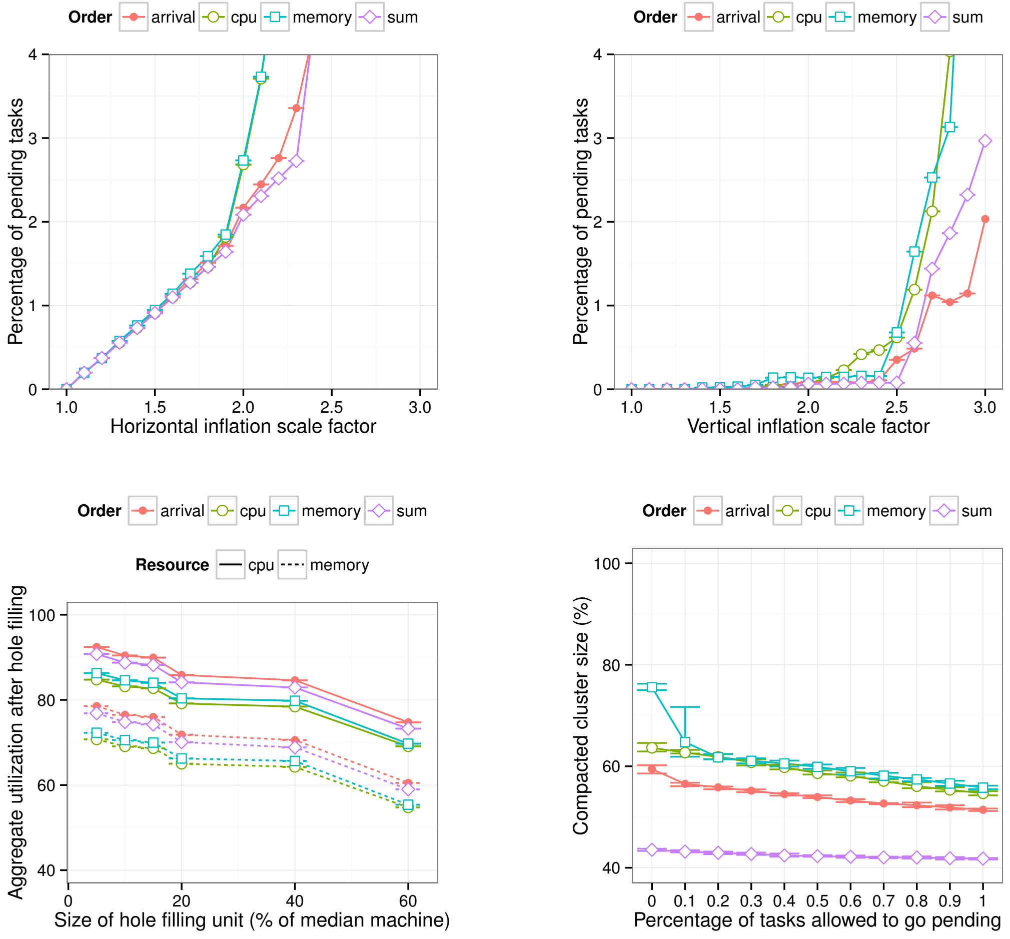

Fig. 5. These are the same plots as in Figure 4, but using data from Cluster 2. The four graphs correspond to different techniques.

cases and hole-filling is able to distinguish only when the cluster load is sufficiently high.

V. Related work

Job scheduling has seen a rich body of work ranging from kernel level scheduling (on a single computer) to scheduling in grids and large warehouse-scale clusters. For a comprehensive survey of scheduling and resource management in cloud clusters, we refer the reader to [18]. For job scheduling strategies in grid context, please see [9], [16], [22].

Warehouse scale clusters have been described and studied in [3], [21]. Detailed analyses and characterization of workloads from such large warehouse scale clusters have been presented in [25], [26], [31]. These analyses highlight the challenges of scale, heterogeneity and constraints inherent in such datacenters.

Much of the prior work in evaluating cluster schedulers has focused on short-lived batch tasks and runtime modeling ([5], [10], [12], [20]), fairness across users or jobs ([8], [15], [32]), and power-minimization ([6], [7], [14]).

Efficient packing of long-running services is an important part of improving utilization in large data centers and is relatively less explored. For example, [17] studies job packing but not at warehouse scale. Algorithms and heuristics for vector bin packing have been studied in various domains ([1], [24], [27], [30]), but they are usually offline methods with the objective of minimizing the number of bins used. And they often ignore the heterogeneity and constraints that are common in warehouse scale datacenters.

VI. Conclusion

In this paper, we address the question of evaluating schedulers in warehouse-scale clusters where the important problem is of packing long-running tasks. Addressing the complexities of heterogeneity and constraints that arise in such clusters, we propose and evaluate four approaches and metrics: aggregate utilization, hole filling, workload inflation and cluster compaction. These approaches have different sweet spots:

1) *Aggregate utilization*: can give a quick, simple measure of overall resource utilization without consideration of fragmentation, stranding and constraints. Not useful for comparing scheduling algorithms especially if they can pack all jobs at low loads.
2) *Hole-filling*: fast, simple assessment of unallocated resources using fixed size slot counting. Unfortu-

nately, it ignores constraints and workload heterogeneity, and cannot make fine grained distinctions between scheduling policies, especially at lower utilization levels.

3) *Workload inflation*: answers "what if?" questions about future workload growth, and is most useful for cluster operators who want to know how much additional workload can be added to a cluster. Provides multiple policy choices on what is inflated and how.
4) *Cluster Compaction*: good for capacity planning and for providing an automated comparison metric for fine-grained distinctions. It has fewer policy choices than inflation and a longer running time.

Which one is best? It depends on the kind of what-if questions you ask. Regardless of which approach you pick, we urge authors of scheduling papers to be explicit about both their assessment objectives and the policy choices they make. We hope this paper has provided information on how to do so, thereby increasing the repeatability of research in this space.

References

[1] Y. Azar, I. R. Cohen, S. Kamara, and B. Shepherd, "Tight bounds for online vector bin packing," in *Proc. ACM Symp. on Theory of Computing*, San Francisco, CA, USA, 2013, pp. 961–970.

[2] B. S. Baker, "A new proof for the first-fit decreasing bin-packing algorithm," *Journal of Algorithms*, vol. 6, no. 1, pp. 49–70, 1985.

[3] L. A. Barroso, J. Clidaras, and U. Hölzle, *The datacenter as a computer: an introduction to the design of warehouse-scale machines*, 2nd ed. Morgan and Claypool Publishers, 2013.

[4] R. Boutaba, L. Cheng, and Q. Zhang, "On cloud computational models and the heterogeneity challenge," *Journal of Internet Services and Applications*, vol. 3, no. 1, pp. 77–86, 2012.

[5] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby, "Benchmarks and standards for the evaluation of parallel job schedulers," in *Job Scheduling Strategies for Parallel Processing*, 1999, pp. 67–90.

[6] J. Chase, D. Anderson, P. Thakar, and A. Vahdat, "Managing energy and server resources in hosting centers," in *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[7] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, "Energy efficiency for large-scale MapReduce workloads with significant interactive analysis," in *Proc. European Conf. on Computer Systems (EuroSys)*, Bern, Switzerland, 2012, pp. 43–56.

[8] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial, "No justified complaints: on fair sharing of multiple resources," in *Proc. Innovations in Theoretical Computer Science (ITCS)*, Cambridge, MA, USA, 2012, pp. 68–75.

[9] F. Dong and S. G. Akl, "Scheduling algorithms for grid computing: State of the art and open problems," Queen's University, Tech. Rep. 2006-504, Apr. 2006.

[10] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2014.

[11] D. G. Feitelson and L. Rudolph, "Towards convergence in job schedulers for parallel supercomputers," in *Proc. Int'l Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1996, pp. 1–26.

[12] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proc. European Conf. on Computer Systems (EuroSys)*, Bern, Switzerland, 2012, pp. 99–112.

[13] E. Frachtenberg and D. G. Feitelson, "Pitfalls in parallel job scheduling evaluation," in *Job Scheduling Strategies for Parallel Processing*, 2005, pp. 257–282.

[14] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "AutoScale: dynamic, robust capacity management for multi-tier data centers," *ACM Trans. on Computer Systems*, vol. 30, no. 4, pp. 14:1–14:26, Nov. 2012.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: fair allocation of multiple resource types," in *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2011, pp. 323–326.

[16] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "Evaluation of job-scheduling strategies for grid computing," in *Proc. IEEE/ACM Int'l Workshop on Grid Computing (GRID)*, Bangalore, India, 2000, pp. 191–202.

[17] S. M. Hussain Shah, K. Qureshi, and H. Rasheed, "Optimal job packing, a backfill scheduling optimization for a cluster of workstations," *The Journal of Supercomputing*, vol. 54, no. 3, pp. 381–399, 2010.

[18] B. Jennings and R. Stadler, "Resource management in clouds: survey and research challenges," *Journal of Network and Systems Management*, Mar. 2014.

[19] M. Lin, A. Wierman, L. Andrew, and E. Thereska, "Dynamic right sizing for power-proportional data centers," *Transactions on Networking (TON)*, vol. 21, no. 5, Oct. 2013.

[20] L. Mai, E. Kalyvianaki, and P. Costa, "Exploiting time-malleability in cloud-based batch processing systems," in *Proc. Int'l Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, Farmington, PA, USA, Nov. 2013.

[21] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. Int'l Symp. on Microarchitecture (Micro)*, Porto Alegre, Brazil, 2011, pp. 248–259.

[22] B. D. Martino, J. Dongarra, A. Hoisie, L. T. Yang, and H. Zima, *Engineering the grid: status and perspective*. American Scientific Publishers, 2006.

[23] "Wikipedia: Monte-Carlo methods," http://en.wikipedia.org/wiki/Monte_Carlo_method, 2014.

[24] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," Microsoft Research, Tech. Rep., 2011.

[25] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM Symp. on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[26] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in Google compute clusters," in *Proc. ACM Symp. on Cloud Computing (SoCC)*, Cascais, Portugal, Oct. 2011, pp. 3:1–3:14.

[27] L. Shi, J. Furlong, and R. Wang, "Empirical evaluation of vector bin packing algorithms for energy efficient data centers," in *Proc. IEEE Symp. on Computers and Communications (ISCC)*, 2013, pp. 9–15.

[28] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing," in *USENIX Conference on Power Aware Computing Systems (HotPower)*, San Diego, CA, USA, Dec. 2008.

[29] C. A. Thekkath, J. Wilkes, and E. D. Lazowska, "Techniques for file system simulation," *Software—Practice and Experience*, vol. 24, no. 11, pp. 981–999, Nov. 1994.

[30] A. Verma, G. Dasgupta, P. D. T. Nayak, and R. Kothari, "Server workload analysis for power minimization using consolidation," in *USENIX Annual Technical Conference*, Jun. 2009.

[31] J. Wilkes, "Google cluster trace," http://code.google.com/p/googleclusterdata/, 2011.

[32] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. European Conf. on Computer Systems (EuroSys)*, Apr. 2010, pp. 265–278.

[33] Q. Zhang, M. Zhani, and J. H. R. Boutaba, "HARMONY: dynamic heterogeneity-aware resource provisioning in the cloud," in *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, 2013, pp. 510–519.

[34] J. Zilber, O. Amit, and D. Talby, "What is worth learning from parallel workloads? a user and session based analysis," in *Proc. ACM Int'l Conf. on Supercomputing (ICS)*, 2005, pp. 377–386.