Reducing Execution Waste in Priority Scheduling: a Hybrid Approach

Derya Çavdar
Bogazici University

Lydia Y. Chen
IBM Research Zurich Lab

Fatih Alagöz
Bogazici University

Abstract

Guaranteeing quality for differentiated services while ensuring resource efficiency is an important and yet challenging problem in large computing clusters. Priority scheduling is commonly adopted in production systems to minimize the response time of high-priority workload by means of preempting the execution of low-priority workload when faced with limited resources. As a result, the system performance may not only suffer from the long queueing time of low-priority workload due to resource starvation, but also non-negligible execution waste owing to repetitive evictions. In this paper, we propose a scheduler, HYBRID, which allows the scheduler to switch between being preemptive and non-preemptive by providing a fixed number of computing resources – sticky slots providing uninterruptible task executions. In addition, to preserve performance advantages of high-priority workload by conventional preemptive priority scheduling, HYBRID also aims to reduce repetitive evictions, response times, and wasted executions caused by the low-priority workload. Trace driven simulation analysis shows that our proposed HYBRID scheduler outperforms conventional preemptive priority scheduling by improving response time of low-priority workload by 15% and reducing wasted executions by 85%.

1 Introduction

Priority scheduling has been widely adopted in the production systems as an effective way to provide differentiated services, examples including Google clusters [10] and Facebook [5]. Certain classes of workloads are given higher priority when allocating resources, especially during the time period of insufficient resources. Typically, production systems especially for big data like workloads dimension their resources by fixed number of slots which defines the amount of workload can be severed concurrently on servers. Response times of high-priority class can be minimized by giving sufficient amount of slots whereas the low-priority class may suffer from the resource starvation and being executed without a performance guarantee due to insufficient provisions of slots.

Recent studies [3, 8] show that the priority scheduling not only can severely harm the low-priority class but also can result in significantly high amount of execution waste due to preemption – high-priority tasks can evict low-priority tasks that are in execution. The amount of resources used for executions of evicted tasks are thus wasted, especially when tasks are non-resumable from the eviction points. Particularly, higher number of low-priority tasks are *repetitively* evicted by high-priority tasks which often require large amount of resources, e.g., long CPU execution time. Consequently, low-priority tasks are repetitively resubmitted to the system where their response times are severely degraded. While the prior work on priority scheduling have different performance objectives such as delay [9, 12] and fairness [6, 4], it is still an open question how to improve the resource efficiency and response time of low-priority tasks and meanwhile guarantee the performance of high-priority tasks especially in the case of preemptive priority scheduling.

In this paper, we develop HYBRID scheduler which combines the principles of first-come-first-served and preemptive priority scheduling by considering the priority of tasks, available resources, and task age. Our objectives are to preserve performance benefits of high-priority tasks, to prevent the resource starvation for low-priority tasks and to minimize execution waste due to repetitive task evictions. HYBRID guarantees the high-priority task response time by occasionally allowing the preemption of low-priority tasks and minimizes the execution waste due to repetitive evictions by occasionally enabling uninterruptible executions for low-priority tasks. We particularly focus on slotted systems where the basic computing units are slots which are capable of serving one task at a time. To allow the scheduler

switching between being non-preemptive and preemptive, we introduce a limited number of special type of slots – sticky slots which provide uninterrupted execution for tasks either from low or high-priority class.

In particular, HYBRID applies two scheduling principles on tasks depending on the availability of sticky slots upon their arrival times: (i) *sticky phase* where all tasks are served by the order of task arrival times, i.e., first-arrived-first-served (FCFS), on sticky slots, and (ii) *regular phase* where all tasks are served according to preemptive priority scheduling principle on regular slots, as there is no sticky slot available. During the sticky phase, HYBRID provides scheduling opportunities for tasks that already have long queueing times such as low-priority tasks undergoing multiple evictions. Controlling the number of sticky slots enables HYBRID to agilely change the execution order of tasks according to either task age or task priority. Using trace driven simulations, we show that HYBRID not only provides better response time for low-priority tasks, i.e., roughly 15% by greatly reducing task evictions, but also leads to a noteworthy decrease on wasted resources, i.e., roughly 85% compared to various preemptive priority schedulers.

In terms of contributions, we develop a novel scheduler HYBRID which preserves the advantages of preemptive priority scheduling for high-priority tasks and reduces the wasted executions due to repetitive evictions of low-priority tasks. The trace driven simulation results show that the proposed concept of sticky slots is very effective in supporting the hybrid design of combining two different scheduling principles.

In the following, we first illustrate the system model considered in this study in Section 2. We detail out the HYBRID scheduler in Section 3, followed by a large number of trace-driven simulations.

2 System Model

Our system model aims to capture the characteristics of heterogeneous resources and workloads where tasks are associated with different priorities. The main components considered here are: tasks, the scheduler and servers. The incoming tasks join the scheduling queue immediately after their arrival to the system. The scheduler controls the resource assignment of tasks based on the server availability in terms of number of computational slots and scheduling discipline employed. The slot-based system is widely used in production systems, e.g. [1, 7], due to its simplicity in allocating resources across heterogenous resources and scalability to handle large amount of workload. Our study is particularly based on the published trace of Google computing clusters – where there are four dominant types of different servers as shown in Table 1 – and focused on a scenario of a cluster consisting of 125 servers. Moreover, we also leverage the workload information in terms of tasks provided in Google trace. In the following section, we describe the working principles of the slot-based system and the computation of response time.

2.1 Slot-based System

In the slot system, tasks are assigned based on the availability of predefined number of slots on each server without knowing the actual amount of servers' available resources and task's resource requirements. A slot is defined as a virtual token which is required to execute a task on a server. The number of slots on a machine acts as a limit on the maximum number of tasks that can concurrently run on that server.

The principles of assigning tasks in the slot-based system are as follows. If there is an available slot in the system, the task occupies that slot and immediately starts its execution. Each task is assigned to only one slot independent of its resource usage. The resource allocations are made based on the actual resource usages of tasks. If priority evictions are allowed, exactly one task is evicted for each evictor task since the system is slot-based. In addition to priority evictions, a task may be evicted due to memory overrun. Furthermore, we assume that task execution is non-resumable from the eviction point, meaning that computational results and resources are thus wasted. We refer successful task execution when the task is not terminated by an eviction. To quantify the execution waste, we focus on CPU seconds which are used for unsuccessful execution of a task whose final outcome is either eviction or drop-out.

Server	quantity	CPU	Memory	slots	average CPU per slot
A	69	0.5	0.50	24	normal
B	38	0.5	0.25	24	normal
C	10	0.5	0.75	24	normal
D	5	1.0	1.0	24	powerful
	3	1.0	1.0	48	normal
total	125	-	-	3072	-

Table 1: Server types and slot configurations.

2.1.1 Workload-aware Slot Configuration

The efficiency of the slot-based system highly depends on the slot configuration. Here, we configure slots according to the workload-aware slot configuration algorithm (WASC) which is proposed in the following study [2]. The rationale is to exploit workload heterogeneity and make slot configurations accordingly in order to better fit workload's requirements. In the particular, for the example of Google trace, high-priority

tasks from the production classes are rare but consumes much more resources. Therefore, WASC configures slots with two kinds of average CPU capacity to better handle the high-priority tasks. Particularly, WASC configures server type A-C in Table1 with 24 slots, and server type D with either 24 slots or 48 slots. The type D servers are low in terms of quantity and equipped with higher amount of resources. Therefore, the average CPU capacity capacity at server D can be "normal" or "powerful" depending on the number of slots.

2.2 Response Time Model for Tasks

We compute the execution time of a task (T_x) by the amount of time that requires to accumulate enough CPU consumption satisfying the CPU demand of the task (Δ_C). The CPU consumption of a task is the integral of the assigned CPU rate Γ_c) over time, as $\Delta_C = \int_0^{T_x} \Gamma_c(t)$. Our slot-based system limits tasks to use exactly one slot and run on no more than one core. Moreover, CPU resources of the server is equally shared by concurrently running tasks. Hence, the maximum CPU rate that a task can get is limited by the core capacity and the minimum CPU rate that a task can get is limited by the average CPU capacity per slot: $\frac{C}{N_s} \leq \Gamma_c(t) \leq \frac{C}{N_c}$. The minimum T_x is determined by the number of cores (N_c) while the maximum T_x is limited by the number of slots (N_s) as shown $\frac{\Delta_C N_c}{C} \leq T_x \leq \frac{\Delta_C N_s}{C}$.

We update the assigned CPU rates at each unit time. Since, a task can not use more than one core, CPU rate is adjusted to one core capacity if the number of running tasks is smaller than number of cores on that machine as shown.

$$\Gamma_c(t) = \begin{cases} \frac{C}{N_r(t)} & \text{if } \frac{C}{N_r(t)} \leq \frac{C}{N_c} \\ \frac{C}{N_c} & \text{if } \frac{C}{N_r(t)} > \frac{C}{N_c} \end{cases}$$

$\Gamma_c(t)$ is the assigned CPU rate of a task and $N_r(t)$ is the number of concurrently running tasks on the server at time t. Our response time model also includes the time spent in the queue and the total time spent on wasted executions due to evictions in non-resumable systems.

3 HYBRID Scheduler

In this paper, we propose a novel scheduler HYBRID which effectively assigns tasks to slots for highly heterogeneous workloads to provide guaranteed execution of low-priority tasks as well as giving precedence to high-priority tasks. Figure 1 depicts the design of HYBRID. To such an end, HYBRID combines two types of principles for the scheduling order of tasks: (i) "first" arrival times of tasks and (ii) priority of tasks, depending on

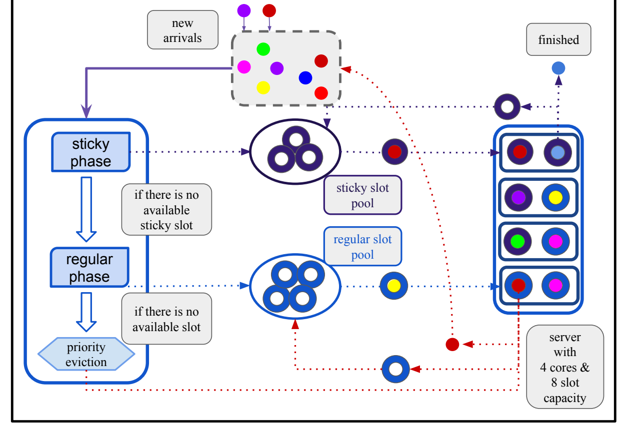


Figure 1: HYBRID scheduler design.

the number of available sticky slots. Sticky slots feature on uninterrupted executions of tasks and the usage of sticky slot is independent of task's priority. We note that the "first" arrival time refers to the time stamp when a task first ever enter the scheduling queue, instead of its consequent re-entering due to evictions if any. Upon task arrivals at the scheduling queue, HYBRID schedules tasks based on the order of their first arrival times, if there are sticky slots available. No tasks can be evicted from a sticky slot. On the contrary, when all sticky slots are occupied, HYBRID schedules tasks which wait in the scheduling queue to regular slots by the order of their priority. As such HYBRID alternates between two phases namely sticky and regular phase. As soon as the sticky slot are all occupied, HYBRID enters the regular phase whereas it switches back to sticky phase upon release of a sticky slot. We note that sticky/regular slots are virtual labels neither holding resources nor reserved on specific machines. The tasks scheduled with regular slots and sticky slots can run on the same machine.

While high-priority tasks can be executed without any interruption, executions of low-priority tasks can switch between being interruptible or uninterruptible based on the state of the scheduler, i.e., sticky or regular phase, respectively. In addition to the advantage of high-priority tasks which are experiencing no eviction resulting into successful execution, sticky phase gives high precedence to low-priority tasks that are already experiencing multiple evictions because of their "early" first arrival time. Overall, the number of sticky slots configured in the system determines the degree that how often low-priority tasks might experience eviction due to insufficient number of slots. In HYBRID, we keep a constant number of sticky slots and the choice of such number is searched empirically. We detail out the scheduling principles in the sticky and regular phase in the following and depict the outline of HYBRID in Figure 1.

3.1 Sticky Phase

At each unit time, HYBRID checks the central scheduling queue. If there are waiting tasks, the tasks are sorted in ascending order by their "first" arrival time to the system. There are two reasons behind this operation. First, the low-priority tasks usually can not get scheduled due to low scheduling precedence over high-priority tasks. Second, repetitively evicted tasks suffer from multiple disadvantages, i.e., high queueing time, wasted executions due to evictions. The evicted tasks thus tend to hold the earliest arrival time to the system [2] among the same class tasks.

At this phase, HYBRID functions as a first come first serve non-preemptive scheduler which ensures the schedulability and uninterrupted execution of tasks. In particular, we use 200 sticky slots (6.5% of all slots) throughout the evaluations in this paper. Such a value is chosen that we can strike a good trade-off between the mitigation of repetitive evictions on low-priority tasks and the potential latency increase on the high-priority tasks. Due to space limitations, we only present results based on the optimal number of sticky slots.

3.2 Regular Phase

HYBRID enters regular phase when there is no available sticky slot. At regular phase, HYBRID works as a fully preemptive priority scheduler. The central scheduling queue is sorted according to tasks' priority in descending order first and then according to the arrival time in ascending order. HYBRID dispatches the task from the head of the queue and schedules the task based on the availability of regular slots. When there is no regular slot available, HYBRID evicts the low-priority task that is not executing on a sticky slot and starts most recently so to schedule a high-priority task that waits in the central scheduling queue.

To better explore the heterogeneity of the underlying hardware and task requirements, HYBRID tries to utilize powerful slots. The average CPU capacity per slot listed in Table 1. Specifically, when scheduling so-called production-class tasks, e.g., priority 9-11 tasks, HYBRID sorts the slots according to their average CPU per slot and assigns tasks to the most powerful slot to satisfy their higher resource demands. For the tasks from non-production classes, they are assigned to a randomly chosen slot. Moreover, when there is no available regular slot upon the arrival of a production class task, HYBRID evicts the task that occupy most powerful slot from the lowest priority class. HYBRID thus ensure the occupancies of such "powerful" slots by the production-class tasks that indeed require higher amount of resources.

4 Evaluation

To evaluate the effectiveness of HYBRID on a heterogenous cluster, we use trace-driven simulations based on Google cluster trace [11]. The workload is composed of tasks from 12 priority classes with diverse resource requirements. Google cluster trace also provides information about server environment in terms of CPU, memory and percentages of server types as well as the workload. All task resource usages and machine resource capacities are normalized with a range between [0,1] according to the machine with maximum capacity. In particular, we evaluate a data center of 125 heterogeneous servers with four dominant types of server shown in Table 1. In our analysis, we use trace-driven 15 hour long simulations with approximately 68.000 tasks in total. The performance metrics of interest are the Wasted Executions (WE) measured in CPU seconds and class response times, particularly the low (class 0) and high-priority ones (class 9). WE is defined as the CPU seconds consumed in executions that are terminated with unsuccessful outcomes, i.e., eviction and drop out.

4.1 Performance of HYBRID

To highlight performance advantages achieved by HYBRID, we compare HYBRID against two types of schedulers: PRI and PRI5. PRI is a standard preemptive priority scheduler which schedules tasks only by their priority class and evicts low-priority tasks to free up slots for high-priority tasks in case of insufficient number of slots. One can also think of PRI as a degenerated version of HYBRID having only regular phase and zero number of sticky slots. Moreover, to mitigate the potential downside of repetitive evictions in PRI, we impose a limit on the number of evictions experienced by tasks. Such a scheduler with limit 5 is called PRI5. When tasks reach eviction limit, they are immediately dropped out by the scheduler. The choice of such a limit is based on empirical evaluations so that the percentage of task drops is under 1% and the resulting maximum number of repetitive eviction per task is similar to HYBRID. By dropping tasks, PRI5 avoids further evictions and hence potential execution waste. Essentially, PRI5 is a reactive approach that bounds the negative impact of evictions whereas HYBRID is a proactive approach to control repetitive evictions through the number of sticky slots. In the following, we present the wasted executions and class response times achieved by HYBRID, PRI and PRI5.

4.1.1 On Wasted Executions

In Table 2, we summarize five metrics related to execution waste: (i) $NE_{priority}$, the total number of priority evictions, (ii) $\overline{pe}$, the average number of evictions per

task that experience at least one eviction, (iii) $\max(pe)$, the maximum number of evictions experienced by tasks, (iv) the number of task drops, and (v) WE, the accumulated wasted CPU seconds. One can expect there exists a positive dependency among WE, the number of evictions and task drops.

metric	HYBRID	PRI	PRI5
$NE_{priority}$	1231	8509	4857
$\max(pe)$	4	14	4
$\overline{pe}$	1.1	2.23	1.72
number of task drops	-	-	400
$WE(10^3)$[cpu.sec]	20.9	141	95

Table 2: Statistics on number of evictions, drops and wasted executions.

As seen from Table 2, HYBRID significantly decreases the total number of priority evictions $NE_{priority}$ by 85% and 75% respectively compared to PRI and PRI5. Moreover, HYBRID scheduler also improves the average number of evictions per evicted task $\overline{pe}$ by 50% and effectively bounds the maximum number of priority evictions per task $\max(pe)$ under 4. Without any surprise, PRI5 can effectively limit the number of maximum evictions experienced by a task, at the expense of 400 task drops. The reduction on the number of evictions and task drops made by HYBRID is directly reflected on wasted executions – a significant improvement of 85 %. Another worth mentioning observation is that, though PRI5 indeed reduces the total number of evictions by 50%, PRI5 only reduces WE by roughly 30%, compared to PRI. The reason behind is that, as PRI5 only reactively drops tasks that already reexperience high number of evictions, i.e., 4, there is still a great amount of wasted executions which are consumed before tasks are dropped. In contrast, HYBRID can successfully minimize the wasted executions by using the sticky slots to proactively modulate the degree of task preemption.

4.1.2 On Response Time

Here, we focus on presenting per-class response times, particularly class 0 (low-priority) and class 9 (high-priority) under HYBRID, PRI, and PRI5. As the low-priority class can experience repetitive evictions, we separate the response times of class 0 into two types: R_{0-ev}, the ones that experience at least one eviction and R_{0-nev}, the ones that never experience any eviction. By comparing these two statistics, we quantify the degradation on the performance of low-priority class due to eviction. Also, one can also view R_{0-ev} as a proxy of response time outlier in the entire system. We summarize R_{0-ev}, R_{0-nev} and the response time of class 9 (R_9) in Figure 2.

Let's first focus on the comparison between HYBRID and PRI. We can see that HYBRID significantly reduces R_{0-ev} by a factor of 2 compared to PRI though there is a slight increase in R_{0-nev}. We attribute the success of HYBRID to the introduction of sticky slots which not only ensures uninterruptible execution but also terminates the loop of repetitive evictions for low-priority tasks. Another significant observation is that, HYBRID also reduces the response time of class 9 by 10% by reducing the extra system load generated by resubmissions due to evictions. Consequently, by allowing the scheduler to switch between scheduling principles of non-preemptive FCFS and preemptive priority, HYBRID not only greatly reduces the outliers from low-priority class, i.e., R_{0-nev}, but also improves the response time of high-priority tasks.

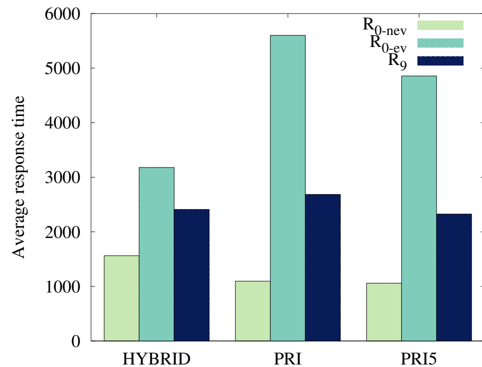


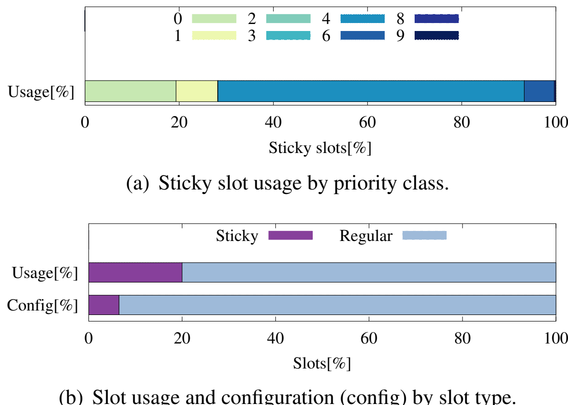
Figure 2: Response time of class 0 and 9 under HYBRID, PRI and PRI5.

In terms of performance of PRI5, we can see that PRI5 indeed improves R_{0-ev} and R_9 slightly without incurring higher R_{0-nev} compared to PRI. Similar to HYBRID, PRI5 reduces the system load by terminating evictions and dropping tasks. Consequently, R_9 is slightly lower. However, due to the reactive nature of PRI5, response time outliers in low-priority class persist. Nonetheless, task dropping in PRI5 may hurt user experiences. Overall, HYBRID successfully eliminates response time outliers in the low-priority and provides more uniform response time among same priority class, while incurring minimum amount of wasted executions due to evictions.

4.1.3 Usage of Sticky Slots

As the effectiveness of HYBRID lies at adopting sticky slots, we provide basic statistics on how HYBRID uses sticky slots on different priorities in Figure 3. First of all, the usage of sticky slots fluctuates with the nature of the workload, i.e., the distribution of priorities. In terms of class usages, more than 90% of the sticky slots are exploited for the successful execution of class 4 and lower priorities as shown in Figure 3(a). The major users of

sticky slots are class 0 and class 4 which are dominant in population.



(a) Sticky slot usage by priority class.

(b) Slot usage and configuration (config) by slot type.

Figure 3: Percentage of configuration of slot types and usage of slot types by tasks.

In terms of overall task usages, 20% of finished tasks use sticky slots for their successful execution as shown in Figure 3(b). Even though only 6.5% of the total number of slots are configured as sticky ones, the usage rate of sticky slots by overall finished tasks is significantly higher. There are two reasons behind. First the order of use of slot types where sticky phase has precedence. Second, sticky slots are mainly utilized by short lived low-priority tasks. Hence, the resource usage associated with the tasks executing on sticky slots is higher than expected. Therefore, we can conclude HYBRID successfully handles disadvantages of low-priority tasks by extensively using sticky slots to provide opportunity for their uninterrupted execution.

5 Conclusion

Motivated by non-negligible execution waste and repetitive evictions in priority scheduling systems, we propose HYBRID scheduler which aims to provide scheduling and execution guarantees for low-priority tasks while preserving the performance of high-priority tasks. HYBRID introduces a novel design where the availability of sticky slots decides the scheduling principles and corresponding phase of the scheduler, i.e., non-preemptive FCFS in the sticky phase and preemptive priority in the regular phase. Using on-production Google cluster traces, our simulation results show that HYBRID is able to achieve lower response time for low-priority tasks as well as a significant reduction on wasted executions compared to various preemptive priority schedulers. Under HYBRID, sticky slots not only improve the scheduling opportunity of low-priority tasks that already experience multiple evictions but also ensure their uninterrupted and successful executions. Overall, HYBRID significantly reduces wasted executions and improves response time of low-priority tasks by mitigating priority evictions as well as repetitive evictions.

6 Acknowledgments

This work has been supported supported by the Swiss National Science Foundation (200021_141002), EU commission FP7 GENiC project (608826) and DPT TAM project (07K120610).

References

[1] Apache hadoop. `http://hadoop.apache.org/`, 2014.

[2] ÇAVDAR, D., CHEN, L. Y., AND ALAGOZ, F. Priority scheduling for heterogeneous workloads: Tradeoff between evictions and response time. *IEEE Systems Journal PP*, 99 (2015), 1–12.

[3] ÇAVDAR, D., ROSÀ, A., CHEN, L., BINDER, W., AND ALAGOZ, F. Quantifying the brown side of priority schedulers: Lessons from big clusters. *SIGMETRICS Performance Evaluation Review 42*, 3 (2014), 76–81.

[4] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI* (2011), pp. 323–336.

[5] GODER, A., SPIRIDONOV, A., AND WANG, Y. Bistro: Scheduling data-parallel jobs against live production systems. In *USENIX ATC* (2015), pp. 459–471.

[6] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *USENIX NSDI* (2011), pp. 295–308.

[7] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX ATC* (2015), pp. 485–497.

[8] ROSA, A., CHEN, L., AND BINDER, W. Understanding the dark side of big data clusters: An analysis beyond failures. In *DSN* (2015), pp. 207–218.

[9] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys* (2013), pp. 351–364.

[10] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *EuroSys* (2015), pp. 1–17.

[11] WILKES, J. More Google cluster data. `https://github.com/google/cluster-data`, 2011.

[12] ZHANG, Q., ZHANI, M., BOUTABA, R., AND HELLERSTEIN, J. Dynamic heterogeneity-aware resource provisioning in the cloud. *IEEE Transactions on Cloud Computing 2*, 1 (2014), 14–28.