CurTail: Distributed Cotask Scheduling with Guaranteed Tail-Latency SLO

Zhijun Wang
The University of Texas at Arlington
Arlington, USA
email: zhijun.wang@uta.edu

Hao Che
The University of Texas at Arlington
Arlington, USA
email: hche@uta.edu

Hong Jiang
The University of Texas at Arlington
Arlington, USA
email: hong.jiang@uta.edu

***Abstract*—Today's user-facing interactive datacenter services, such as web searching and social networking, have to meet stringent tail latency Service Level Objectives (SLOs). Unfortunately, due to the scale-out nature of the workloads, how to enable both tail-latency-SLO guarantee for such services and high resource utilization remains a critical challenge. In this paper, we propose a distributed Cotask scheduler with guaranteed Tail-latency SLO (CurTail), aiming at providing both job tail-latency-SLO guarantee and high resource utilization. CurTail is a top-down, holistic approach. It decouples an upper job-level design from a lower task-level design. At the job level, a decomposition technique is proposed to translate a given job tail-latency SLO into task-level performance budgets for tasks in a cotask, i.e., the collection of tasks spawned by a job. At the task level, the task budgets are translated into both task compute and networking resource demands, hence allowing for distributed cotask scheduling. The preliminary testing results based on simulation indicate that CurTail can indeed provide job tail-latency SLO guarantee at high resource utilization.**

***Index Terms*—Tail latency SLO guarantee, cotask scheduling, datacenter.**

I. Introduction

To date, datacenter service providers generally overprovision datacenter resources to provide high assurance of meeting Service Level Objectives (SLOs) for datacenter services, e.g., stringent tail-latency SLOs for user-facing interactive services. For instance, aggregate CPU and memory utilizations in a 12,000-server Google cluster are mostly less than 20% and 40%, respectively [1]. As datacenters are approaching their capacity limits, in terms of, e.g., computing capacity and power budget [2], *how to improve datacenter resource utilization while providing SLO guaranteed services* becomes an important design objective for job scheduling, called *the objective* for short hereafter. To this end, however, one must successfully tackle two key challenges.

The first challenge is how to translate job-level SLOs into precise runtime system resource demands at the task level. Today's user-facing interactive services are predominantly scale-out by design. Namely, a job may spawn a large number of tasks (the exact number is called the job fanout degree), collectively known as a cotask [3], to be dispatched to, queued at, processed by workers and the resulting data flows, collectively known as a coflow [4]–[10], returned from a large number of servers. The job response time is determined by the time the resulting data of the slowest task is returned and hence, is a strong function of job fanout degree. Clearly, a job scheduler must know the exact cotask/coflow resource demands, so that the right amounts of compute and networking resources can be allocated to achieve the objective.

The second challenge is that the objective calls for joint compute and networking resource allocation. With interleaved task dispatching, task computing, and resulting data returning per job execution, it becomes clear that compute and networking resources must be jointly allocated to be effective.

The existing solutions are simply not up to the above challenges. First, largely due to the lack of a means to do the translation, the existing cotask-aware (e.g., [3], [9], [11], [12]) and coflow-aware job scheduling solutions (e.g., [4]–[6]) are centralized by design and hence not scalable, and exclusively focused on average performance targets, e.g., minimizing average job/coflow completion time, rather than meeting job-tail-latency SLOs.

Second, most existing job scheduling and resource provisioning solutions are point by design, concerned with either compute (e.g. [13]–[17]) or networking (e.g., [6], [7], [18], [19]) aspects of resource provisioning, rather than both jointly. This makes it difficult for the existing solutions to achieve the objective.

Third, the existing tail-latency-aware job scheduling solutions (e.g., [20]–[24]) are exclusively focused on storage applications and jobs with fanout degree of one only. Some solutions focusing on outlier alleviation have been developed to shorten the job tail latency. For example, several solutions of task-size-aware task reordering in a task queue have been proposed [25]–[27] to avoid head-of-line blocking of small-sized tasks by large-sized ones to reduce the overall task tail latency. CPU power control schemes [28], [29] have been designed to Dynamically adjust Voltage and Frequency Scaling (DVFS) for task servers based on task execution time to save energy and maintain low task tail latency. However, the approaches taken by such solutions cannot be applied to jobs with job fanout degrees larger than one, as the resource demands for tasks belonging to jobs with different fanout degrees are different.

To achieve the objective of high resource utilization while providing tail latency guarantee, this paper proposes a distributed **C**otask scheduler with g**u**a**r**anteed **Tail**-latency SLO (CurTail). CurTail is a top-down, holistic approach. It decou-

 ISBN: 978-1-61208-853-2

ples the upper job-level design from the lower task-level or runtime-system design. At the job level, by leveraging a prior work [30], [31], we propose a decomposition technique that translates a given job tail-latency SLO into a task performance budget shared by all the tasks in the cotask of a job. This effectively decomposes a complex job-level cotask/coflow resource allocation problem into individual task/flow resource allocation subproblems at the task level. The design at this level is independent of the underlying runtime systems to be used and, hence, is portable to any datacenter platforms.

At the task or runtime system level, the task performance budgets are first translated into task compute and networking resource demands. Then, the proposed distributed task and flow schedulers allocate the resources to match the resource demands, hence, achieving the objective. The major contributions of the paper are:

- CurTail is a top-down approach, it decouples an upper job-level design from a lower task-level design and is independent on underlying systems, and hence it can be easily implemented;
- Curtail jointly allocates compute and networking resources based on the task resource demand to meet tail latency SLO, and hence can greatly improve system resource utilization.

The preliminary testing results based on datacenter simulation demonstrates that CurTail can indeed provide tail-latency-SLO guarantee and high resource utilization.

The rest of paper is structured as follows. Section II gives the detailed descriptions of CurTail and Section III presents the performance evaluation of CurTail, finally Section IV concludes the paper.

II. CurTail

CurTail schedules cotasks in two distinct logical steps. First, at the job level, the job tail-latency SLO for a given service is translated into task response time budgets for all the tasks a job spawns. Second, at the task level, the right amount of compute and networking resources are allocated to individual tasks that meet the task budgets, hence, achieving the objective. In what follows, we first give an overview of CurTail and then discuss the two steps, separately.

A. CurTail Overview

With reference to Figure 1, CurTail works as follows. When a job scheduler in a master receives a job with a given tail-latency SLO and a given fanout degree, K, it translates the tail-latency SLO into a task response time budget shared by all the tasks the job spawns. As long as the task response time, i.e., the sum of the task dispatch time, task queuing time, task compute time, and flow completion time for every flow in the task coflow is within the task budget for the tasks belonging to the job, the tail-latency SLO for that job is guaranteed to be met. This effectively decouples a complex job-level resource allocation problem into distributed task-level subproblems at individual workers the tasks are mapped to. The design at this

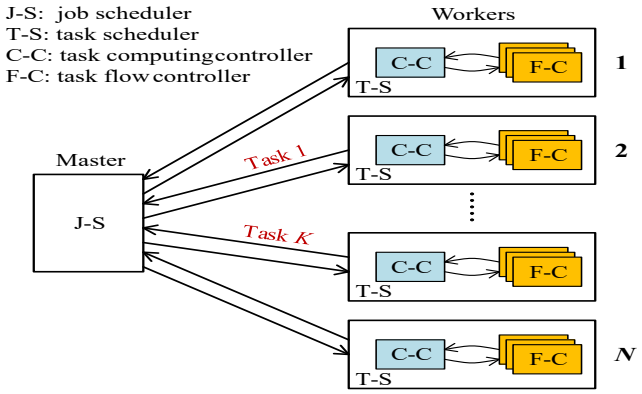

Fig. 1. A job scheduler, J-S, runs in a master node and distributed task schedulers, T-S, run in individual workers in the cluster, each of which is mainly composed of a compute controller, C-C, and a flow scheduler, F-C, per flow emitted from the worker.

TABLE I
THE 99-TH PERCENTILE TAIL-LATENCY VS. JOB FANOUT DEGREE

Fanout degree	1	10	100	1000
Tail-latency (ms)	23.03	34.51	46.03	57.54

level is runtime system independent and hence, can be precomputed offline for jobs of different fanout degrees and is portable to any datacenter platforms.

Then, the tasks, together with the task budgets, are dispatched to K workers in a cluster to be processed. CurTail does not dictate what task dispatching algorithm should be used, and hence, can work with any task dispatching algorithms. It focuses on the task compute and task flow resource allocation at individual workers. Upon the arrival of a task at a worker, based on the budget, the task scheduler, T-S, in the worker, then sets parameters in a task compute controller, C-C, and a flow controller, F-C. Collectively, these controllers ensure that datacenter compute and networking resources are allocated with high precision to meet task budgets, and hence, SLOs for individual jobs at high resource utilization.

B. Job-Tail-Latency-to-Task-Budget Translation

We leverage an existing solution for a general black-box Fork-Join model (i.e., with all the fork nodes treated as black boxes) [30], [31]. This solution states that the pth-percentile job tail-latency x_p can be approximately expressed in terms of the mean (E) and variance (V) of task response time, and job fanout degree (K), i.e., $x_p \approx x_p(K, E, V)$ (see [30], [31] for details). This approximation is found to be sufficiently accurate at high load (e.g., 80% or above) for a wide range of Fork-Join structures of practical interests. In particular, with tail cutting [32], [33], or equivalently, light-tailed task execution time distribution, the solution is found to be accurate even at low load. Since tail-cutting techniques have been widely deployed to combat stragglers in practice [32], we assume that the approximation is accurate at any load level in practice.

 ISBN: 978-1-61208-853-2

In what follows, we make three important observations with regard to this approximation.

First, we observe that $x_p \approx x_p(K, E, V)$ is an increasing function of K. As K increases, the slowest task in the cotask of a job is likely to become slower, resulting in a longer job tail-latency. For example, assume that $E = 5$ ms and $V = 25$ ms^2, the 99th-percentile (i.e., $p = 99$) tail-latency for jobs with different fanout degrees are listed in Table I. As one can see, it increases from about 23 ms to 35 ms and then to 59 ms as job fanout degree increases from 1 to 10 and then to 1000. These results also indicate that the job tail-latency is a strong function of job fanout degree.

Second, we note that $x_p \approx x_p(K, E, V)$ must also be an increasing function of both E and V, simply because the slowest task gets slower as E and/or V increases and hence, the job tail-latency also increases. As a result, E and V can be viewed as *task performance budgets* in order to meet x_p. Specifically, to ensure that the job tail-latency is no larger than x_p, the means and variances of the task response time for all the tasks in cotasks must not exceed E and V, respectively. Moreover, we observe that to keep x_p unchanged as job fanout degree K increases, either E or V or both must be reduced. The implication of this observation is significant. It means that to meet the job tail-latency SLO for a given service in terms of x_p, the task budgets, and hence, per-task resource demand, for jobs with different fanout degrees are different.

Third, we note that the translation of a given job tail-latency x_p to the task budgets is one-to-many, admitting virtually unlimited number of $\{E, V\}$ budget pairs. This is because the change of x_p due to the increase (decrease) of E can be offset by properly decreasing (increasing) V and vice versa. In other words, a smaller budget in terms of V allows a bigger budget in terms of E, and vice versa. One may narrow down to one pair of task budgets by letting $\sqrt{V} = \alpha E$, where α is a tunable parameter, which as we shall discuss in more detail, may be estimated based on measurement. Now by taking the inverse of $x_p \approx x_p(K, E, V) \equiv x_p(K, E, \alpha)$, we have, $E \approx E(K, \alpha, p, x_p)$, the task budget in terms of the task mean response time. It means that for a service with tail-latency SLO in terms of x_p and α that guarantees that V will be met, as long as a job scheduler can ensure that the mean task response time for all the tasks is within $E(K, \alpha, p, x_p)$, the job tail-latency SLO is guaranteed.

As an example, for a service with the job tail-latency SLO in terms of $x_{99} = 100$ ms, Figure 2 (a) gives the mean task response time budgets at various α values for jobs with fanout degrees 1, 10 and 100, respectively. We can see that the mean task budget is sensitive to α. When α increases from 0.4 to 2.0, the mean task budget decreases from about 44, 33, and 27 ms to 10, 6, and 4 ms for jobs with fanout degree 1, 10 and 100, respectively.

Figure 2 (b) gives the mean task response time budget versus fanout degree at $\alpha = 0.5$, 1 and 2, respectively. The budget is sensitive to job fanout degree when it is small (from 1 to 10), and then become less sensitive as it further increases. The budget decreases from about 38, 22, and 10 ms to 28, 15, and 6 ms as the fanout degree increases from 1 to 9 at $\alpha = 0.5$, 1 and 2, respectively. It confirms that the jobs with different fanout degrees may have quite different task budgets. In other words, the task resource demands for tasks from jobs with different fanout degrees can be significantly different.

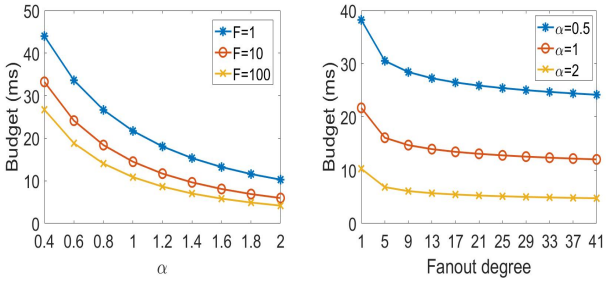

Fig. 2. Mean task response time budget: (a) varying with α; and (b) varying with job fanout degree.

C. Task Resource Allocation

This step is runtime system dependent, and hence, must be carried out at individual workers at runtime. We assume that the task dispatch overhead is small and can be treated as a small fixed task delay ΔE, which is assumed to have been deducted from E already. Namely, E now represents the mean task response time budget for task queuing, compute and task result return. Consider a service with the tail-latency SLO defined by x_p and task budget $E(K, \alpha, p, x_p)$ for jobs of fanout degree K. Further, consider task k in the jth job mapped to worker q, and denote $r^c_{j,k}$ and $r^f_{j,k}$ as the task compute rate and task flow rate at worker q, respectively. Then, we let,

$$r^c_{j,k} \geq \Lambda^c_{K,q}, \quad r^f_{j,k} \geq \Lambda^f_{K,q}, \quad j \in S(K), \tag{1}$$

where $S(K)$ is the set of all jobs with fanout degree K, and $\Lambda^c_{K,q}$ and $\Lambda^f_{K,q}$ are the minimum compute rate and flow rate constraints for any task in job j mapped to worker q, which in turn, satisfy the following inequalities,

$$\frac{W^c}{\Lambda^c_{K,q}} + \frac{W^f}{\Lambda^f_{K,q}} \leq E(K, \alpha, p, x_p), \tag{2}$$

where W^c and W^f are the predicted mean task compute workload size and mean flow size for all tasks in that service.

Clearly, as long as both the compute rate, $r^c_{j,k}$, and flow rate, $r^f_{j,k}$, satisfy the above inequalities for all K tasks in the cotask of a job, the job is guaranteed to meet its tail-latency SLO. Note that this guarantee is on a per job basis and hence, works even when different jobs have different tail-latency SLOs, i.e., different jobs are assigned different x_p and p pairs.

Now, the questions remain to be answered are how to estimate α, or equivalently, V, and how to determine and allocate the minimum compute rate, $\Lambda^c_{K,q}$, and the minimum flow rate, $\Lambda^f_{K,q}$. Assume that the task compute response time and the associated flow completion time are independent random variables. Then $V = V_c + V_f$ [34], where V_c and V_f are the budgets for the variances of the task compute time

 ISBN: 978-1-61208-853-2

and flow completion time, respectively. In what follows, we deal with flow related parameters, $\Lambda^f_{K,q}$ and V_f, separately from compute related parameters, $\Lambda^c_{K,q}$ and V_c. In CurTail, the network condition can be measured through packet round trip time and the unloaded job tail latency can be computed based on the task response time by excluding the queuing time, and hence its overheads are light.

Estimating $\Lambda^f_{K,q}$ and V_f: Since the datacenter network resources are shared by all the flows, one has limited control over these two parameters. So in CurTail, these parameters are, to a large extent, obtained based on measurement. CurTail applies Minimum Rate Guarantee (MRG) [35], a soft minimum flow rate guaranteed congestion control protocol, to maintain an average flow completion time with high probability, which serves as the flow controller, F-C, in Figure 1. Based on the measurement of the network condition, a proper minimum rate $\Lambda^c_{K,q}$ that can be sustained with high probability is set in MRG. Likewise, V_f will simply be set at the measured variance of the flow completion time.

Estimating $\Lambda^c_{K,q}$ and V_c: Again, V_c is simply set at the measured variance of the task compute time. Together with the estimated V_f above, V is set at V_c+V_f, with possibly a small extra margin to guard against the measurement errors. With given V, K, and x_p, α or $E = E(K, p, x_p, \alpha)$ is then uniquely determined. From Eq. (2), we have,

$$\frac{W^c}{\Lambda^c_{K,q}} \leq E(K, \alpha, p, x_p) - \frac{W^f}{\Lambda^f_{K,q}} \equiv E_c(K, \alpha, p, x_p), \qquad (3)$$

where $E_c(K, \alpha, p, x_p)$ is the task compute budget. While $\Lambda^f_{K,q}$ and W^f are well defined and measurable, $\Lambda^c_{K,q}$ and W^c are not easy to quantify and measure. Hence, in the CurTail design, instead of attempting to directly estimate the minimum compute rate, $\Lambda^c_{K,q}$, we focus directly on how to allocate the task compute resource to meet the compute budget, $E_c(K, \alpha, p, x_p)$.

Let T^c_q be the unloaded mean task compute response time for a service at worker q. By "unloaded", we mean that the entire compute resource in a worker is allocated to the task without resource contention at worker q. We assume that T^c_q can be acquired by measurement. Clearly, we must have, $T^c_q \leq E_c(K, \alpha, p, x_p)$, otherwise, the compute budget cannot be met, even with the entire worker compute resource allocated to the task. In each worker, tasks are scheduled using a time-sharing scheduler, i.e., the compute controller, C-C, in Figure 1. To meet the compute response time budget, the scheduler at a worker allocates at least $p^c_{K,q} = T^c_q/E_c(K, \alpha, p, x_p)$ percent of the total compute resource to the task from that service. If all the workers have equal compute resource, then $T^c_q = T^c$ and $p^c_{K,q} = p^c_K$, $\forall q$.

III. Simulation Testing

Simulation setup: Consider two datacenter services, one with a tail-latency SLO, whose jobs are called T-jobs, and the other without SLO requirement, whose jobs are called B-jobs. We treat B-jobs as a single best-effort job and all the B-tasks, i.e., the tasks from B-jobs, share a First-in-Fisrt-Out (FIFO) queue at each worker, handled by a single thread. Each T-task, i.e., a task from a T-job, arriving at a worker, is handled by a new thread in the worker. CurTail allocates all the additional compute resources to the B-tasks, provided that p^c_K percent of the total compute resource is allocated to a T-task. In the absence of a B-task at a worker, the additional worker resources are then equally shared by T-tasks. Obviously, so long as the sum of p^c_K's for all the T-tasks mapped to the worker is less than one, the task compute budgets for all the T-tasks can be met. The additional compute resource, if available, will then be dedicated to the first B-task from the B-task FIFO queue.

The task processing time is time sliced with the slice size set at 1 ms. Each T-task is serviced for multiple time slices on average before context switching. If the execution time of a task is less than 1 ms, it can be switched out before the end of the time slice. This ensures that a context switching will consume no more than a few percent of the CPU resource, as each context switching takes about 50 μs to finish [36], [37].

As aforementioned, CurTail does not specify how tasks should be dispatched. To avoid the possible bias as a result of the use of a specific task dispatching algorithm, we consider two extreme task dispatching algorithms, one with the global information and the other with no information at all. For the former one, when a job arrives at a job scheduler, the tasks of the job are distributed to different workers which have the least numbers of tasks. For the latter one, upon a job arrival at a job scheduler, the tasks for the job are randomly distributed to workers. We test the performance of CurTail against a baseline case where each worker runs two strict priority FIFO queues. The high/low priority queue stores T-tasks/B-Tasks. The tasks from low priority queue cannot be served unless the high priority queue is empty.

Consider a datacenter with a 5x5 leaf-spine network topology and with each rack having 80 hosts (4 job schedulers and 76 workers). The bandwidth/propagation delay is set at 10Gbps/10μs between a host and a leaf node and 10Gbps/20μs between a leaf node and a spine node. While individual jobs in B-job are large jobs in terms of task execution time, representing background batch applications, T-jobs are small jobs, representing user-facing interactive applications. About equal numbers of the two types of jobs are generated, which arrive following a Poisson process. The task execution time for each task follows an exponential distribution with averages of 40 ms and 5 ms for the B-jobs and T-jobs, respectively.

Some background flows running among randomly selected worker pairs are also generated. The flow arrival process is a Poisson process and the flow arrival rate is adjusted so that the total network traffic load is the same as the compute load. This ensures that both compute and network resources will be simultaneously stressed as the load increases.

We further assume that each T-task or B-task execution generates a single task flow with flow size randomly selected from 1K to 500 Kbytes, resulting in an average size, $W^f = 250$

 ISBN: 978-1-61208-853-2

TABLE II
THE COMPUTE BUDGETS, E_c, AND THE PERCENTAGES OF COMPUTE RESOURCE, p_K^c, IN A WORKER HAVE TO BE ALLOCATED FOR A T-TASK FROM A JOB WITH FANOUT DEGREE K.

Fanout degree (K)	5	10	15	20	25
E_c(ms) (load≤80%)	13.1	11.5	10.7	10.2	9.8
E_c(ms) (load 90%)	12.1	10.5	9.7	9.2	8.8
p_K^c(%) (load ≤ 80%)	38.2	43.5	46.7	49.0	51.0
p_K^c(%) (load 90%)	41.3	47.6	51.5	54.3	56.8

Kbytes for T-tasks. The flow rate $\Lambda_{K,q}^f$ that can be guaranteed is found to be 1 and 0.75 Gbps, $\forall q$, at the network loads of 80% or less and 90%, respectively. Then, the corresponding mean flow budget for T-tasks is set as 3 ms and 4 ms (including about ΔE=1 ms for task dispatch time) for load 80% or less and load 90%, respectively.

In this study, the fanout degrees for B-jobs and T-jobs are randomly selected from 1 to 50 and from the set of values (5, 10, 15, 20, 25), respectively. All the T-jobs share the same tail-latency SLO, with the 99th-percentile tail-latency set at 100 ms. By considering both the measured variances of the task flow and task compute times, $\alpha = 1$ is found to allow the variance budget guarantee. Then, we have the following task response time budgets (derived from [30]): 16.1, 14.5, 13.7, 13.2 and 12.8 ms for T-jobs with fanout degrees 5, 10, 15, 20 and 25, respectively. Finally, the compute budgets and the percentages of compute resource that have to be allocated in a worker (as the unloaded mean task compute time is 5 ms) are computed and listed in Table II.

Simulation results and analysis: First, we consider the case with global information. The 99th percentile tail-latency for T-jobs and the average job response times for both job types are used as the performance metrics. Figure 3 depicts the testing results. We see that for the baseline solution, the 99th-percentile tail-latency increases from about 150 ms to more than 250 ms when both compute and network loads increase from 50% to 90%, simultaneously. Note that all the tail latencies presented are measured for all T-jobs as a whole. Due to the lack of space, the tail-latencies for jobs of individual fanout degrees which are also found to meet the tail-latency SLO, are not given. The results indicate that a scheduler even with global information and using strict priority queuing cannot provide job tail-latency guarantee at medium and high loads (50% or higher), as in-service B-tasks can still block T-tasks from getting the worker resources. This explains why to date, the datacenters have to run at 20% to 50% resource utilization, in order to meet stringent tail-latency SLOs. On the other hand, for CurTail, the tail-latency increases from about 45 ms to about 80 ms only, below the tail-latency SLO of 100 ms. The

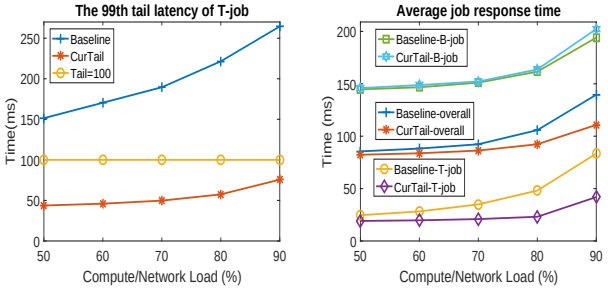

Fig. 3. Task dispatching with global information

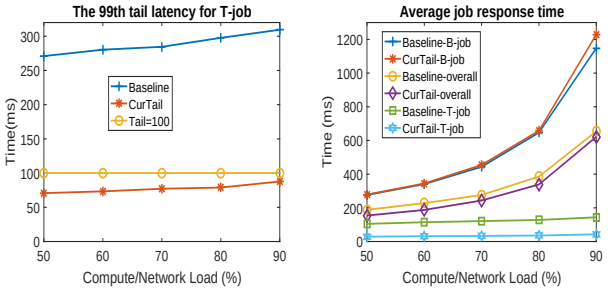

Fig. 4. Random task dispatching

reason why CurTail can perform better than that required by the tail-latency SLO even at high load is that with global information for task dispatching, T-tasks still have good chances to find low load workers to map to, even though the overall load is high. As a result, they are still able to garner some additional compute and network resources to further push the tail-latency lower than the required SLO.

For CurTail, with the needed resources allocated to T-jobs with high precision, the average T-job response time increases from about 20 ms to 40 ms when the load changes from 50% to 90%. As a result, the overall average job response time for T-jobs also improves by about 50% at high load compared to the baseline solution. Meanwhile, CurTail offers almost the same performance for B-job as the baseline solution even at high load. CurTail achieves up to 20% better performance for overall job response time at high load.

Now, consider the case with random task dispatching. In this case, the task placement is much less balanced than the previous case, resulting in much longer tail-latency and average job response time. Figure 4 gives the testing results. As we can see, despite the less balanced load, CurTail still meets the tail-latency SLO, reaching about 90 ms at 90% load, close to the tail-latency SLO, whereas the tail-latency for the baseline solution reaches more than 300 ms. Due to the randomness of task placement, T-tasks have much less chance to find workers with low load, making it less likely to be able to garner much additional resource at high load. This clearly demonstrates the importance of using a solution like CurTail to meet the stringent tail-latency SLO, especially at high resource utilization. Similarly, CurTail performs about 70% better and has slightly better performance for average

 ISBN: 978-1-61208-853-2

T-job response time and overall job response time over the baseline solution, respectively, at the cost of less than 10% increase of the average B-job response time.

The above results strongly suggest that CurTail indeed provides tail-latency guarantee and high resource utilization for single Fork-Join job structure. An open issue is how to extend CurTail to deal with jobs with multi-stage Fork-Join structures or even a general Directed Acyclic Graph (DAG) workflow.

IV. Conclusions

In this paper, we propose CurTail, a distributed **C**otask scheduler with g**u**a**r**anteed **Tail**-latency SLO (CurTail) aiming at providing job tail-latency-SLO guarantee and high resource utilization. CurTail is a top-down and holistic approach. It decouples an upper job-level design from a lower task-level design. At the job level, a decomposition technique is proposed to translate a tail-latency SLO to a task performance budget at the task level, which is runtime system independent and hence, portable to any datacenter platforms. In turn, the task performance budget is further translated into task resource demands at individual workers the tasks are mapped to. A distributed budget-aware cotask scheduling is developed. The preliminary testing results based on datacenter simulation indicate that CurTail can indeed provide job tail-latency SLO guarantee at high resource utilization.

Acknowledgment

This work was supported by the US NSF under Grant No. CCF XPS-1629625, CCF SHF-1704504 and CCF SHF-2008835.

References

[1] C. Reiss, A. Tumanov, G. Ganger, R. Katz and M. Kozuch, "Heterogeneity and dynamicity of clouds at scale," Proceedings of the ACM SoCC, 2012.

[2] L. Barroso, J. Clidaras and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines," Synthesis Lectures on Computer Architecture, vol. 8, no3, pp.1–154, 2013.

[3] Y. Zhao, S. Luo, Y. Wang and S. Wang, "Cotask scheduling in cloud computing," Proceedings of IEEE ICNP, 2017.

[4] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: Near-Optimal Network Design for Coflows," Proceedings of ACM SIGCOMM, 2018.

[5] S. Ahmadi, S. Khuller, M. Purohit and S. Yang, "On scheduling coflows,", Proceedings of MOS IPCO, 2017.

[6] M. Chowdhury and I. Stoica, Ion, "Efficient coflow scheduling without prior knowledge," Proceedings of ACM SIGCOMM, 2015.

[7] M. Chowdhury, Y. Zhong and I. Stoica, "Efficient coflow scheduling with varys," Proceedings of ACM SIGCOMM, 2014.

[8] Z. Qiu, C. Stein and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," Proceedings of ACM SPAA, 2015.

[9] F. R. Dogar, T. Karagiannis, H. Ballani and A. Rowstron, "Decentralized task-aware scheduling for data center networks," Proceedings of ACM SIGCOMM, 2014.

[10] M. Choedhury, and I. Stoica, "Coflow: A networking abstraction for cluster applications," Proceedings of ACM HotNets, 2012.

[11] B. Tian, C. Tian, H. Dai and B. Wang, "Scheduling coflow of multi-stage jobs to minimize the total weighted job completion time," Proceedings of IEEE INFOCOM, 2018.

[12] A. Munir, T. He, R. Raghavendra, F. Le and A. Liu, "Network Scheduling Aware Task Placement in Datacenters," Proceedings of ACM CoNEXT, 2016.

[13] Y. Xu, Z. Musgrave, B. Noble and M. Bailey, "Bobtail: Avoiding Long Tails in the Cloud," Proceedings of the USENIX NSDI, 2013.

[14] A. Ferguson, P. Bodik, E. Boutin and R. Fonseca, "Jockey : Guaranteed Job Latency in Data Parallel Clusters," Proceedings of the 7th ACM EuroSys, 2012.

[15] V. Vavilapalli et al., "Apache Hadoop YARN: Yet Another Resource Negotiator," Proceedings of the ACM Annual Symposium on Cloud Computing (SoCC), 2013.

[16] B. Hindman et al., "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," Proceedings of the USENIX NSDI, 2011.

[17] Z. Wang et al., "Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler," Proceedings of the ACM SoCC, 2019.

[18] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," Proceedings of the ACM SIGCOMM, 2011.

[19] C. Wilson, H. Ballani, T. Karagiannis and A. Rowstron, "Better Never than Late: Meeting Deadlines in Datacenter Networks," Proceedings of ACM SIGCOMM, 2011.

[20] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz and I. Stoica, "Cake: Enabling High-level SLOs on Shared Storage Systems," Proceedings of the ACM SoCC, 2012.

[21] N. Li, H. Jiang, D. Feng and S. Zhang, "PSLO: enforcing the Xth percentile latency and throughput SLOs for consolidated VM storage," Proceeding of EuroSys, 2016.

[22] T. Zhu, A. Tumanov, M. Kozuch, M. Harchol-Balter and R. Ganger, "PriorityMeister: Tail Latency QoS for Shared Networked Storage,", Proceedings of the ACM SoCC, 2014.

[23] T. Zhu, D. Berger and M. Harchol-Balter, "SNC-Meister: Admitting More Tenants with Tail Latency SLOs," Proceedings of the ACM SoCC, 2016.

[24] T. Zhu, M. Timothy and M. Harchol-Balter, "WorloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees," Proceedings of the ACM SoCC, 2017.

[25] J. Li, N. Sharma, D. Ports and S. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," Proceedings of ACM Symposium on Cloud COmputing (SoCC), 2014.

[26] P. Misra, M. Borge, I. Goiri, A.Lebeck, W. Zwaenepoel and R. Bianchini, "Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints," Proceedings of Eurosys, 2019.

[27] G. Prekas, M. Kogias and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks," Proceedings of ACM SOSP, 2017.

[28] M. E. Haque, Y. He, S. Elnikety, T.D. Nguyen, R. Bianchini and K. McKinley, "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," Proceedings of Annual IEEE/ACM MICRO), 2017.

[29] S. Kanev, K. Hazelwood, G. Wei and D. Brooks, "Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications," Proceedings of IEEE International Workshop/Symposium on Workload Characterization, 2014.

[30] M. Nguyen, S. Alesawi, N. Li, H. Che and H. Jiang, "ForkTail: A Black-box Fork-Join Tail Latency Prediction Model for User-facing Datacenter Workloads," Proceedings of HPDC, 2018.

[31] M. Nguyen, S. Alesawi, N. Li, H. Che and H. Jiang, "A Black-Box Fork-Join Latency Prediction Model for Data-Intensive Applications," IEEE Transactions on Parallel and Distributed Systems, vol 31, no 9, pp.1983-3000, 2020.

[32] J. Dean and L. Barroso, "The Tail at Scale," Communications of the ACM, vol 56, no 2, pp.74–80, 2013.

[33] L. Suresh, M. Canini, S. Schmid and A. Feldmann, "C3: cutting tail latency in cloud data stores via adaptive replica selection," Proceeding of USENIX NSDI, 2015.

[34] "Statistical Review," http://www.kaspercpa.com/statisticalreview.htm.

[35] L. Ye, Z. Wang, H. Che and C. Lagoa, "TERSE: A Unified, End-to-end Traffic Control Mechanism to Enable Elastic, Delay Adaptive and Rate Adaptive Services", IEEE Journal on Selected Areas in Communications, vol26, no. 1, pp.938–950, 2011.

[36] Tsuna, "How long does it take to make a context switch?" https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html, 2010.

[37] F. David, J. Carlyle and R. Campbell, "Context Switch Overheads for Linux on ARM Platforms", Proceeding of the 2007 Workshop on Experimental Computer Science, 2007.

 ISBN: 978-1-61208-853-2