Using the Macroflow Abstraction to Minimize Machine Slot-time Spent on Networking in Hadoop

Bingchuan Tian, Chen Tian, Jiajun Sun, Junhua Yan, Yizhou Tang,
Wei Wang, Haipeng Dai, Nai Xia, Guihai Chen, and Wanchun Dou
State Key Laboratory for Novel Software Technology, Nanjing University, China

ABSTRACT

Machine slot-time spent on data transmission has direct impact on average *job completion time* (JCT). In this paper, we propose *Macroflow*, a networking abstraction that can capture the primitive scheduling granularity of machine slot-time. We demonstrate that minimizing machine slot-time is equivalent to minimizing the average *macroflow completion time* (MCT). We prove that minimizing MCT to be strongly NP-hard and focus on developing effective heuristics. We propose the *Smallest-Macroflow-First* (SMF) and *Smallest-Average-Macroflow-First* (SAMF) heuristics that greedily schedule macroflows based on their network footprint. To work with existing commodity switches, priority discretization is performed to classify macroflows into a small number of priority queues.

CCS CONCEPTS

• **Networks** → **Cloud computing**;

KEYWORDS

Macroflow; network scheduling; datacenter networking

ACM Reference Format:
Bingchuan Tian, Chen Tian, Jiajun Sun, Junhua Yan, Yizhou Tang, Wei Wang, Haipeng Dai, Nai Xia, Guihai Chen, and Wanchun Dou. 2018. Using the Macroflow Abstraction to Minimize Machine Slot-time Spent on Networking in Hadoop. In *APNet '18: 2nd Asia-Pacific Workshop on Networking, August 2–3, 2018, Beijing, China.* ACM, Beijing, China, 7 pages. https://doi.org/10.1145/3232565.3234504

APNet '18, August 2–3, 2018, Beijing, China

ACM ISBN 978-1-4503-6395-2/18/08…$15.00
https://doi.org/10.1145/3232565.3234504

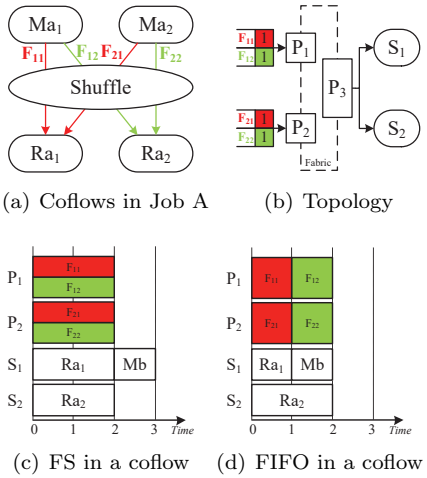
1 INTRODUCTION

In a data parallel computation frameworks such as Hadoop, jobs consume *machine slot-time*. Jobs are scheduled in the granularity of a series of *tasks* by the framework scheduler, where each task is assigned to a machine slot to read the input data, perform computation, and output results. A task's machine slot-time denotes the occupied machine slot duration spent on network and computation together. A job's machine slot-time is the sum of all its tasks' slot-time.

Reducing average JCT is crucial for application performance. When machine slots become insufficient, machine slot-time spent on networking has a direct impact on average *job completion time* (JCT). The sooner the occupied machine slot can be released, the quicker the jobs waiting in the queue can be scheduled.

State-of-the-art network researches focus on various network metrics but ignore the machine slot-time. Many work minimize *flow completion time* (FCT) [1–3, 8, 10, 11, 13]. Recently, the *coflow* abstraction decreases the gap between network and application metrics [4–7]: a coflow includes all correlated flows in a communication stage; its *all-or-nothing* semantic constrains that all flows of the stage must finish together for the completion of the stage. Minimizing *coflow completion time* (CCT) might lead to reduced average JCT [5–7].

There could be a large JCT penalty if network scheduling ignores the occupied machine slot-time spent on data transmission. Consider the example in Fig. 1: Job A has two mappers M_{a1}/M_{a2}, and two reducers R_{a1}/R_{a2} with negligible computation durations; totally there would be 4 flows in its coflow (Fig. 1(a)); there is an additional Job B, which only has a single mapper task M_b with 1 slot-time computation waiting to be scheduled. There exist ingress ports $P1/P2$ in this datacenter fabric, and each port can send 1 unit of data in one time unit; egress port $P3$ can receive 2 units of data per time unit (Fig. 1(b)); M_{a1}/M_{a2} are already finished, and each flow has 1 unit of shuffle data to be collected; two machine slots $S1/S2$ behind $P3$ hold R_{a1}/R_{a2} running independently. Apparently, Job A's JCT is equal to CCT, and the CCT is fixed (2 time units for Job A to



(a) Coflows in Job A (b) Topology

(c) FS in a coflow (d) FIFO in a coflow

Figure 1: An example: average JCT can be reduced without change CCT.

complete) regardless of flow scheduling. However, the average JCT can be quite different with different flow scheduling policies. A widely-accepted mantra is to finish all flows simultaneously within a coflow [5, 6]. Suppose we use fair sharing among flows, then all four flows finish in time 2. In this case, M_b can be allocated at time 3: the resulted average JCT is 2.5 time units (Fig. 1(c)). As a comparison, if we let flows F_{11} and F_{21} have priority, then they can be finished in 1 time unit without impacting the coflow finish time (Fig. 1(d)). In this case, R_{a1} has received all data and can be completed in time 1; then M_b can be scheduled to occupy the machine slot released by R_{a1}; as a result, the average JCT is only 2 time units.

The key observation is that to minimize machine slot-time on data transmission, neither flow nor coflow is the right abstraction for network scheduling. In the above example, machine slot-time spent by Job A can be minimized from 4 (Fig. 1(c)) to 3 (Fig. 1(d)), with its CCT unchanged. For an individual reducer in a shuffle phase, its computation stage can start as soon as all constituent flows are finished, which is independent of all other reducers. Prioritizing some reducers' flows over others, and finishing them earlier, can complete existing tasks earlier; in turn, new tasks can be allocated earlier and the average JCT can be reduced.

In this paper, we propose *Macroflow*, a networking abstraction that can capture the primitive scheduling granularity of machine slot-time. Each macroflow is a collection of flows between a single reducer and all mappers in the shuffle. The flows of a macroflow are independent to each other: there is no precedence among them; the completion of the latest flow is regarded as the completion of the macroflow. Shown in Fig. 1, two flows F_{11} and F_{21} together comprise a macroflow with R_{a1} as the destination; Given this definition, a coflow is a collection of macroflows, each with a distinct reducer. For example, we can use reducer R_{a1}/R_{a2} to denote the two macroflows in Job A's coflow. Macroflow is an abstraction with a granularity between flow and coflow. Marcoflow is equal to coflow only in the special case when there is only one reducer in a phase. The abstraction allows the network to take scheduling decisions on the collection to achieve an optimized goal. We summarize our contributions as follows.

- We demonstrate that the problem of minimizing the total machine slot-time of reducers is equivalent to minimizing the average *macroflow completion time* (MCT). We prove that minimizing average MCT to be strongly NP-hard and focus on developing effective heuristics.
- We propose the *Smallest-Macroflow-First* (SMF) heuristics that greedily schedules a macroflow based on its network footprint.
- The SMF approach might interleave macroflows from different jobs, since the correlations among macroflows in the same coflow are ignored. Correspondingly, we propose the *Smallest-Average-Macroflow-First* (SAMF) heuristics that greedily schedules all macroflows of a coflow together, based on the average footprint of all macroflows in this coflow.
- To make our algorithms work with existing commodity switches in datacenters, a priority discretization algorithm is developed to classify macroflows into a small number of priority queues.

2 OVERVIEW

2.1 Problem Statement

Minimizing MCT: For a job with N reducers, each reducer will occupy a machine slot from the shuffle starting time to the completion of its computation. We denote the occupation time of a reducer R_i as t_i, which consists of two parts. The first part is the time used for gathering data from mappers, *i.e.*, time to finish the macroflow, which we define as t_i^s. The second part is the time used for processing the data, which we define as t_i^c. Therefore, the total machine slot-time occupied by the reducers is $\sum_{i=1}^{N}(t_i^s + t_i^c)$. Note that the first part, $\sum_{i=1}^{N} t_i^s$, is equal to $N \times MCT$, where MCT is the average macroflow completion time. The second part, $\sum_{i=1}^{N} t_i^c$, is the total data processing time, which can be regarded as invariant. Therefore, to minimize the total machine slot-time, we can instead minimize the MCT.

NP Hardness: The MCT minimization problem can be reduced from the concurrent open shop problem [6], which is NP-hard [12]. Here we give the skeleton of our proof only, and details are omit due to space limitation. Consider a shuffle with m different mappers and we need to transfer data with an amount of p_{ij} from mapper M_i to reducer R_j. Suppose that all mappers have network capacity of 1, and the reducers have no network capacity limit. In this case, any concurrent open shop problem with m machines and N orders where order j requires

p_{ij} processing time on the specific machine i [9], can be converted to the above shuffling problem. As minimizing the order completion time with $m \geq 2$ machines in the concurrent open shop problem is NP-hard [9], minimizing the MCT is also NP-hard when there are more than one mappers. Given the complexity, we instead focus on developing effective heuristics.

2.2 Architecture

In addition to its primary objective of minimizing MCT, we expect a practical design to satisfy the following goals.

- **No-global-scheduling.** For scalability concern, we should not centrally schedule either flow start time or flow rate.
- **Work-conserving.** The system must fully utilize available bandwidth.

The key idea is to only decide flow priority and let the network enforces the priority decision. We use a coordinator to gather/distribute macroflow/coflow information among machines. Based on the running jobs, a network priority is assigned for every running flow. Our Linux kernel modules attach the priority value to each packet of a pass-through macroflow. The network uses preemptive scheduling for packets based on their flows' priority. This design choice is important for fault tolerance and scalability; distributed priority enforcement also provides work-conserving bandwidth utilization. If the switches support an arbitrary number of priority queues (*e.g.*, pFabric [2]), our scheduling can get the best performance improvement. Otherwise, to be deployment-friendly to existing commodity switches, a priority discretization algorithm is developed (§3).

2.3 Scheduling Opportunities

We demonstrate the advantages of macroflow over coflow using real-world traces. We analyzed a Hadoop trace collected from a 3000-machine, 150-rack cluster [4]. Based on the number of mappers and reducers in the job's shuffle phase, we classify coflows to four network patterns: 1-to-1, M-to-1, M-to-N and 1-to-N. Here M-to-1 means that there are M mappers and only 1 reducer in the coflow, and vice visa. The network footprint of different coflow patterns (percentage in total number of jobs) are shown in Table 1. We also break down coflows to corresponding macroflows with the average footprint of macroflows enclosed in parentheses. For 1-to-1 and M-to-1 jobs, results are the same for coflow and macroflow since each job has only one reducer. So, we only show macroflow results for M-to-N and 1-to-N coflows.

1-to-1 and M-to-1 Coflows: For 1-to-1 coflows, over 95% jobs have footprint less than 10 MB, and over 87% jobs even have footprint less than 1 MB. Less than 5% jobs have footprint between 10 MB and 1 GB, and none of the job is larger than 1 GB. M-to-1 jobs are similar. Over 40% jobs have footprint less than 10 MB, and over 48% jobs have footprint between 10 MB and 100 MB. Only 0.48% jobs have footprint larger than 1 GB. We then analyze the distribution of the number of mappers per job. For 1-to-1 jobs, the network bottleneck can be near to either its mapper or its reducer. For M-to-1 jobs, over 63%/98% jobs have less than 10/30 mappers respectively. Usually, the reducer is the network bottleneck for M-to-1 coflows.

Table 1: Coflow (Macroflow) Footprint Pattern

Size(MB)	1-to-1	M-to-1	M-to-N	1-to-N
0-10	95.12%	40.28%	7.85%(12.86%)	0%(92.31%)
10-100	2.44%	48.34%	4.29%(32.14%)	69.23%(7.69%)
100-1000	2.44%	10.90%	14.29%(34.29%)	26.92%(0%)
$\geq$ 1000	0%	0.48%	73.57%(20.71%)	3.85%(0%)

M-to-N and 1-to-N Coflows: As a comparison, over 73% M-to-N jobs have over 1 GB data to shuffle and over 87% of them have over 10 reducers. When looking from the macroflow level, M-to-N jobs demonstrates large diversity. Over 12% jobs have average macroflow footprint less than 10 MB, while there are also over 20% jobs have average macroflow footprint larger than 1 GB. 1-to-N is the most interesting type. Around 70% of them have footprint between 10 MB to 100 MB, and over 15% jobs have more than 50 reducers each. However, at macroflow level, over 92% jobs have average macroflow footprint less than 10 MB.

Optimization Opportunities: Compared with the coflow concept, the new macroflow abstraction provides scheduling opportunities to reduce machine slot-time spent on networking. Some coflows are more *parallel* than others in terms of the number of reducers. Some flows that have low priority when scheduling with the coflow semantics, now can have higher priority when scheduling with the macroflow semantics. Such opportunities of priority inversion are common in our analyzed trace: with the SCF scheduling, all 1-to-N job's flows are lower in priority than flows of at least 40% M-to-1 jobs, since their coflows are larger in footprint. If scheduling at the macroflow-level, as shown in Table 1, as high as 80% 1-to-N jobs' flows can be prioritized, since their macroflows are smaller in footprint. There are also opportunities for M-to-N jobs. For example, one M-to-N shuffle is 5% larger than another shuffle, but it has 10× more reducers. Prioritizing this shuffle has a large chance of saving computing slot-time.

2.4 Scheduling Policies

SMF: We propose the *Smallest-Macroflow-First* (SMF) heuristics that greedily schedules a macroflow based on its network footprint. SMF is inspired by *Shortest-Job-First* (SJF). It is reported that *Smallest-Effective-Bottleneck-First* (SEBF) and *Minimum-Allocation-for-Desired-Duration* (MADD) heuristics, which greedily

schedule based on network bottleneck's completion time, might be marginally better than a smallest-size based solution [6]. However, they require heavy weight centralized scheduling and end-host rate control, which does not meet our design goals.

SAMF: The SMF approach might interleave macroflows from different jobs, since the correlations among macroflows in the same coflow are ignored. This might result in the increase of tail latency for jobs with multiple reducers. Correspondingly, we propose the *Smallest-Average-Macroflow-First* (SAMF) heuristics that greedily schedules all macroflows of a coflow together, based on its all contained macroflow's average footprint.

Discussion: Unlike Hadoop shuffle, there could exist scenarios where the size of each flow is unknown or not available to the scheduling system, *i.e.*, the non-clairvoyant scenarios [3, 4]. Here we can follow the Least-Attained-Service (LAS) rule, instead of the SJF rule.

3 PRIORITY DISCRETIZATION

The scheduling results are the relative priority among macroflows. However, existing commodity switches in datacenters usually have only 4-8 priority queues. In this section, we consider the problem of assigning a given set of macroflows to a handful of priority queues so that their average MCT is minimized.

3.1 Formulation

To make the formulation trackable, we also assume that there is only one bottleneck in the network [3]; macroflows with the higher priority are transferred first; macroflows with the same priority have an equal share of the bottleneck. Without lost of generality, we assume that the capacity of the bottleneck is one. Suppose there are n macroflows and their size are given as $d_i, i = 1 \dots n$; flows are sorted by their size, *i.e.*, $d_1 \leq d_2 \dots \leq d_n$. In general, if we have n different priorities, the SMF scheduling will have the smallest MCT.

Consider a simple sample with $n = 4$ macroflows with size of $d_1 = 1$, $d_2 = 2$, $d_3 = 3$, and $d_4 = 4$. Consider the case that we assign a higher priority to macroflow 1 and 2, and denote the assignment as (12)(34). In this case, macroflow 1 and 2 will be transmitted first and each of them has an equal transmission rate of 0.5. At time 2, macroflow 1 will finish transmission so the completion time for macroflow 1 is $f_1 = 2$. After macroflow 1 finished, macroflow 2 will occupy the full capacity of the bottleneck and transmit with a rate of 1. At time 3, macroflow 2 will finish transmission and macroflow 3 and 4 starts. Similarly, macroflow 3 and 4 get equal share rate of 0.5 and macroflow 3 finishes at time 9. Therefore, the average MCT for this priority assignment is 6, as shown in the second row of Table 2. We observe from Table 2 that different priority assignment gives different average MCT.

Table 2: MCT for different priority assignment

Assignment	f_1	f_2	f_3	f_4	Average MCT
(1)(234)	1	7	9	10	6.75
(12)(34)	2	3	9	10	6
(123)(4)	3	5	6	10	6

We can calculate the completion time for a given macroflow as follows.

LEMMA 3.1. *The completion time for a macroflow i with size of d_i and priority k is equal to*

$$f_i = \sum_{j=1}^{i} d_j + (n_k - i)d_i, \tag{1}$$

where n_k is the total number of macroflows that have the priority smaller or equal to k.

PROOF. Consider the time that macroflow i finishes. At that time, all macroflows with higher priority than k should have already finished. Furthermore, macroflows with the same priority of k and a smaller size than macroflow i should also have finished as they at least get the same share of the bottleneck capacity as macroflow i. Now consider the macroflows with the same priority of k and a larger size than macroflow i. These macroflows have an equal share of the bottleneck as macroflow i. So, each of these macroflows transmits data of d_i when macroflow i finishes transmission. As there are $n_k - i$ such macroflows, the total amount of data they have transmitted is $(n_k - i)d_i$. If we plus the amount of data of macroflows that has a smaller or equal size than macroflow i, which is $\sum_{j=1}^{i} d_j$, we can get the completion time of macroflow i as in Eq.(1). □

From Lemma 3.1, it is easy to get the following results.

COROLLARY 3.1. *Changing the priority of a macroflow that has smaller size than macroflow i does not change the completion time of macroflow i.*

The average MCT of N macroflows with K different priorities can be given by:

THEOREM 3.1. *For a coflow with N macroflows with sorted size of $d_i, i = 1 \dots N$, the total MCT is given by:*

$$MCT = \sum_{i=1}^{N}(N - 2i + 1)d_i + \sum_{k=1}^{K}\left(n_k \sum_{i=n_{k-1}+1}^{n_k} d_i\right) \tag{2}$$

PROOF. The total MCT is $\sum_{i=1}^{N} f_i$. From Eq.(1), we can see d_i appears in the first term in the completion time of macroflows with index equal or larger than i. As there are $(N - i + 1)$ macroflows with index equal or larger than i, d_i appears in the first term of Eq.(1) for $(N - i + 1)$ times. We can see that d_i appears in the

second term of Eq.(1) for $(n_k - i)$ times. Summing the two cases, we can get the result in Eq.(2). □

3.2 Algorithm for SMF

Theorem 3.1 gives important hints for minimizing MCT. The first term in Eq.(2) is independent to the priority assignment. So, we only need to focus on the second term, which expands to:

$$n_1 d_1 + n_1 d_2 + \ldots + n_2 d_{n_1+1} + n_2 d_{n_1+2} + \ldots \quad (3)$$

To minimize this term, we can first assign all macroflows to the lowest priority, *i.e.*, priority K. We then scan the macroflows to see if we can raise the priority of some macroflow. Note that we always move the macroflow with the smallest size in priority k, *i.e.*, macroflow $n_{k-1}+1$, to the higher priority level $k-1$. Therefore, before moving macroflow $n_{k-1}+1$ to a higher priority, the summation of second term in Eq.(2) is:

$$n_1 d_1 + \ldots + n_{k-1} d_{n_{k-2}+1} + n_{k-1} d_{n_{k-2}+2} + \ldots + n_k d_{n_{k-1}+1} + \ldots \quad (4)$$

Note that by Corollary 3.1, macroflows with size larger than the moving macroflow will not be effected, so we can safely ignore the terms containing d_j with $j > n_{k-1}+1$. When we move macroflow $n_{k-1}+1$ to priority $k-1$, we can see the number of n_{k-1} is increased to $n_{k-1}+1$, while n_k is not changed. So the summation in Eq.(4) changes to:

$$n_1 d_1 + \ldots + (n_{k-1}+1) d_{n_{k-2}+1} + (n_{k-1}+1) d_{n_{k-2}+2} + \ldots + (n_{k-1}+1) d_{n_{k-1}+1} + \ldots \quad (5)$$

Comparing Eq.(4) and Eq.(5), we can see the total MCT is changed by:

$$\sum_{i=n_{k-2}+1}^{n_{k-1}} d_i - (n_k - n_{k-1} - 1) d_{n_{k-1}+1} \quad (6)$$

From Eq.(6), we observe that promoting macroflow $n_{k-1}+1$ from priority k to priority $k-1$ has two effects. The first term in Eq.(6) is an increase in the MCT of macroflows already in the $k-1$ priority. This means by add a new macroflow to priority $k-1$, the total MCT increase in the existing macroflows increases by the amount that equals the sum of all macroflows in that priority. The second term in Eq.(6) is an decrease in the MCT of the promoted macroflow, which is equal to $(n_k - n_{k-1} - 1) d_{n_{k-1}+1}$, *i.e.*, timing the size of the promoted macroflow by the number of macroflows in the original priority minus one. Therefore, if we have:

$$(n_k - n_{k-1} - 1) d_{n_{k-1}+1} > \sum_{i=n_{k-2}+1}^{n_{k-1}} d_i, \quad (7)$$

promoting the given macroflow will result in a net decrease in the total MCT.

We use Algorithm 1 to find priority assignment for a given set of macroflows. Our algorithm first assigns all macroflows to the lowest priority and then iteratively moves small macroflows to higher priority when they satisfy the condition in Eq.(7). Note that after promoting a macroflow into a higher priority level, the condition in Eq.(7) is always satisfied even if we change the priority of other macroflows. This is because the number of macroflows in priority level $k-1$ can only decrease, so that the penalty of moving macroflow i into level $k-1$ only reduces. Furthermore, the number of macroflows in priority level k can only increase afterwards, so that the gain of moving macroflow i into level $k-1$ only increases. Therefore, once we move a macroflow into a higher priority level, we never need to move it back. As we have K priority levels and N macroflows and each macroflow can be changed by at most $K-1$ times, our algorithm will converge in $N(K-1)$ iterations. Noting that we need to check at most $K-1$ times to choose a macroflow and adjust its priority in each iteration, the upper bound of the time complexity of our assignment algorithm is $O(NK^2)$.

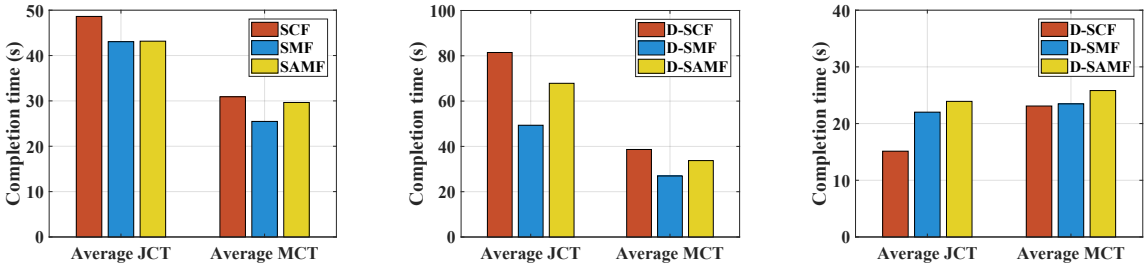
Algorithm 1: Priority Assignment

Input: The sorted size of macroflows, d_i, $i = 1 \ldots N$; Number of priority levels, K.
Output: Priority of each macroflow

```
1 Assign all macroflows to the lowest priority K.
2 while Priority assignment changed in previous iteration do
3     for k=2 to K do
4         Find the first (smallest) macroflow i in priority level k.
5         if (n_k − n_{k−1} − 1) d_{n_{k−1}+1} > Σ_{i=n_{k−2}+1}^{n_{k−1}} d_i then
6             Move marcoflow i to priority level k − 1.
7             Update n_k and n_{k−1}.
```

As an illustration, let's revisit our example with two priority levels. Consider the case that we assign macroflow 1 and 2 to the higher priority. We have $n_1 = 2$ and $n_2 = 4$. We can verify Eq.(1), *e.g.*, $f_2 = d_1 + d_2 + (2-2) * d_2 = 3$. We can also see that the total MCT is $(4-2+1)d_1 + (4-4+1)d_2 + (4-6+1)d_3 + (4-8+1)d_4 + 2d_1 + 2d_2 + 4d_3 + 4d_4 = 5d_1 + 3d_2 + 3d_3 + 1d_4 = 24$. If we move macroflow 3 to priority 1, the existing priority 1 macroflows (macroflow 1 and 2) will have a total increase in the MCT of $1 + 2 = 3$. And the macroflow 3's MCT will be decreased by $(n_2 - n_1 - 1)d_3 = d_3 = 3$. So, moving macroflow 3 will not change the total MCT as expected.

Adaptation for other algorithms: To make SAMF and SCF also supported by commodity switches, we extend the above algorithm for them respectively (*i.e.*, D-SAMF and D-SCF). The same algorithm can be applied to SCF directly by raising the network granularity from macroflow level to coflow level. For SAMF, suppose there are m coflows and their average macroflow size are given as $d_i, i = 1 \ldots m$ sorted by ascending order *i.e.*, $d_1 \leq d_2 \ldots \leq d_m$; there are $k_i, i = 1 \ldots m$ macroflows



(a) With infinite priority levels (b) With limited priority levels (c) A counter-example

Figure 2: Simulation results.

in the coflows respectively. We expand the coflow list to a pseudo macroflow list by adding d_i for k_i times in the list. After that, our algorithm can be applied to the pseudo macroflow list to find the boundary.

4 EVALUATION

4.1 Methodology

We evaluate our algorithms by performing a replay of the collected Facebook log [4, 6] with a flow-level simulator. We assume there are 30 hosts connected to a switch via a 1Gbps link, while the computation resource in each host is limited. Note that link bandwidth does not matter, because details of transport protocols are abstracted in a flow-level simulator. Currently, our algorithm only aims at MapReduce-like frameworks, thus we generate jobs following the MapReduce job patterns. The Facebook log contains only the shuffle information of a job, thus we generate mappers according to shuffle size to guarantee that the time spent on map-phase and reduce-phase is comparable. Without loss of generality, here we assume mappers contain computation phase only, and reducers contain no extra computation phase. In addition, we assume only when all mappers in a job finish could shuffle starts.

We choose the average JCT and the average MCT as metrics for evaluation, and compare our algorithm with coflow-based scheduling algorithm. Results show that our algorithm performs better when the system is heavily-loaded thus computation resource is lacking. In other scenarios, perhaps we may use coflow abstraction directly.

4.2 Simulation Results

We evaluate our algorithms with both infinite (*i.e.*, sufficiently large) and limited priority levels in heavily-loaded scenarios, and results are shown in Fig. 2(a) and 2(b), respectively. We use prefix `D` to mark the later case in figures. Macroflow-based scheduling algorithms perform better than coflow-based algorithm; the average JCT and MCT are decreased by up to 39.43% and 20.36%, respectively. Performance of SAMF lies between SMF and SCF, which indicates that, under this scenario, the JCT-decreasing caused by saving computation resource is larger than JCT-increasing caused by interleaving coflows. In addition, the same conclusion can also be obtained from the fact that the average JCT and the average MCT is highly-correlated, which demonstrates that saving computation resource does lead to a smaller JCT. As a result, macroflow abstraction can be a trade-off in a (temporarily) heavily-loaded system.

Meanwhile, we noticed that macroflow abstraction performs worse than coflow abstraction when the system load is not high. Comparing with aforementioned scenario, we evaluate our algorithms with 1.5× more computation resource in each host, and reduce the traffic load by 30%, and results can be found in Fig. 2(c). It is clear that when there is no need to saving computation resource, using coflow abstraction directly appears to be a better choice.

5 CONCLUSION

Network researchers used to optimize network metrics while ignore occupied machine slot-time spent on data transmission, which may lead to inferior job completion time. In this paper, We propose a networking abstraction *Macroflow* and study the inter-macroflow scheduling problem. Trace-driven simulations demonstrate that our algorithms can significantly reduce the average JCT for jobs.

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their valuable comments. This work was supported in part by the National Science and Technology Major Project of China under Grant Number 2017ZX03001013-003, the Fundamental Research Funds for the Central Universities under Grant Number 0202-14380037, the National Natural Science Foundation of China under Grant Numbers 61772265, 61602194, 61502229, 61672276, and 61321491.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data center tcp (dctcp). In *in Proc. ACM SIGCOMM 2011*, Vol. 41. ACM, 63–74.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM 2013*, Vol. 43. ACM, 435–446.

[3] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*. USENIX.

[4] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 393–406.

[5] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. 2011. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM*, Vol. 41. ACM, 98–109.

[6] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In *ACM SIGCOMM*. ACM, 443–454.

[7] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. 2014. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 431–442.

[8] Chi-Yao Hong, Matthew Caesar, and P Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *Proc. ACM SIGCOMM 2012*, Vol. 42. ACM, 127–138.

[9] Joseph Y-T Leung, Haibing Li, and Michael Pinedo. 2007. Scheduling orders for multiple product types to minimize total weighted completion time. *Discrete Applied Mathematics* 155, 8 (2007), 945–970.

[10] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. 2014. Friends, not foes: synthesizing existing transport strategies for data center networks. In *ACM SIGCOMM 2014*. ACM, 491–502.

[11] Ali Munir, Ihsan Ayyub Qazi, Zartash Afzal Uzmi, Aisha Mushtaq, Saad N Ismail, M Safdar Iqbal, and Basma Khan. 2013. Minimizing flow completion times in data centers. In *in Proc. IEEE INFOCOM, 2013*. IEEE, 2157–2165.

[12] Thomas A Roemer. 2006. A note on the complexity of the concurrent open shop problem. *Journal of scheduling* 9, 4 (2006), 389–396.

[13] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM*, Vol. 42. ACM, 139–150.