Labeling Scheduler: A Flexible Labeling-based Jointly Scheduling Approach for Big Data Analysis

Xin Li*§, Zhuzhong Qian†, Jianjun Qiu*, Xiaolin Qin*, Jie Wu‡
*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics
†State Key Laboratory for Novel Software Technology, Nanjing University
‡Center for Networked Computing, Temple University
§State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences

***Abstract*—The emerging Non-Volatile Memory (NVM) technology has given rise to an opportunity to accelerate big data analysis. In this paper, we investigate the joint job and data scheduling problem in private cloud data center with a hybrid storage system, and we propose *Labeling Scheduler*, a flexible labeling-based approach for jointly scheduling. The core idea of the approach is to introduce the labeling system to characterize the features of big data analysis jobs and data objects, and conduct data replacement dynamically between NVM and disk. To the best of our knowledge, this is the first work to introduce the labeling methodology to the big data analysis problem in the cloud data center with a hybrid storage system. We conduct extensive simulations and the simulation results show that the *Labeling Scheduler* has a significant improvement on system utility compared to the method without labeling information. In addition, the *Labeling Scheduler* guarantees a high NVM hit rate, which is valuable for NVM endurance enhancement.**

***Index Terms*—cloud data center, hybrid storage system, NVM, joint scheduling, labeling method**

I. Introduction

There is a strong need to conduct big data analysis to gain valuable information for organizations. For these data-intensive analysis jobs, timely output is critical for decision making and interactive services. This makes makes job scheduling the primary issue for big data analysis in cloud data centers, and has been studied in various aspects, including data locality [1], makespan [2], energy [3], fairness [4], resource sharing [5], and data placement [6].

Traditionally, the data analysis procedure reads data from disk to memory. It is time consuming because the read performance of disk is rather poor, compared to the memory. In addition, the current DRAM is approaching scalability limits [7], which makes it hard to extend the capacity to realize in-memory computing [8]. The emergence of Non-Volatile Memory (NVM) [9] makes it possible to realize storage-class memory and in-memory computing. However, it is still impractical to use the NVM independently due to its limited write performance and endurance [9]. The hybrid storage system consisting of NVM, DRAM, and disk is the practical architecture for physical machines (PMs) [10] [11] [12]. NVM is inappropriate for general usage as a memory device because of its drawbacks. However, for big data analysis jobs, most operations on datasets are readings and few writings are required, which is perfect for using NVM. This motivates us to utilize NVM to improve big data analysis.

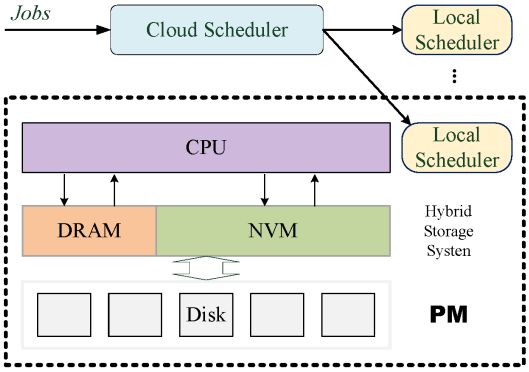


Fig. 1. System Scenario

In this paper, we investigate the joint job and data scheduling problem in private cloud data center with a hybrid storage system, which includes DRAM, NVM, and disk. Generally, for the job scheduling issue, the controller (*cloud scheduler*) assigns the jobs to PMs, and *local scheduler* will determine the final assignment for jobs, as shown in Fig. 1. The representative *cloud scheduling* methods include delay scheduling [1], deadline-aware scheduling [13], and locality-aware scheduling [14]. Here, we focus on the *local scheduler* and take the hybrid storage devices into account. This means that data scheduling is very important since the location of data object affects the job execution time significantly. It is preferable to place data in NVM. As a result, there must be some data replacement between NVM and disk to achieve better performance.

To represent the efficiency of big data analysis, we introduce *utility* to measure the timeliness for job completion, and we take *utility* maximization as the major objective for the jointly scheduling problem. According to the response time requirement, we classify the jobs into two categories, interactive job and batch job. For the interactive jobs, we need to return the analysis results in a real-time manner. On the other hand, the batch jobs can be processed within longer duration. Hence, we define various utility functions based on the job features. Intuitively, shorter job execution time guarantees faster job response, which brings more *utility*. Hence, the core issue is to speed up job execution. As mentioned above, the job completion time can be shortened if its input data is stored in the NVM. However, it is challenging to guarantee that the

wanted data objects are always stored in NVM due to limited storage slots. There must be efficient data replacement between NVM and disk to achieve a greater NVM hit rate, where *NVM hit* means the input data of the executing job is stored in NVM.

We introduce the labeling method [15] to characterize the data features. Here, the features imply the comprehensive priority to occupy NVM resource. Based on the labeling system, we propose `labeling scheduler`, a flexible labeling-based approach for joint job and data scheduling. We conduct extensive simulations, and the results show that the labeling system has a significant improvement on utility. To the best of our knowledge, this is the first paper to introduce the labeling system for joint job and data scheduling in private cloud data center with a hybrid storage system. The main contributions of our work can be summarized as follows:

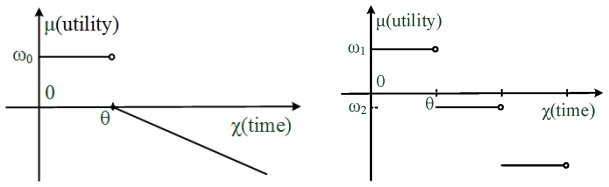
- We formulate the joint job and data scheduling problem in private cloud data center with a hybrid storage system. We introduce the emerging NVM to speed up the big data analysis job execution, and establish the data replacement model between disk and NVM.
- We take the data features into account for data replacement and build a labeling method to describe the data features. Based on the labeling system, we propose `labeling scheduler`, a flexible labeling-based approach for joint job and data scheduling.
- We conduct extensive simulations based on one Alibaba cluster trace. The results show that our approach has a significant performance improvement compared to the typical scheduling algorithm without considering the data features.

II. Scenario and Problem Statement

A. Scenario and Preliminaries

We consider the jointly scheduling problem with the scenario as shown in Fig. 1. Given a private cloud data center with a hybrid storage system for big data analysis, we assume that the big data analysis jobs will not rewrite the data object. Hence, we let the DRAM be used to record the data-analysis results and non-data-analysis data object as usual. The input data objects for big data analysis jobs are stored in NVM or disk. Compared to the limited space for NVM, the disk is sufficient for storing data. The *cloud scheduler* assigns the coming jobs to the physical machines (PMs) according to the global job scheduling algorithm [1] [16]. It forms a job queue in the PM. Then, the *local scheduler* is in charge of jointly scheduling the jobs and data objects on the PM with the hybrid storage system. Because of the privateness of the cloud data center, it is feasible to obtain the business details by summarizing the history information. Hence, we can assume that the cloud scheduler knows the job profiles from the job submitter, and the cloud scheduler will forward the job details to the local scheduler when assigning jobs.

For each job, it may contain multiple tasks to accomplish special works. To clearly describe the job, we use a tuple to characterize it as $J = \langle \Gamma_a, \mathcal{I}, \theta, \chi, \omega, \mathcal{S} \rangle$, where



(a) Interactive Job (b) Batch Job

Fig. 2. Utility Functions

- Γ_a is the job arrival time to the cloud system;
- θ is the required duration to return the analysis result;
- χ is the real duration between job arrival and completion;
- ω is the full utility for finishing the job in time;
- $\mathcal{S}$ is the set of tasks contained in the job. For the items in $\mathcal{S}$, we call them the job's children tasks and the job is known as their parent job. The job is completed only when all children tasks are finished;
- $\mathcal{I}$ is used to identify the job is an interactive job or batch job. For the job J_k, we have

$$\mathcal{I}(J_k) = \begin{cases} 1, & J_k \text{ is an interactive job;} \\ 0, & \text{otherwise.} \end{cases}$$

For each task, there is one data object as its input data, i.e. the analysis object. Though there is one input data object for each task, there may be many tasks that need the same data object. This means there are diverse analysis or business requirements on the same data set. We can also use a tuple to represent the task as $A = \langle \Gamma_e, D \rangle$, where Γ_e is the expected task execution time when the wanted data object is stored in NVM, and D is the input data for the task. We also use the function $d(A)$ to represent the input data for task A, i.e. $d(A) = D$. Furthermore, to express the relationship between job and task, we define a new function as $job(A_i) = J_k$, which means that task A_i belongs to job J_k, i.e. $A_i \in \mathcal{S}(J_k)$.

From the perspective of task execution, the physical computing resource is split into m computing slots, or virtual machines (VMs), and the NVM is also split into n storage slots. It is a widely used resource usage model [16] [17]. For each task execution, it will occupy one VM and read data from one slot in NVM or disk.

B. Utility Functions

We aim to maximize the total utility for the system, so it would be ideal to finish all the jobs before their deadlines. However, the resource is limited, and we need to select some jobs with high priority to occupy the computing resource slots. Therefore, there may be some jobs that cannot be completed before the deadline, leading a negative utility, i.e. penalty.

For the interactive jobs, the utility will drop to negative if the completion time goes beyond the deadline. We define the utility for interactive jobs as:

$$\mu(J_i) = \begin{cases} \omega_0, & \chi < \theta; \\ -\alpha(\chi - \theta), & \text{otherwise.} \end{cases}$$

This function is shown in Fig. 2(a).

For the batch jobs, most of them are periodic jobs. There will be positive utility if they are completed within the decision

cycle. Let the attribute θ represent the decision cycle, then we can define the utility for the batch job as:

$$\mu(J_i) = \omega_1 - (\omega_1 - \omega_2)\lfloor\frac{\chi}{\theta}\rfloor.$$

This function is also shown in Fig. 2(b). The utility function means that the analysis result is useful for decision making within each cycle.

If the workload is high, it is expected that the PM cannot finish all the arrival jobs in time. Hence, we should allow the PM to reject some jobs. In this case, the utility for the rejected job is 0.

C. Job Execution Time

There are two cases for each task execution: *NVM-case* and *disk-case*. *NVM-case* implies that the input data of the task is stored in NVM, and the *disk-case* means the task needs to migrate the data from disk to NVM first, after which the task can be executed. Hence, we know that the task execution time for the disk-case consists of two parts, the time to read data from disk to NVM, and the task execution time under the NVM-case. We use $\Gamma_N(A_i)$ to represent the task execution time under the NVM-case, which could be given by the cloud scheduler based on the history information. Hence, according to the location of the input data of the task, we can define the task execution time easily. We have

$$\Gamma_e(A_i) = \begin{cases} \Gamma_N(A_i), & loc(d(A_i)) = 0; \\ \Gamma_N(A_i) + \Gamma_r, & \text{otherwise.} \end{cases}$$

$$= \Gamma_N(A_i) + loc(d(A_i)) \cdot \Gamma_r.$$

Here, Γ_r is the data migration time, and the data location $loc(D_i)$ is defined as:

$$loc(D_i) = \begin{cases} 0, & D_i \text{ is stored in the NVM;} \\ 1, & \text{otherwise.} \end{cases}$$

For the job execution time, the value varies as the execution mode of the tasks. Here, we define the job execution time as the value when all the tasks are executed in serial. Hence, we have

$$\Gamma_e(J_k) = \sum_{A_i \in J_k} \Gamma_e(A_i). \quad (1)$$

To gain a positive utility while job scheduling, we should be aware that there must be some critical time point to assign a computing slot to the job J_k. The point is known as Last Scheduling Time (LST), which means that the utility will be positive if the job can be executed before LST. According to the job (J_k) profile, we can infer the LST as

$$LST(J_k) = \Gamma_a(J_k) + \theta(J_k) - \Gamma_e(J_k). \quad (2)$$

Here, the $LST(J_k)$ is an absolute time as arrival time $\Gamma_a(J_k)$, not the relative time as deadline $\theta(J_k)$. In addition, we should be aware that the LST may change if some input data of the job is migrated from disk to NVM. Hence, we will update the job LSTs online.

D. Problem Statement and Analysis

Jointly Scheduling Problem. For the given private cloud data center with multiple PMs, let there are m computing resource slots for task execution, and each task occupy one slot once. We also split the NVM into n storage slots, and each data object will occupy one slot, which is similar to the data block in HDFS. The jobs arrive at the cloud system online, and there is positive utility ω for timely job completion. The cloud scheduler will forward the jobs to the PM and form a job queue $\mathcal{Q}$. For each PM, the local scheduler must schedule the jobs and data objects according to the job profiles $\cup J_i$ $(1 \leq i \leq \kappa)$, such that the system utility is maximized. The objective can be represented as:

$$max. \quad \sum_{i=1}^{\kappa} \mu(J_i).$$

Theorem 1: The jointly scheduling problem is NP-hard.

Proof: We prove the theorem by showing a special case is NP-hard. We assume that there are many offline jobs, each of them contains one task and the NVM capacity is sufficient to store all the input data. Hence, the job $\cup J_i$ $(1 \leq i \leq \kappa)$ can be represented by a two-tuple $\langle \omega_i, \Gamma_i \rangle$, where ω_i is the utility to complete the job, and Γ_i is the job execution time. The problem is to select part of the jobs such that the total utility is maximized. The selection constraint is that the total job execution time is no more than Θ. Then, we show that this special case can be reduced from the Knapsack problem.

For the Knapsack problem, given a set of items $\cup B_i$ $(1 \leq i \leq \kappa)$, each with a weight w_i and a value v_i, the problem is to select some of the items so that the total weight is less than or equal to Φ and the total value is maximized. We can construct the jointly scheduling problem by setting $\omega_i = w_i$ $(1 \leq i \leq \kappa)$, $\Gamma_i = v_i$, and $\Theta = \Phi$. If there exists a solution for the Knapsack problem, we can select the associated jobs from the packed items. On the other hand, if there is a selection solution for jobs to maximize the utility, we can pick the associated items in Knapsack problem such that the total value is maximized. Hence, the jointly scheduling problem is NP-hard. ■

We analyze the problem and find that the features of data are critical for data placement. This motivates us to explore the labeling method and extract the data semantics, which will help us to conduct data selection during data replacement.

III. Labeling System

In this section, we will first build a labeling system for the data objects, and propose the labeling scheduler, a labeling-based scheduling approach.

A. Label: hotness

For the jointly scheduling problem, the key issue is to place proper data in NVM. Intuitively, the data object that will be frequently read should be placed in NVM. Hence, we define a label *hotness* to represent the popularity degree. The basic idea of *hotness* is to record the number of tasks that take the data object as input data within a time duration; this implies

the frequency of reading the data, and also the importance to guarantee timely job completion.

The value of *hotness* increases if some arrived jobs take the data as input, which could be expressed as

$$hotness(D_i) \mathrel{+}= H \cdot K,$$

where H is a constant, and K is the number of jobs that take D_i as the input.

Furthermore, the increment for interactive job and batch job should be different, because the deadline of interactive job must be much shorter than batch job. Hence, we divide the constant H into two parts, $H1$ and $H2$. $H1$ is the common increment for both interactive job and batch job, and $H2$ is the extra increment for interactive job. As a result, when a set of jobs arrive at the system at some time-slot, and let $U(D_i)$ be the subset whose element (job) takes D_i as input data,we have

$$hotness(D_i) \mathrel{+}= \sum_{J_k \in U(D_i)} H1 + H2 \cdot \mathcal{I}(J_k)$$

We should also be aware that it is unreasonable to accumulate the value infinitely, because the value should reflect the case within some time duration. To express this requirement, let the value decrease at each time-slot. Thus, at the beginning of each time-slot, the value of hotness decreases by 1.

For data objects with the similar hotness, it can be different for different cases. For example, one data is showing importance in the coming future, which means it will be read with higher frequency in the following time-slots. Another data may be read frequently in the past, and it needs some time let the data be cold. Hence, we need anther label to distinguish the cases, or the hotness trend. Hereby, we introduce another label to indicate the hotness trend for the data.

First, we will record the hotness value periodically with period equal to T, or, say, sample the hotness value. Then we can get the difference between the current sampling value and the last sampling value, and we use ΔH to represent the difference. ΔH can be a positive or negative value. It is easy to understand that the positive ΔH implies the data will be important in the coming future.

So far, we will use the two labels, $hotness$ and $\Delta H(D_i)$, to represent priority of whether some data should be stored in NVM. Actually, we will combine the two label as $hotness(D_i) + \Delta H$ in the following analysis.

B. Label: class

The label *hotness* shows the important feature of data object itself. However, the data is bound to the jobs. The label *hotness* cannot reflect the *host job* state. For the data object with similar $hotness$ and ΔH, the one whose *host job* is waiting in the job *queue* should be placed in NVM with higher priority, obviously. Hence, it is necessary to define a new label to deal with this case. Here, we will introduce the new *class* label by classifying the data into 5 classes according to the host jobs. We will discuss this for interactive job and batch job, respectively.

For the interactive jobs in the job *queue*, they have similar features expect LST. This is because there is only one task for each interactive job. The job selection is equal to task selection for the interactive jobs. However, it is different for batch jobs, since there may be multiple tasks for each batch job. We need to choose both job and task for batch jobs. Hence, there is a problem about the task selection for batch jobs. Should we always select the tasks from one single job or take no account of the parent job of the tasks?

Statement I: *We should always select the tasks for the same parent job until the job completion.*

Analysis: Say we have two batch jobs J_1 and J_2 in the job queue, both of which have multiple children tasks. We assume that the two job have the same utility function. We have one computing slot to conduct the task execution, so we need to decide the execution sequence of the tasks of both jobs. It may be alternant, one from J_1 and next from J_2, like $A_1(J_1)$, $A_1(J_2)$, $A_2(J_1)$, $A_2(J_2)$, ... Or, the tasks of J_1 are executed first, and the tasks from J_2 will be executed after J_1 is finished. According to the previous knowledge, the expected execution time for the two jobs is $\Gamma_e(J_1)+\Gamma_e(J_2)$. This means whatever the execution task sequence of J_1 and J_2. One of the two jobs, say J_2, accomplishes the work after $\Gamma_e = \Gamma_e(J_1) + \Gamma_e(J_2)$ time-slots, which is a fixed value. Hence, the utility of J_2 is fixed. We should maximize the utility for J_1. The best method to maximize $\mu(J_1)$ is to minimize the job execution time of J_1. Obviously, we should schedule the tasks of J_1 first. This means we should pick one specific job if there are many batch jobs in the queue to achieve maximized utility.

Based on the above analysis, we will pick only one batch job with higher priority to occupy the computing slot, even if there are multiple batch jobs. Hence, we can classify the batch jobs into two categories. Here, we define the picked job as *actress batch job*, while other batch jobs in the queue are *audience batch jobs*. The *audience batch jobs* watch until the *actress batch job* accomplishes the work, and the new *actress batch job* will be selected. We will show the selection algorithm in the next section.

So far, we have classified the job into multiple classes. Based on the status of the jobs, we give the definition of label *class* as follows.

$$class(D_i) = \begin{cases} 0, & \exists A_k,\ d(A_k) = D_i,\ A_k \text{ is running;} \\ 1, & \exists J_k \in \mathcal{Q},\ I(J_k) = 1 \wedge D_i \in d(J_k); \\ 2, & \exists J_k \in \mathcal{Q},\ D_i \in d(J_k), \text{ and} \\ & J_k \text{ is the actress batch job;} \\ 3, & D_i \text{ is read by interactive job before;} \\ 4, & \exists J_k \in \mathcal{Q},\ D_i \in d(J_k), \text{ and} \\ & J_k \text{ is an audience batch job;} \\ 5, & \text{otherwise.} \end{cases}$$

According to the *class* definition, we know that the data read by the running tasks is labelled by *class-0*. For the data that can be treated as the input data by the interactive jobs in the queue $\mathcal{Q}$, it will be labelled as *class-1*. For the batch jobs,

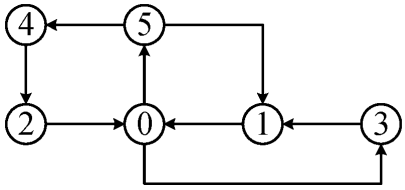


Fig. 3. Class State Transformation.

the data wanted by the actress batch job is marked as *class-2*, while the data will be marked as *class-4* for the audience batch jobs. We should be aware that, once some new actress batch job is selected, the related data will be labelled as *class-2* newly. For the data that are not read by the jobs in the queue, *class-3* implies that the data is the input data for some interactive job before, while *class-5* is used to represent other cases. Actually, the data with label *class-5* comes from the *class-0* data whose host job is a batch jobs. For the data with label *class-3*, it will be updated to *class-1* if some new coming interactive job takes it as input. It means that the class label of the data wanted by interactive jobs will transform between *class-1* and *class-3*.

For example, say we have data D_1 with the default label *class-5*. If some batch job J_2 arrives at the PM, and one of the children tasks A_3 takes D_1 as the input data, this event will trigger the class label to be changed to *class-4* since J_2 is regarded as one audience batch job first. When J_2 is picked as the new actress batch job, the class label of D_1 changes to *class-2*. It will further becomes *class-0* when the host task is scheduled to occupy the computing slot. After the task execution, the label returns to *class-5*. We show the class state transformation in Fig. 3.

IV. Labeling Scheduler

A. Basis Ideas

The local scheduler is responsible for the local PM resources, including computing resource slots, storage slots in NVM and disk, bus between disk and NVM, and data. The major task is to allocate the resources to the jobs, such that the utility by finishing the jobs is maximized. To gain more utility, it seems that the PM should execute as many jobs as possible. However, because the given resource must be limited, there is a maximal workload, and going beyond will result in a negative utility. Hence, the local scheduler may reject some jobs from the cloud scheduler. This procedure is known as *job admission*, which determines whether the PM should accept or reject the coming job. For the accepted jobs, they are added into the job queue $\mathcal{Q}$ first.

For the computing resource, once some computing slot is available, the local scheduler will pick one task of some job to occupy the resource. It means *job scheduling* is in charge of the allocation of computing resource. On the other hand, the storage slots in NVM are always full regularly. But we can conduct *data placement* between disk and NVM by the bus, which means to migrate some data from disk to NVM. Hence, *data placement* controls the bus and data for the system.

Algorithm 1 $admission(J_i)$

Input: J_i: the arrived job, $\mathcal{Q}$: the set of jobs of the queue.

1: **for** *each* $A_k \in \mathcal{S}(J_i)$ **do**
2: $\quad hotness(d(A_k)) += H_1 + H_2 \cdot \mathcal{I}(J_i)$;
3: $load \leftarrow 0$;
4: $LST(J_i) \leftarrow \Gamma_a(J_i) + \theta(J_i) - exe(J_i)$;
5: **for** *each* $J_k \in Q$ **do**
6: $\quad$ **if** $deadline(J_k) < LST(J_i)$ **then**
7: $\quad\quad load += expLoad(J_k)$;
8: **if** $\lceil \frac{load}{m} \rceil \leq LST(J_i)$ **then**
9: $\quad$ accept J_k;
10: $\quad$ update data label $class$;

Based on the above analysis, we propose the *labeling scheduler*, which consists of 3 parts: job admission, job scheduling, and data replacement. The basic ideas for the 3 components are summarized as:

- *Job Admission*. For the new candidate job, compare the workload and usable computing resource before the LST of the job.
- *Job Scheduling*. Once some computing slot is available, take LST of the jobs in queue $\mathcal{Q}$ as the major concern for job/task selection.
- *Data Replacement*. Take both the hotness and class into account comprehensively for data selection when the data bus is free.

B. Load-aware Job Admission

For each job arrival, say J_i, it triggers the *job admission* procedure, as shown in Alg. 1. From the definition of hotness, the value will be increased no matter which decision is made for the arrived job J_i. Hence, we first update the hotness value for the wanted data objects by J_i. Next, we can obtain the $LST(J_i)$ from the job profile (line 4), and we will measure the necessary workload before $LST(J_i)$ (line 5-7). The necessary workload means the jobs must be completed before $LST(J_i)$, because their deadline is less than $LST(J_i)$. Then, we compare the necessary workload and usable computing resource (line 8). The PM will accept the job if the expected computing resources are sufficient for the coming job J_i (line 9). In addition, once J_i is accepted into the queue $\mathcal{Q}$, the class label of input data ($d(J_i)$) will change (line 10). For example, the class label of the input data will move from *class-3* to *class-1* if J_i is an interactive job. The details and class transformation rule are given in Section III-B.

In this algorithm, one of the key points is to measure the workload for each job. The workload of job J_k ($jobLoad(J_k)$ in Alg. 1) is similar to the job execution $\Gamma_e(J_k)$, defined in Eq. 1. The $jobLoad(J_k)$) is also defined by its tasks.

$$jobLoad(J_k) = \sum_{A_i \in J_k} (\Gamma_N(A_i) + loc(d(A_i)) \cdot p \cdot \Gamma_r),$$

where p is a probability value.

From the definition, we know that if the input data of all $A_i \in J_k$ are stored in the NVM, the value of $jobLoad(J_k)$ and

Algorithm 2 LST: $jobScheduling(VM_i)$

Input: $\mathcal{Q}$: the set of jobs of the queue.

1: $LST \leftarrow \infty$, $\gamma \leftarrow \infty$;
2: $J \leftarrow NULL$, $A \leftarrow NULL$;
3: **for** *each* $J_k \in Q$ **do**
4: **if** $LST < LST(J_k)$ **then**
5: $LST \leftarrow LST(J_k)$;
6: $J \leftarrow J_k$;
7: **for** *each* $A_i \in J$ **do**
8: **if** $\gamma > \Gamma_e(A_i)$ **then**
9: $\gamma \leftarrow \Gamma_e(A_i)$;
10: $A \leftarrow A_i$;
11: assign VM_i to task A;
12: $class(d(A)) \leftarrow 0$;

$\Gamma_e(J_k)$ are exactly the same. Generally, $jobLoad(J_k)$ is less than $\Gamma_e(J_k)$ due to the probability value p. This is because we think some of the wanted data by J_k will be migrated to NVM from disk during the scheduling. In detail, the data will be migrated to NVM with probability equal to p. This provides an opportunity for the PM to accept more jobs.

C. LST-based Job Scheduling

Once there is any free computing slot, say VM_i, *job scheduling* needs to pick one task from the jobs in the queue $\mathcal{Q}$, as shown in Alg. 2. The basic idea is to select the job with the least LST (line 3-6). Then, we pick the task with minimal execution time of the job (line 7-10). Next, assign the available computing slot VM_i to the selected task A (line 11). Lastly, update the data label *class*, and set the new value of the $class(d(A))$ to 0 (line 12).

For the task execution, if the input data of task A is not stored in the NVM, it is necessary to migrate data $d(A)$ from disk to NVM. Actually, this is guaranteed by the proposed *class* label and the operation $class(d(A)) \leftarrow 0$, and the migration will occur at the next free time for the data bus. We will show the details in the next subsection. Furthermore, most of the selected tasks have their input data stored in NVM. If some task needs to read from disk first, the execution must be longer, and this task can be excluded with high probability during the task selection within the selected job (line 7-10).

D. Labeling-based Data Replacement

The data bus is used to migrate data from disk to NVM, and each migration occupies multiple time-slots to accomplish the data transmission. When the bus is free to migrate new data, the *data replacement* need to answer the following questions.

- Q1: Is it necessary to migrate a new data to NVM?
- Q2: Which data should be migrated to NVM?
- Q3: Which data in NVM have to be replaced?

To answer the above questions and solve the problems, we propose a labeling-based data replacement algorithm, as shown in Alg. 3. The basic idea is to take NVM and disk as the first division and second division of the league, and there is one chance for the teams (data) to promote from

Algorithm 3 Label: $dataReplacment()$

Input: Mi and Kj: the data subsets,
δ: threshold (constant number)

1: $M \leftarrow merge(M0, M1, M2, M3, M4, M5)$;
2: $K \leftarrow merge(K0, K1, K2, K3, K4, K5)$;
3: **if** $M \neq \emptyset$ **then**
4: $D_C \leftarrow lastData(M)$;
5: **if** $K \neq \emptyset$ **then**
6: $D_H \leftarrow firstData(K)$;
7: **if** $class(D_H) = 2 \&\& |M2| \leq \delta$ **then**
8: **if** $K3 \neq \emptyset$ **then**
9: $D_H \leftarrow firstData(K3)$;
10: **if** $class(D_H) < class(D_C)$ **then**
11: $migrate(D_H, D_C)$;
12: **else if** $class(D_H) = 3$ && $class(D_C) = 3$ **then**
13: **if** $hotness(D_H) + \Delta H(D_H) > hotness(D_C) + \Delta H(D_C)$ **then**
14: $migrate(D_H, D_C)$;

second division (disk) to first division (NVM). The promotion rule is to compete between the best one from second division (disk), say D_H, and the worst one from first division (NVM), represented by D_C. The answer of Q1 is yes if D_H wins, else no. Obviously, D_H is the answer for Q2, and D_C is the answer for Q3.

Hence, the key issue is to choose D_H and D_C. According to the data location and data class, we can divide the data set into 12 subsets: $M0$, $M1$, $M2$, $M3$, $M4$, $M5$, $K0$, $K1$, $K2$, $K3$, $K4$, $K5$. For the subset Mi or Kj, we have

$$\forall D_r \in Mi, loc(D_r) = 0 \;\&\&\; class(D_r) = i,$$

$$\forall D_t \in Kj, loc(D_t) = 1 \;\&\&\; class(D_t) = j.$$

The symbol "M" means the data item is stored in NVM, while "K" refers to the data item stored in the disk. "i" or "j" is the value of *class* label.

For the data items in each subset, we further sort the items in decreasing order by $hotness(D_r) + \Delta(D_r)$. Here, we assume Mi and Kj are all sorted subsets. Furthermore, we merge the subsets Mi together, and sort the items by $class$ value (primary key) and $hotness$ value (secondary key) (line 1). It means that any item from subset $M1$ is prior to the items from $M2$, $M3$, $M4$, and $M5$. Similarly, we merge the subsets Kj together, and sort the items by the same method (line 2).

Next, we select the last item by $lastData(M)$ as the coldest data D_C in NVM (line 3–4). We can also select the first item from disk by $firstData(K)$ (line 6), but we take more consideration. Recall the definition of $class - 2$, it is the data wanted by the *actress batch job*, which has a longer deadline than the interactive jobs. Hence, the tasks in the actress batch job have less chance to be scheduled, and it is not a good idea to occupy too many storage slots. The storage slot may be useful for the data with label $class - 3$, the coming job may take it as input data. This is the reason why we make the decision in line 7 and line 8.

For the chosen D_H and D_C, the one with the smaller $class$ value win the competition. This is shown in line 10. If D_H win the competition, the bus will be occupied by migrating data D_H from disk to NVM (line 11). The function $migration(D_H, D_C)$ will change the location of the two data objects. The new locations should be: $loc(D_H) = 0$, and $loc(D_C) = 1$. If D_H and D_C have the same $class$ value (except $class = 3$), no migration is needed, which means it is unnecessary to replace the data with the same class. However, if the data objects are input data for interactive jobs, we prefer to place the data with a larger $hotness$ value to NVM. This is the core idea for hotness comparison (line 12-13). No migration occurs if D_H loses the competition.

V. Evaluation

A. Simulation Settings

There are two main algorithms for our approach: LST $jobScheduling(VM_i)$ and Label $dataReplacement()$. In addition, we also implement two other algorithms FCFS and FIFO. FCFS (first come first service) works for the job admission and job scheduling, as it will select the earliest job in the queue when computing resource is available. The basic idea for FIFO is to actively migrate the input data of jobs in queue, and replace the data that enters the NVM earliest and is not the input data for jobs in queue. This is a feasible replacement solution without any data features. Hence, we will implement three approaches: LST+Label, LST+FIFO, and FCFS+FIFO.

For the physical resource, the computing slot number m and the NVM slot number n are defined based on the typical PM configuration. Let $m = 64$, $n = 60$ be the typical setting, and change n from 30 to 100. For the utility settings, it should be a value about the job execution, and interactive jobs provide more unit utility than batch jobs. We set $\omega_0 = 5$, $\omega_1 = 3$, and $\omega_2 = -2$. We conduct the simulations in 3 cycles. For each cycle, there are 1000 time-slots. The jobs arrive at the PM at some specific time-slot, and there may be multiple job arrivals at one time-slot. There are more than 20000 jobs during the 3 cycles. For the $hotness$ value, we set $H_1 = H_2 = 2$. The data migration time from disk to NVM is set as 1 time-slot.

B. Result Analysis

Fig. 4 shows the results on utility with various n values. The x-axis is the value of n, while y-axis is the final utility. From the results we know that our approach (LST+Label) achieves the maximum when there are more than 30 NVM slots. In fact, the job acceptance rate is about 99.1%, which means the job workload is greater than the PM resource, and the job admission algorithm works to reject some jobs. The result implies that our approach can utilize the limited NVM resource efficiently even though there are not too many NVM slots. The utilities for LST+FIFO and FCFS+FIFO grow as n increases. This is because the data migration leads to longer job execution time, which produces negative utility or zero utility (reject job). The difference between LST+Label and

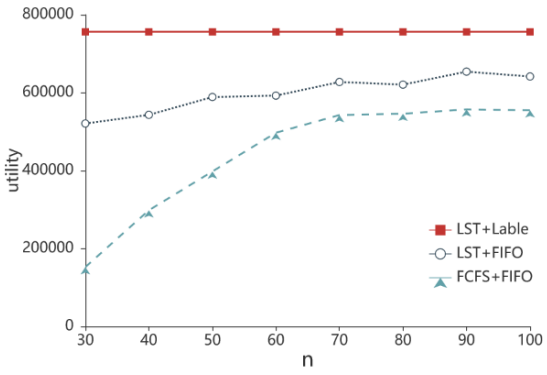
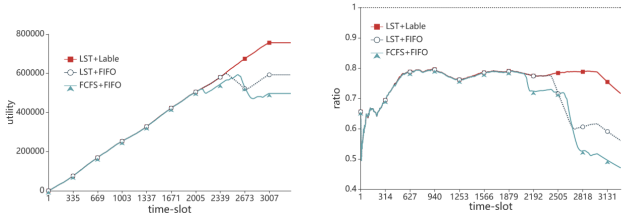
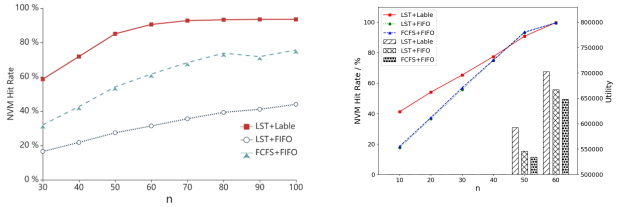


Fig. 4. Utility: $m = 64$



(a) Utility Value (b) Ratio

Fig. 5. Details of utilities: $m = 64$, $n = 60$



(a) NVM hit rate when $m = 64$ (b) NVM hit rate when $m = 56$

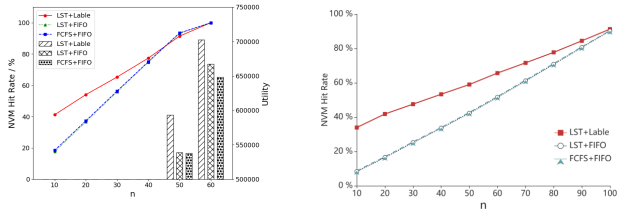
Fig. 6. NVM Hit Rates Analysis

LST+FIFO is whether to label the data, so this proves that data labeling system actually improves the performance.

We show the utility details as time when $n = 60$ in Fig. 5. The accumulated utility is shown in Fig. 5(a), where x-axis is the time-slot. The three approaches have nearly the same utility in the first half of duration. This is because the computing resource and NVM are sufficient to accomplish the jobs. However, when more jobs, especially the batch jobs, arrive at the PM, the data replacement policies show distinguished effects. The frequent data migrations cause longer job execution time, and lead to negative utility. This is the reason for utility decrement around the 2400 time-slot.

For each given time-slot t, we can calculate the maximized utility $max(t)$, which is the theoretical upper bound. It should be $max(t) = m \cdot \omega_0 \cdot t$, when all computer resources are used to execute the interactive job. Let the accumulated utility at t^{th} time-slot be $au(t)$. We show the ratio $\frac{au(t)}{max(t)}$ in Fig. 5(b). We can see the similar result mentioned above. At the beginning stage, most of the jobs are still running, which is the reason why the ratio is low.

NVM Hit Rate Analysis. One of the key issues for NVM is endurance enhancement. NVM hit rate affects the number



(a) Normal Job Workload (b) Double Job Workload

Fig. 7. Analysis on Interactive Jobs

of write operations significantly. Hence, we take NVM hit rate as another concern. Its value is equal to the rate that the task execution has the wanted input data in NVM. For the same job workload, we analyze the NVM hit rate when $m = 64$ and $m = 56$ in Fig. 6. The NVM hit rate increases as more NVM slots are available for all approaches. Though the utility of FCFS+FIFO is lower than LST+FIFO (Fig. 4), its NVM hit rate is larger. This is because the FCFS+FIFO approach rejects more jobs, and fewer data is needed, which brings both lower utility and greater NVM hit rate.

The core of our approach to gain more utility is to conduct data replacement actively. This brings greater utility and extra data migrations. The extra data migrations are more likely to lead to NVM hit, especially when there are more computing slots and fewer NVM slots. Hence, our approach gains larger utilities (Fig. 4 and Fig. 6(b)) and greater NVM hit rate (Fig. 6(a)). This is also the reason why the NVM hit rate of our approach is still high, but a bit less than other two approaches when $m = 56,\ n = 50$. The results imply that our approach could have significant performance improvement for large-scale jobs and data scheduling scenario.

Analysis on Interactive Jobs. For the big data analysis jobs, one of the most important objectives is to support the realtime interactive services. Hence, we also evaluate our approach for pure interactive jobs. Fig. 7 shows the result on VNM hit rate, which mainly affects the job response time. Our LST+Label approach has stable performance improvement as n increases. In Fig. 7(a), the weakness when $n = 50$ suffers from the same reason mentioned above, and the rate will be 1.0 when $n \geq 60$ because it is sufficient to store all input data. From the figure, we know that our approach still has the largest utility. There is more future work on the tradeoff between utility and NVM hit rate for interactive jobs.

The results in Fig. 7(b) show the case when we double the job workload within the same time duration. There are more than 100 different jobs with various input data. From the result, we know that the LST+Label approach shows significant improvement on NVM hit rate. This also demonstrates that our approach has stable and good performance for large-scale jobs and data scheduling problem.

VI. Conclusion

In this paper, we investigate the joint job and data scheduling problem for utility maximization in cloud data center with a hybrid storage system. We model the jobs, utilities and job execution time, and formulate the joint scheduling problem. We first introduce the labeling system to the scheduling problem, and propose a flexible labeling-based approach for joint job and data scheduling. We conduct extensive simulations, and the results show that the labeling method actually works. The proposed `labeling scheduler` has significant performance improvement by labeling method.

Acknowledgment

This work is supported in part by the National Key R&D Program of China under Grant 2018YFB1003900, in part by the National Natural Science Foundation of China under Grant 61802182, in part by the Jiangsu Natural Science Foundation under Grant BK20160813, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010.

[2] Y. Zhu, Y. Jiang, W. Wu, L. Ding, A. Teredesai, D. Li, and W. Lee, "Minimizing makespan and total completion time in mapreduce-like systems," in *IEEE INFOCOM*, 2014.

[3] J. Fu, J. Guo, E. W. Wong, and M. Zukerman, "Energy-efficient heuristics for job assignment in processor-sharing server farms," in *IEEE INFOCOM*, 2015.

[4] C. Chen, W. Wang, and B. Li, "Performance-aware fair scheduling: Exploiting demand elasticity of data analytics jobs," in *IEEE INFOCOM*, 2018.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *USENIX NSDI*, 2011.

[6] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *ACM SIGCOMM*, 2015.

[7] T. Hirofuchi and R. Takano, "RAMinate: Hypervisor-based virtualization for hybrid main memory systems," in *ACM SoCC*, 2016.

[8] W. Sun, N. Zhang, W. Lou, and Y. T. Hou, "When gene meets cloud: Enabling scalable and efficient range query on encrypted genomic data," in *IEEE INFOCOM*, 2017.

[9] L. Wang, C. Yang, and J. Wen., "Physical principles and current status of emerging non-volatile solid state memories," *Electronic Materials Letters*, vol. 11, no. 4, pp. 505–543, 2015.

[10] Y. Wang, K. Li, J. Zhang, and K. Li, "Energy optimization for data allocation with hybrid SRAM+NVM SPM," *IEEE Transactions on Circuits and Systems*, vol. 65, no. 99, pp. 307–318, 2018.

[11] H. Liu, Y. Chen, and X. Liao, "Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures," in *International Conference on Supercomputing*, 2017.

[12] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, and C. Li, "Extending the lifetime of NVMs with compression," in *IEEE Design, Automation and Test in Europe Conference*, 2018.

[13] Z. Zheng and N. B. Shroff, "Online multi-resource allocation for deadline sensitive jobs with partial values in the cloud," in *IEEE INFOCOM*, 2016.

[14] Q. Xie, A. Yekkehkhany, and Y. Lu, "Scheduling with multi-level data locality: Throughput and heavy-traffic optimality," in *INFOCOM*, 2016.

[15] Y. Bao and S. Wang, "Labeled von Neumann architecture for software-defined cloud," *Journal of Computer Science and Technology*, vol. 32, no. 2, pp. 219–223, 2017.

[16] X. Li, J. Wu, Z. Qian, S. Tang, and S. Lu, "Towards location-aware joint job and data assignment in cloud data centers with nvm," in *IPCCC*, 2017.

[17] X. Li, J. Wu, S. Tang, and S. Lu, "Let's stay together: Towards traffic aware virtual machine placement in data centers," in *IEEE INFOCOM*, 2014.