The Journal of Systems and Software 104 (2015) 141–151

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Scheduling parallel jobs with tentative runs and consolidation in the cloud

Xiaocheng Liu*, Yabing Zha, Quanjun Yin, Yong Peng, Long Qin

College of Information System and Management, National University of Defense Technology, Changsha City, Hunan Province 4100073, China

ARTICLE INFO

Article history:
Received 13 July 2014
Revised 7 March 2015
Accepted 7 March 2015
Available online 12 March 2015

Keywords:
Cloud computing
Parallel job scheduling
Workload consolidation

ABSTRACT

Since the success of cloud computing, more and more high performance computing parallel applications run in the cloud. Carefully scheduling parallel jobs is essential for cloud providers to maintain their quality of service. Existing parallel job scheduling mechanisms do not take the parallel workload consolidation into account to improve the scheduling performance. In this paper, after introducing a prioritized two-tier virtual machines architecture for parallel workload consolidation, we propose a consolidation-based parallel job scheduling algorithm. The algorithm employs tentative run and workload consolidation under such a two-tier virtual machines architecture to enhance the popular FCFS algorithm. Extensive experiments on well-known traces show that our algorithm significantly outperforms FCFS, and it can even produce comparable performance to the runtime-estimation-based EASY algorithm, though it does not require users to provide runtime estimation of the job. Moreover, our algorithm allows inaccurate CPU usage estimation and only requires trivial modification on FCFS. It is effective and robust for scheduling parallel workload in the cloud.

1. Introduction

Recently, it is common for high performance computing (HPC) parallel applications to run in the cloud mainly due to the easy-to-use and cost-effective run way provided by the cloud computing paradigm. In addition, infrastructure as a service (IaaS) providers (such as Amazon EC2) now provide HPC instances[1] to cater for HPC parallel applications. For a cloud provider, carefully scheduling parallel jobs submitted by users is essential to guarantee its quality of service (QoS) (Bui et al., 2010; Zhu et al., 2011).

First-come first-serve (FCFS) (Schwiegelshohn and Yahyapour, 1998) and Extensible Argonne Scheduling sYstem (EASY) (Lindsay et al., 2013; Mu'alem and Feitelson, 2001) are the two most widely used parallel job scheduling methods, they are both easy-to-implement and fair (Etsion and Tsafrir, 2005). Users continuously submit their jobs into the waiting queue of the resource allocator, FCFS/EASY (or other scheduling algorithms) assign them into physical processors for run, as shown in Fig. 1(a). EASY relies on job's process number and the estimation of job's runtime, however the runtime of a job running in a cloud cannot be well estimated because:

1. Ubiquitous random factors within a parallel job make the range of its runtime extremely wide.
2. The capacity of the computing resource is hidden and thus unknown to users, hence, the runtime estimation is beyond the users.

For EASY, over-estimation may lead to a long wait time and possibly to excessive CPU quota loss, while under-estimation may lead to a risk that the job will be killed before its termination (Mu'alem and Feitelson, 2001). FCFS only needs job's process number and does not require job's runtime information to make scheduling decisions, but it suffers from severe processor fragmentation problem. If currently free processors cannot meet the requirements of the head job, these free processors therefore remain idle. Moreover, both FCFS and EASY do not consider the idle CPU cycles caused by parallel jobs themselves. Parallel jobs often involve computing, communication and synchronization phases. A process in a parallel job may frequently wait for the data from other processes, during waiting, the CPU utilization is low.

In this paper, we design an algorithm named aggressive consolidation-based first-come first-serve (ACFCFS) which intends to achieve the following goals:

1. Preserve the FCFS order of jobs when processors are available.
2. Do not need job's runtime estimation.
3. Do not need the support of job migration, which migrates part of (or all) the processes of a job from their original processors to new ones, either through a static way or a dynamic way.
4. Can effectively use the idle CPU cycles caused by the parallel jobs themselves.
5. Produce comparable performance to EASY.

* Corresponding author. Tel.: +86 8 4573 389 8021.
E-mail address: nudt200203012007@163.com (X. Liu).

[1] Amazon aws. http://aws.amazon.com/ .

http://dx.doi.org/10.1016/j.jss.2015.03.007
0164-1212/

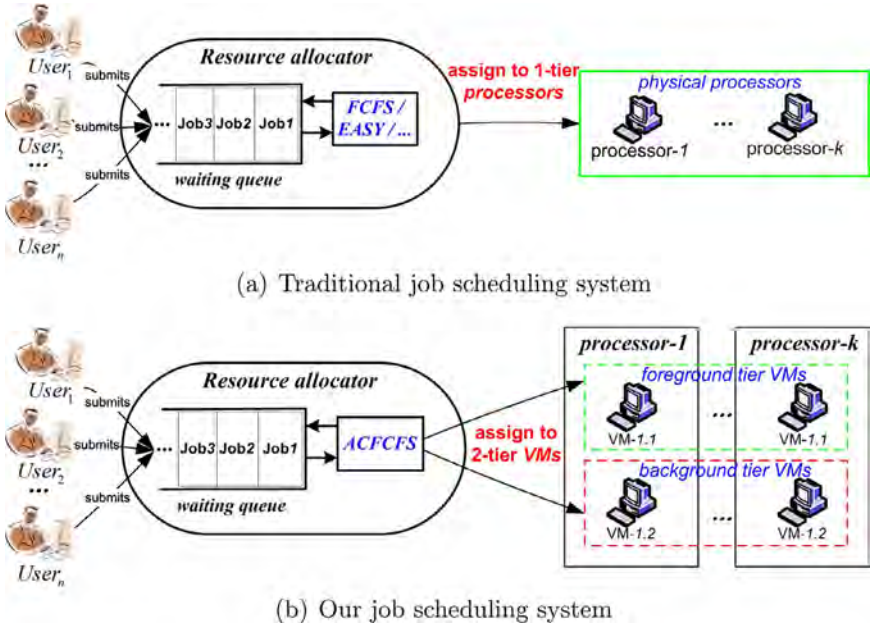


Fig. 1. Comparison of our parallel job scheduling algorithm and traditional ones.

ACFCFS can be characterized in twofold:

1. It employs tentative run rather than job migration to alleviate the pain of processor fragmentation. Tentative run is a try to find chances to make use of the idle CPU cycles, it is a type of run which should be killed later once it violates the scheduling rules.
2. It uses parallel workload consolidation to make use of the idle CPU cycles caused by parallel jobs themselves. Parallel workload consolidation combines parallel workloads of different servers on a set of target servers to improve the utilization of computing resources. We proposed a prioritized two-tier processor partition method to organize virtual machines (VMs) in Liu et al. (2013) for parallel workload consolidation in this paper. The two-tier VMs have different CPU priorities, the one with high CPU priority is named foreground VM and the one with low CPU priority is named background VM. A foreground VM and a background VM are pinned to one physical processor. In this prioritized two-tier VMs architecture, through experiments on collocating VMs running different parallel jobs, we found that there is little performance degradation (less than 4%) of jobs running in foreground VMs and meanwhile jobs running in background VMs can effectively make use of the remaining compute capacity left from the foreground VMs.

Based on the tentative run technique and the parallel workload consolidation method supported by the prioritized two-tier VMs architecture, our parallel job scheduling system is depicted by Fig. 1(b). ACFCFS needs users to specify job's process number and CPU utilization estimation of job's processe(s), the scheduling of ACFCFS happens in the two tier VMs. More specifically, FCFS and SMallest-Job-First (SMJF) work together to schedule jobs onto the foreground tier VMs, and SMJF is employed to let job of lower parallelism (with less parallel processes) run first in the background tier. Jobs scheduled by SMJF are for tentative runs while jobs scheduled by FCFS are for formal runs.

Our evaluation results show that ACFCFS significantly outperforms FCFS and produces comparable performance to the runtime-estimation-based EASY algorithm on well-known parallel workloads. In addition, our algorithm allows inaccurate CPU usage estimation and requires trivial modification on FCFS. It is effective and robust for scheduling parallel workload in the cloud.

The remainder of this paper is organized as follows: Section 2 reviews the related work. Section 3 presents our consolidation-based algorithms in the two-tier VMs architecture. Section 4 gives the evaluation results and Section 5 concludes this paper.

2. Related work

2.1. Traditional parallel job scheduling methods

FCFS, EASY and gang are three widely discussed parallel job scheduling algorithms in the literature (Etsion and Tsafrir, 2005). FCFS (Schwiegelshohn and Yahyapour, 1998) is the most basic method, it does not require job's runtime information to make scheduling decisions, but it suffers from severe processor fragmentation. EASY (Lifka, 1995) was developed for IBM SP1 which allows short/small jobs to use idle processors when the job arrives earlier than them but does not have enough number of processors to run. EASY makes reservation for the head job in the FCFS queue to protect the fare of scheduling. Gang scheduling (Feitelson and Jettee, 1997; Wiseman and Feitelson, 2003) allows resource sharing among multiple parallel jobs. The computing capacity of a processor is divided into time slices for sharing among the processes of jobs. The gang scheduling algorithm manages to make all the processes of a job progress together so that one process will not be in sleeping state when another process needs to communicate with it. Gang scheduling has not been widely used due to its limitations in practice (Etsion and Tsafrir, 2005). Moreover, FCFS, EASY and gang all fail to address the utilization degradation problem caused by the parallelization of parallel jobs.

2.2. Parallel job scheduling with tentative run

Thebe et al. propose a typical case which extends base scheduler (such as FCFS) by trial timed-test run (Thebe et al., 2009). In the extended scheduler, all jobs should experience a timed run before being committed for normal run. Test run can identify failing jobs more quickly which will die prematurely either because of bugs or because their execution environment changes. It is very effective for workloads which contain many such kind of jobs or many really short jobs. But the trial timed test run needs to specify how long the test run is, and in addition it brings limited performance improvement and even produces performance degradation (for example, –181% in a case). Similar work on the combination of test run and existing scheduler has been discussed by Perkovic and Keleher (2000),

Table 1
Statistic on real traces.

Center name	Percentage of jobs with varying size in short jobs								
	=1(%)	=2(%)	(2, 4](%)	(4, 8](%)	(8, 16](%)	(16, 32](%)	(32, 64](%)	(64, 128](%)	>128(%)
LANL	0.0	0.0	0.0	0.0	0.0	63.6	14.0	13.4	9.0
CTC	36.7	7.5	13.4	14.6	13.5	8.8	4.2	0.9	0.4
KTH	30.1	14.5	19.3	16.0	11.9	4.7	2.6	0.9	0.0
SDSC	27.6	15.6	14.4	15.4	13.2	8.5	4.7	0.6	0.0

Snell et al. (2002), Lawson and Smirni (2002) and Lawson et al. (2002). Unlike to these works, tentative run in this paper is the run that may be killed during its execution, not for a certain time.

2.3. Parallel job scheduling in virtualized environments

In cloud computing, a common way of using idle CPU cycles is to do it through server consolidation. At this stage, most of the existing efforts (Sonmez et al., 2009; Speitkamp and Bichler, 2010) consider the scheduling for sequential applications (such as web service) or tasks. Some other efforts focus on parallel jobs in the cloud but are from the viewpoint of space-sharing scheduling scheme nor do not take consolidation into account (Moschakis and Karatza, 2012; Nicod et al., 2011; Sodan, 2009). Moschakis and Karatza (2012) study the performance of a distributed cloud computing model, based on the EC2 architecture that implements a gang scheduling scheme. Our method differs from their work in: (1) our work focuses on the space-sharing scheduling scheme rather than the time-sharing scheme; (2) our work is from the provider's viewpoint but theirs is from the tenant's point of view. Sodan (2009) and Nicod et al. (2011) discuss a method of reshaping jobs with the aid of virtual machine technologies. Unlike our consolidation-based scheduling, their work is job molding based scheduling. Our previous work in Liu et al. (2013) takes consolidation into account for scheduling, but it needs the help of job migration which is not applicable in some cases.

3. Scheduling algorithms

In this section, we describe our tentative run and consolidation based parallel job scheduling algorithms. We first introduce our parallel workload consolidation method devised in Liu et al. (2013), then a basic algorithm and a refined algorithm are discussed. The refined one is an improvement of the basic one.

3.1. A two-tier VMs architecture for priority-based parallel workload consolidation

Hardware virtualization technology provides an easy-to-use way for parallel workloads consolidation in cloud computing. For parallel workloads, in order to improve the CPU utilization of their host processors, we here partition each processor into two-tier[2] VMs by pinning two virtual CPUs (VCPUs) on the processor and then allocating these two VCPUs to the two VMs. For a parallel job, its execution time running in either tier VMs is stretched if there exists no CPU priority control on the VMs, because of the context switch between the two tiers. Reducing the number of context switches can straightforwardly improve the utilization of host processors, thus we assign the VMs on one tier with high CPU priority (say, assigning a weight of *10,000* through the CrediteScheduler[3] of Xen (Barham et al., 2003)) and assign the VMs on the other tier with low CPU priority (by assigning a weight of *1*). The tier with high CPU priority is called foreground (***fg***) tier and the one with low CPU priority is called background (***bg***) tier. In this setting, the number of context switches can be reduced significantly because the background VM only uses CPU cycles when its corresponding foreground VM is idle. Under this prioritized two-tier VMs architecture, experiments in our small cluster conclude that (Liu et al., 2013):

1. The average performance loss of jobs running in the foreground tier is between 0.5% and 4% compared to those running in the processors exclusively (one-tier VM), we simply model the loss as a uniform distribution.
2. When a foreground VM runs a job with higher CPU utilization than 96%, collocating a VM to run in the background tier does not benefit the job running in it due to that context switch incurs overhead and the background VM has little chance to run.
3. When a foreground VM runs a job with low CPU utilization, the job running in the collocated background VM can get significant portion of physical resources to run. For a single-process background job, the utilization of the idle CPU cycles is between 80% and 100% and roughly follows uniform distribution. For a multi-processes background job, the value is between 20% and 80%, and can be modelled by a normal distribution with $\mu = 0.43$ and $\sigma = 0.14$.

Based on the observation upon this two-tier VMs architecture, we discuss our scheduling algorithms in the following sections.

3.2. Basic algorithm

3.2.1. Algorithm description

Our basic algorithm under the two-tier VMs architecture is named conservative consolidation-based first-come first-serve (CCFCFS). We call CCFCFS "conservative" to distinguish it from the "aggressive" version described later (in Section 3.3). In CCFCFS, scheduling happens in two tiers and the concept is:

1. Use FCFS to schedule runnable jobs onto the foreground VMs and use SMallest Job First (SMJF) to deploy all possible jobs onto the background VMs for tentative runs.
2. To preserve the properties of FCFS, the foreground scheduling considers both jobs in the waiting queue (a FCFS queue) and jobs running in the background tier when making scheduling decisions.
3. When the foreground scheduling picks up a job that is running in the background VMs to run, there are two situations:
   (a) if ***the collocated foreground VMs' corresponding background VMs are all idle***: ***change*** the priorities of these background VMs to high while ***changing*** the priorities of the corresponding foreground VMs to low.
   (b) otherwise: ***kill*** the job in the background VMs and restart it from the beginning on the newly allocated foreground VMs.

In order to make use of the capacity of the background VMs, we use SMJF to schedule jobs to background VMs. This increases the chance that a job finishes in the background VMs because of:

1. Small jobs are very likely to be short jobs. This is primary due to long jobs are often executed in high parallelism to reduce the execution time. Table 1 gives the percentage of jobs with varying

[2] The method can extend to "k-tier ($k > 2$)" VMs, this paper only takes "two" as an example.

[3] Creditscheduler. http://wiki.xen.org/xenwiki/CreditScheduler.

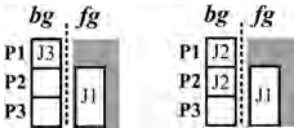


Fig. 2. Example of the observation that small jobs are very likely to be short jobs.

size in short jobs[4] in four famous supercomputer centers. The four centers are Los Alamos National Lab (LANL), Cornell Theory Center (CTC), Swedish Royal Institute of Technology (KTH) and San Diego Supercomputer Center (SDSC) respectively. Statistics in Table 1 validate this phenomena.

2. Running small jobs in background VMs can use the computing resource more effectively. Fig. 2 illustrates an example of this situation. In Fig. 2, we assume that the CPU usage of processes in J1 is 70% and the CPU usages of processes in J2 and J3 are both 100%.
3. It is easier to switch small jobs from the background tier to the foreground tier, because they are more likely to qualify the criteria "***the collocated foreground VMs' corresponding background VMs are all idle***".

In addition, CCFCFS only dispatches a job to run in a background VM if the utilization of corresponding foreground VM is lower than a threshold (96% according to Section 3.1). The foreground VM utilization can be obtained from the profile of foreground jobs, or from the runtime monitoring system in a datacenter.

The algorithm of CCFCFS consists of two sub-algorithms which handle job arrival and job departure events. When a job arrives or a job finishes in the foreground VMs, Algorithm 1 is invoked to schedule runnable jobs to both the foreground tier and the background tier. When a job finishes in the background VMs, Algorithm 2 is invoked to schedule runnable jobs to the background tier.

3.2.2. Example

Fig. 3 (b) illustrates an instance of CCFCFS and the corresponding example of FCFS is given in Fig. 3(a). Let there be five processors (P1–5) and seven jobs (J1–7) initially. Each job is denoted by (n, t), where n is the number of processors required and t is the execution time. We here assume that a job with lower index arrives earlier than a job with bigger index. Each processor has two tier VMs denoted as *fg* and *bg* in Fig. 3. For illustration convenience here, we assume that the process in a single-process job incurs a CPU usage of 100% and each process within a multi-processes job involves a CPU usage less than the utilization threshold (96% according to Section 3.1).

At time 0, J1 is placed onto the foreground VMs of P4–5 according to FCFS; J3–5 is deployed onto the background VMs of P1–5 according to SMJF. We use a simple process to collocate a background VM with a foreground VM, as shown in the ***JobDispatch*** function in Algorithm 2. This process matches the background VM (that is likely to incur high processor utilization) to the foreground VM (that is likely to incur low processor utilization).

At time 5 (we assume J5 advances 3 time units during time 0–5 here), J1 and J5 depart from the system. J2 is scheduled by FCFS onto the foreground VMs of P2–5, and J7 is scheduled by SMJF onto the background VMs of P4–5.

At time 10, J2 and J4 depart from the system (although J4 runs in the background tier during its lifetime, it actual occupies P1 exclusively because its foreground VM is always idle during its execution. Similarly, J3 runs in P2–3 exclusively during time 0–5), J3 and J6 are picked up by FCFS to run in the foreground tier. For J3, it can be switched from the background VMs to the foreground VMs at its original processors by ***swapping*** the CPU priorities.

[4] We here call a job is short when its runtime is less than the average runtime of the workload.

Algorithm 1. CCFCFS – **job arrival**/***fg* job departure** procedure.

```
input : J_q: a list of jobs waiting in the queue;
        J_bg: a list of jobs running in the bg tier;
1  begin
2      /*Find jobs which will be deployed onto the fg tier*/
3      J_Wfg ← SelectJobs(J_q,J_bg);
4      /*Deploy the selected jobs onto the fg tier*/
5      DeploySelectedJobs(J_Wfg,J_q);
6      /*Try to deploy jobs onto the bg tier*/
7      TryToRunMoreJobs('BG',J_q); /*See this function in
       Algorithm 2*/
8  function SelectJobs(J_q,J_bg)
9  begin
10     J_Wfg = null, a job list which will be deployed onto the
       fg tier;
11     J_c ← J_q ∪ J_bg, the merged job list of J_q and J_bg;
12     N^f_idle ← the number of current idle VMs in the fg tier;
13     /*Select jobs according to FCFS*/
14     sort J_c in in ascending order of their arrival time;
15     N_j ← the number of processes in the first job of J_c;
16     while N_j ≤ N^f_idle do
17         N^f_idle = N^f_idle − N_j;
18         insert j into J_Wfg;
19         N_j ← the number of processes in the next job;
20     return J_Wfg;
21 function DeploySelectedJobs(J_Wfg,J_q)
22 begin
23     for each job j in J_Wfg do
24         if j is from the bg tier and all its fg VMs are idle
           then
25             swap the CPU priorities of its bg VMs and fg
               VMs;
26             remove j from J_Wfg;
27     for each job j in J_Wfg do
28         if j is from the bg tier then
29             kill j;
30             JobDispatch(FG, j);
31         else
32             JobDispatch(FG, j);
33             remove j from J_q;
```

At time 15, J3 departs from the system, J7 is ***killed*** from the background VMs of its original processors (P4–5) and then restarted from the beginning at the foreground VMs of the newly allocated processors (P2–3).

3.3. Refined algorithm

In CCFCFS, the scheduling strategy in the foreground tier is FCFS. This manner leads to severe foreground VM fragmentation. Improving the utilization of the foreground VMs is the key in design of algorithms under the two-tier VMs architecture. This section devises an aggressive version of CCFCFS, namely, the aggressive consolidation-based first-come first-serve (ACFCFS).

3.3.1. Algorithm description

To handle a background job departure, ACFCFS behaves the same as CCFCFS, as shown in Algorithm 2. For the procedure of a job arrival & a foreground job departure, ACFCFS is more aggressive than CCFCFS as shown in Algorithm 3. The aggressiveness of ACFCFS is shown in line 4 (the invoking of the ***TryToRunMoreJobs*** function) of

Algorithm 2. CCFCFS & ACFCFS – ***bg* job depature** handling.

```
   input : J_q: a list of jobs waiting in the queue;
1  begin
2      TryToRunMoreJobs('BG',J_q);
3  function TryToRunMoreJobs(flag,J_q)
4  begin
5      sort the jobs of J_q in ascending order of their size;
6      /*this case is only usded in AFCFS*/
7      if flag == FG then
8          for each job j in the sorted J_q do
9              N_j ← the number of processes in j;
10             N^f_idle ← the number of current idle VMs in the
               fg tier;
11             if N_j ≤ N^f_idle then
12                 JobDispatch('FG',j);
13     else
14         for each job j in the sorted J_q do
15             N_j ← the number of processes in j;
16             N^b_idle ← the number of current idle VMs in the
               bg tier;
17             if N_j ≤ N^b_idle then
18                 JobDispatch('BG',j);
19 function JobDispatch(flag, j)
20 begin
21     sort parallel processes of j in descending order of their
       CPU usage;
22     if flag == FG then
23         p ← sorted idle fg VMs in ascending order of their
           bg VM load;
24         for each process j_i in the sorted list do
25             place j_i to p_i and run j_i in the fg VM;
26     else
27         p ← sorted idle bg VMs in ascending order of their
           fg VM load;
28         for each process j_i do
29             place j_i to p_i and run j_i in the bg VM;
```

Algorithm 3. ACFCFS – **job arrival/*fg* job departure** procedure.

```
   input : J_q: a list of jobs waiting in the queue;
           J_bg: a list of jobs running in the bg tier;
1  begin
2      J_Wfg ← SelectJobs_Refined(J_q,J_bg);
3      DeploySelectedJobs(J_Wfg,J_q);
4      TryToRunMoreJobs('FG',J_q); //try to run more jobs to the fg
       tier
5      TryToRunMoreJobs('BG',J_q); //try to run jobs to the bg tier
6  function SelectJobs_Refined(J_q ,J_bg)
7  begin
8      J_Wfg = null, jobs will be deployed to the fg tier;
9      J_Efg = null, jobs will be evicted from the fg tier;
10     sort J_q ∪ J_bg according to job arrival time;
11     N^f_idle ← current idle fg VMs number;
12     for each job j in J_q ∪ J_bg do
13         N_j ← the number of processes in j;
14         if N_j ≤ N^f_idle then
15             N^f_idle = N^f_idle − N_j and then insert j into J_Wfg;
16         else
17             J_pt ← jobs now running in the fg tier but arrive later
               than j;
18             N^f_pre ← number of fg VMs occupied by jobs in J_pt;
19             if N_j ≤ (N^f_pre + N^f_idle) then
20                 add the number of fg VMs running jobs in J_pt
                   into N^f_idle according to descending order of their
                   arrival time until N^f_idle is not less than N_j, and
                   insert them into J_Efg ;
21                 N^f_idle = N^f_idle − N_j;
22             else
23                 break;
24     RefineEvictedJobs(J_Efg, N^f_idle) and then kill all the jobs in
       J_Efg;
25     return J_Wfg;
26 function RefineEvictedJobs(J_Efg ,N^f_idle)
27 begin
28     if N^f_idle > 0 then
29         sort J_Efg in ascending order of their size;
30         for each job j in J_Efg do
31             N_j ← the number of processes in j;
32             if N_j ≤ N^f_idle then
33                 N^f_idle = N^f_idle- N_j and then remove j from J_Efg;
34     for each job j in J_Efg do
35         if all the bg VMs of j are now idle then
36             remove j from J_Efg, swap the CPU priorities of its bg
               VMs and fg VMs;
```

Algorithm 3, which makes tentative run not only happen in the background tier but also in the foreground tier.

Tentative run in the foreground tier triggers job preemption in the foreground tier, which may violate the rule of FCFS. In runtime, ACFCFS may evict some preemptive jobs currently running in the foreground tier to make room for the preempted jobs to avoid this violation. The procedure of the job selection (accompanied by job eviction) is implemented by the ***SelectJobs_Refined*** function in Algorithm 3.

3.3.2. Example

Fig. 3 (c) gives an example of ACFCFS.

At time 0, after placing J1 onto the foreground VMs of P4–5 according to FCFS, J4 and J3 are deployed onto the foreground VMs of P1–3 for tentative runs (their run may be killed or switched to the background tier in future) by SMJF. Then J5 and J7 are assigned onto the background VMs of P2–5 by SMJF.

At time 5, J1 and J5 depart from the system (we here also assume J5 advances 3 time units during time 0–5). As the total number of foreground VMs (5) that are either idle or occupied by the jobs (J3–4) which arrive later than J2 is greater than the process number of J2 (4). So ACFCFS starts to evict some jobs to make room for J2. First, J4 and then J3 are initially selected as candidate jobs to be evicted (according to lines 8–23 in Algorithm 3). Then J4 remains in its original position without being evicted because there is one extra foreground VM left (according to line 28–33 in the ***RefineEvictedJobs*** function); J3 is switched (by swapping the CPU priorities of its foreground VMs and background VMs) to the background VMs of its original processors as these background VMs are all idle at this time (according to line 34–36 in the ***RefineEvictedJobs*** function). J2 is then deployed onto the foreground VMs of P2–5 at last.

At time 10, J2, J4 and J7 depart from the system (we assume J7 advances 5 time units during time 0–10 here), J3 is switched from the background tier to the foreground tier by CPU priorities swapping and J6 is deployed to the foreground tier from the waiting queue. Then J3 and J6 depart from the system at time 15 and time 20 respectively.

4. Evaluation

4.1. Evaluation methodology

Our algorithms are evaluated by trace-driven simulation (Liu et al., 2013; Mu'alem and Feitelson, 2001), traces here include four real

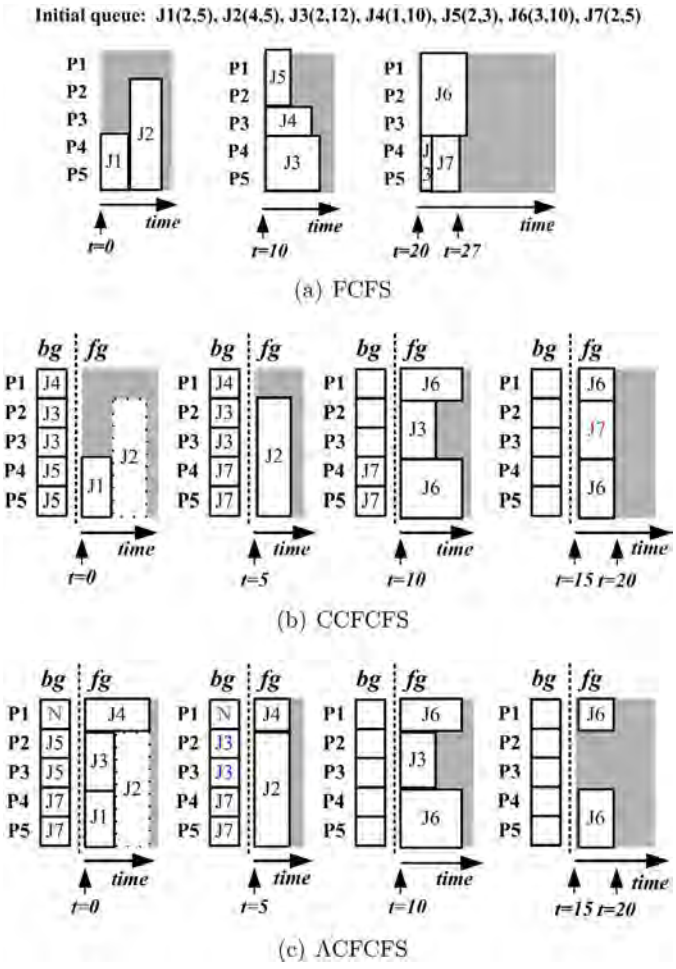


Fig. 3. Examples of FCFS, CCFCFS and ACFCFS.

traces and two synthetic traces. The four real traces are LANL-CM5 (LANL), TC-SP2 (CTC), KTH-SP2 (KTH) and SDSC-SP2 (SDSC), and the system load of them are 0.74, 0.84, 0.69 and 0.83 respectively. The two synthetic traces are generated by Feitelson workload model (denoted by *FWload*, 350,000 jobs) (Feitelson, 1996) and Jann workload model (*JWload*, 100,000 jobs) (Jann et al., 1997). EASY is also evaluated in this paper for comparison. Since EASY needs the estimation of job's runtime to make decisions, for real traces we use the actual runtime as the job's estimated runtime when the estimated runtime is less than the actual runtime. For synthetic traces we use Tsafrir model (Tsafrir et al., 2005; Tsafrir) to add the estimated runtime for each job.[5] As all traces do not contain the CPU usage information of processes, we assign a CPU usage of 100% to a process if it is from a single-process job, otherwise, a random number between 40% and 100% is assigned as the CPU usage.

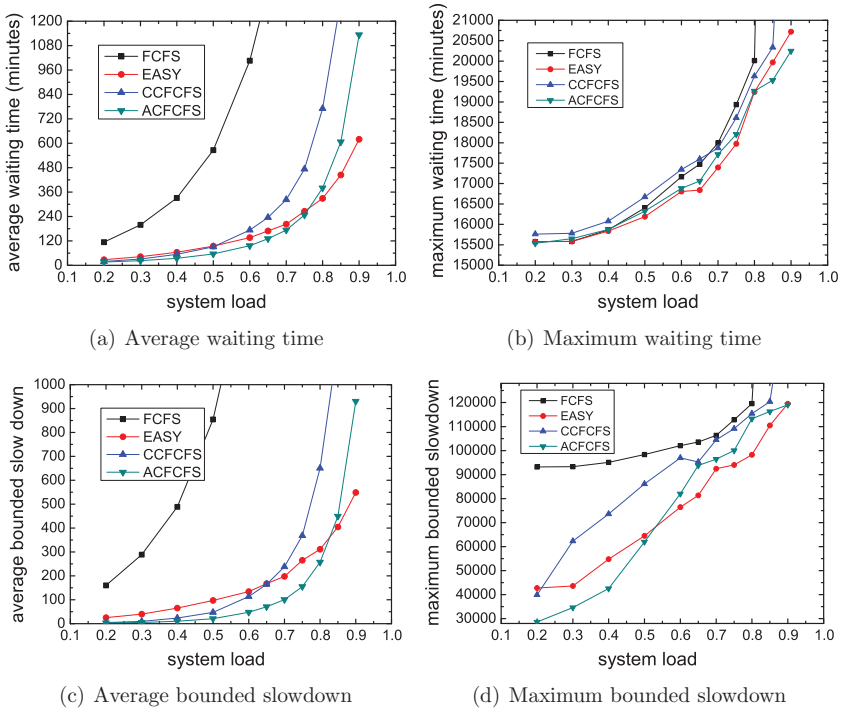
In the two-tier VMs architecture, the progress of a process running in the foreground VM and the background VM can be calculated by the following equations respectively:

$$t = \begin{cases} T & \text{if its } bg \text{ VMs are all idle} \\ T \cdot (1 - \text{loss}) & \text{otherwise} \end{cases}$$

$$t = \begin{cases} T & \text{if its } fg \text{ VMs are all idle} \\ T \cdot \text{eff} & \text{else if } \text{CPU}_i \geq \text{CPU}_r \\ T \cdot \text{eff} \cdot \dfrac{\text{CPU}_i}{\text{CPU}_r} & \text{otherwise.} \end{cases}$$

Where, T is the length of a time slice, denoting the progress of a process running on a dedicated processor in a time slice; *loss* is the performance degradation of jobs running in the foreground tier due to the context switch; CPU_r is the CPU utilization of the background process on a dedicated processor; CPU_i is the portion of unused CPU cycles in the processor; eff is a variable between 0 and 1, representing how much time in a time slice effectively contributes to the progress of the process. According to Section 3.1, *loss* is a random number

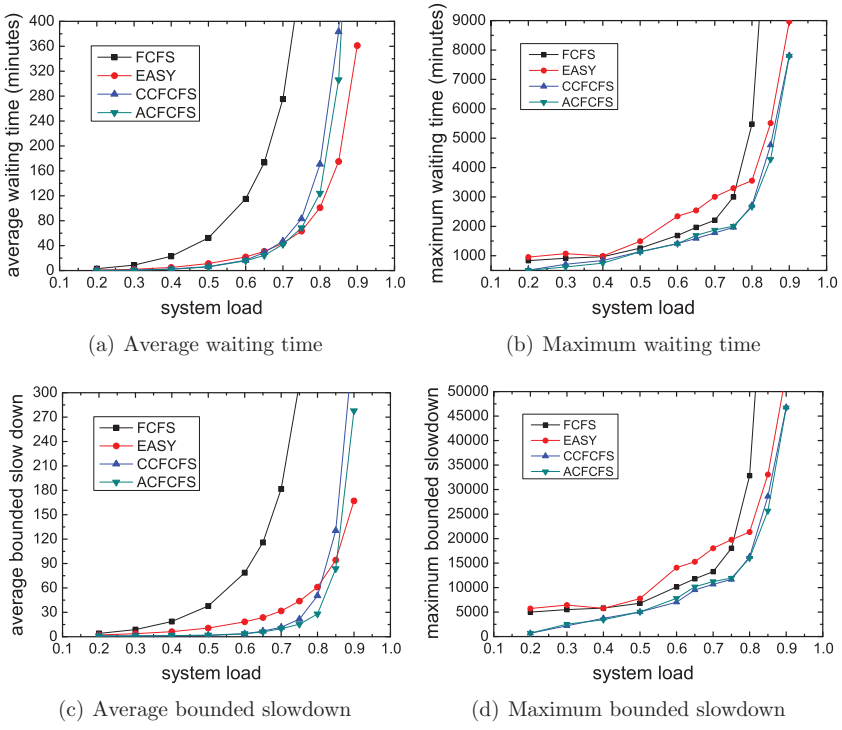
[5] All the traces and models can be downloaded from the parallel workload archive (Feitelson).



(a) Average waiting time (b) Maximum waiting time

(c) Average bounded slowdown (d) Maximum bounded slowdown

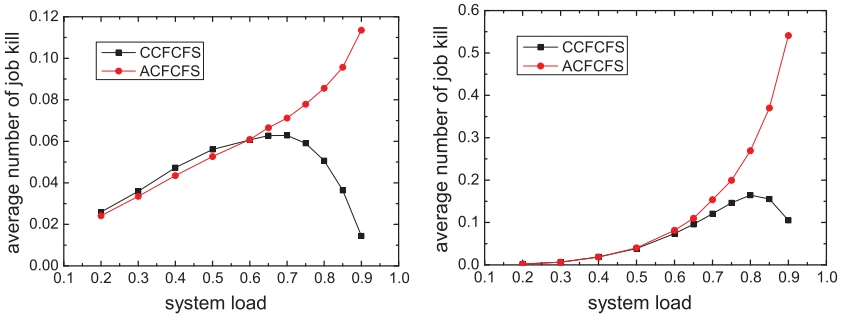
Fig. 4. Performance for *FWload*.



(a) Average waiting time (b) Maximum waiting time

(c) Average bounded slowdown (d) Maximum bounded slowdown

Fig. 5. Performance for *JWload*.



(a) *FWload* (b) *JWload*

Fig. 6. Average number of job kill.

between 0.5% and 4% (uniformly distributed), eff is a random number between 0.8 and 1.0 (uniformly distributed) in the case of a single-process job and a random number between 0.2 and 0.8 (normally distributed with $\mu = 0.43$ and $\sigma = 0.14$) in other cases.

We use the *average waiting time* and the *average bounded slowdown* as metrics for performance evaluation. The waiting time of a job is defined as $w_t = t_f - t_a - t_e$, and the bounded slowdown of a job is defined as $sd_b = (t_f - t_a)/\max(\Gamma, t_e)$, where t_f is the finish time of the job, t_a is the arrival time of the job, t_e is the execution time of the job. Compared with slowdown, the bounded slowdown metric is less affected by very short jobs as it contains a minimal execution time element Γ. According to Feitelson et al. (1997), we set Γ to 10 s. To show more aspects of the scheduling results, we further use the *maximum waiting time* and the *maximum bounded slowdown* as metrics in this paper.

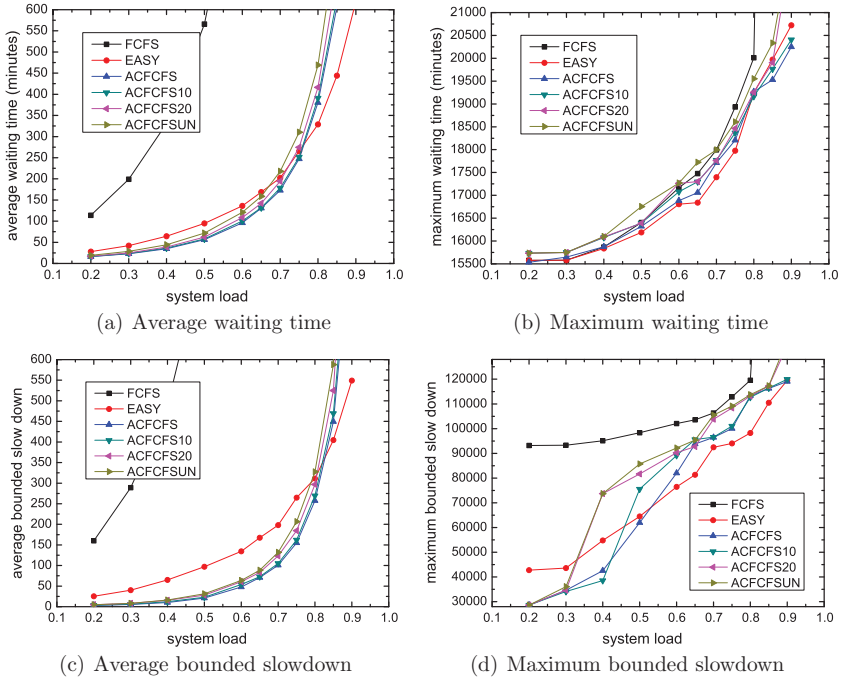
4.2. Evaluation results

Figs. 4, 5 and 6 show the performance of our algorithms for the workload models, results in columns (labeled in FCFS/EASY/CCFCFS/ACFCFS) of Table 2 gives the performance for the real traces. We have the following observations from the results:

1. Both CCFCFS and ACFCFS produce significant better performance than FCFS, whether from the viewpoint of the average value or the viewpoint of the maximum value. ACFCFS beats CCFCFS to much extend by paying the price of more job killings (especially in higher load). Compared with FCFS, ACFCFS produces an average of 94.2% improvement on the average waiting time and an average of 97.5% improvement on the average bounded slowdown for *FWload*. The improvements for *JWload* are 92.4% and 92.9% respectively.
2. For the average metrics as well as the maximum metrics, ACFCFS can produce comparable performance to EASY especially on average bounded slowdown. For the two workload models, EASY can outperform ACFCFS on the average waiting time/average bounded slowdown only when the system load is bigger than 0.8/0.85.
3. *FWload* benefits more from ACFCFS than *JWload* because *JWload* workload contains about 40% (Jann et al., 1997) single-process jobs but the percentage in *FWload* is less than 18% (Feitelson, 1996). The high percentage of single-process jobs in *JWload* reduces the

Table 2
Performance for real traces. ACFCFSUN denotes that the algorithm does not use CPU usage information at all. ACFCFS100 means that all the processes involved consume the whole CPU cycles of their host processors.

Trace	FCFS	EASY		CCFCFS		ACFCFS		ACFCFSUN		ACFCFS100	
		Value	Imp(%)	Value	Imp(%)	Value	Imp(%)	Value	Imp(%)	Value	Imp(%)
Average waiting time (min)											
LANL	1324.5	138.7	89.5	779.2	41.2	264.9	80.0	309.6	76.6	296.2	76.9
CTC	12797.6	249.8	98.0	566.5	95.6	345.9	97.3	471.6	96.3	810.8	81.0
KTH	6498.2	114.2	98.2	154.0	97.6	105.2	98.4	128.6	98.0	127.2	98.3
SCSC	25865.5	353.1	98.6	2125.9	91.8	607.4	97.7	1115.5	95.7	1446.3	89.7
Maximum waiting time (min)											
LANL	7837	3001	61.7	7283	7.0	4583	41.5	4838	38.3	4917	37.3
CTC	31923	5400	83.1	8453	73.5	6077	81.0	6973	78.2	10155	68.2
KTH	16972	4370	74.3	4605	72.9	4984	70.6	5406	68.1	5295	68.8
SCSC	91820	7661	91.7	17909	80.5	9600	89.5	14267	84.5	19274	79.0
Average bounded slowdown											
LANL	1279.7	73.1	94.3	520.2	59.3	100.4	92.2	123.1	90.4	143.5	88.8
CTC	4275.2	52.0	98.8	76.3	98.2	27.2	99.4	39.1	99.1	96.2	97.8
KTH	7518.7	90.8	98.8	73.3	99.0	30.8	99.6	38.3	99.5	76.3	99.0
SCSC	14031.4	92.9	99.3	479.8	96.6	94.1	99.3	192.5	98.6	290.2	97.9
Maximum bounded slowdown											
LANL	46956	10580	77.5	43464	7.4	23579	49.8	25105	46.5	29318	37.6
CTC	174853	9982	94.3	23306	86.7	13272	92.4	13039	92.5	29410	83.2
KTH	101160	14805	85.4	18604	81.6	11521	88.6	13430	86.7	12705	87.4
SCSC	521304	7145	98.6	25127	95.2	11406	97.8	18551	96.4	27575	94.7



(a) Average waiting time (b) Maximum waiting time

(c) Average bounded slowdown (d) Maximum bounded slowdown

Fig. 7. Performance for *FWload* with estimated CPU usage. Here and hereafter, ACFCFS10 denotes the estimation error of CPU usage is less than 10%; ACFCFSUN depicts that the algorithm does not use CPU usage information at all.

total idle CPU cycles ACFCFS can use on the one hand, and benefits the packing of FCFS on the other hand.

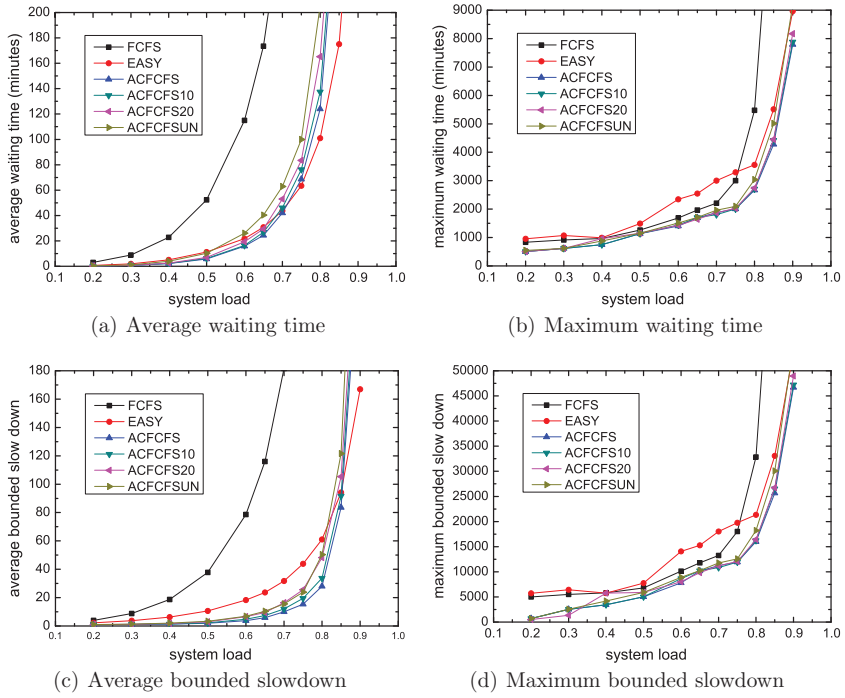
4. The improvement ACFCFS makes on bounded slowdown is more significant than that on the average response time due to the background execution of small jobs. Small jobs are very likely to be short jobs as mentioned in Section 3.2.1, hence earlier execution of short jobs contributes more for the average bounded slowdown.
5. The number of job kills increases as the system load increases in ACFCFS. The number of tentative runs in the foreground tier increases as the system load increases. Moreover, these tentative runs are much easier to be killed because they are very likely to violate the rule of FCFS in case of high system load.
6. The number of job kills is small when the system load is low or high in CCFCFS, and is big in modest system load. When the system load is low, background jobs can be switched to the foreground tier easier; when the system load is high, more (small) jobs can benefit from the background execution.

In the results described above, ACFCFS shows better performance than CCFCFS. We will use ACFCFS for further comparison in the following discussions.

4.3. Discussions

4.3.1. Impact of the accuracy of CPU usage estimation

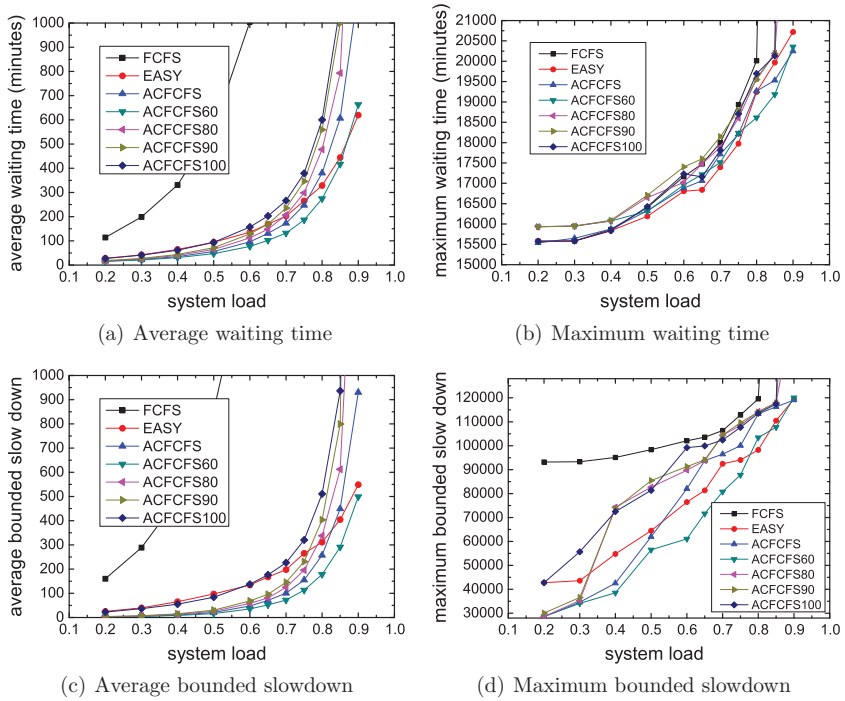
ACFCFS relies on the CPU usage information of parallel processes of jobs to make scheduling decisions. The information can be obtained from profiling a job in test runs or based on user's estimation. Either



(a) Average waiting time (b) Maximum waiting time

(c) Average bounded slowdown (d) Maximum bounded slowdown

Fig. 8. Performance for *JWload* with estimated CPU usage.



(a) Average waiting time (b) Maximum waiting time

(c) Average bounded slowdown (d) Maximum bounded slowdown

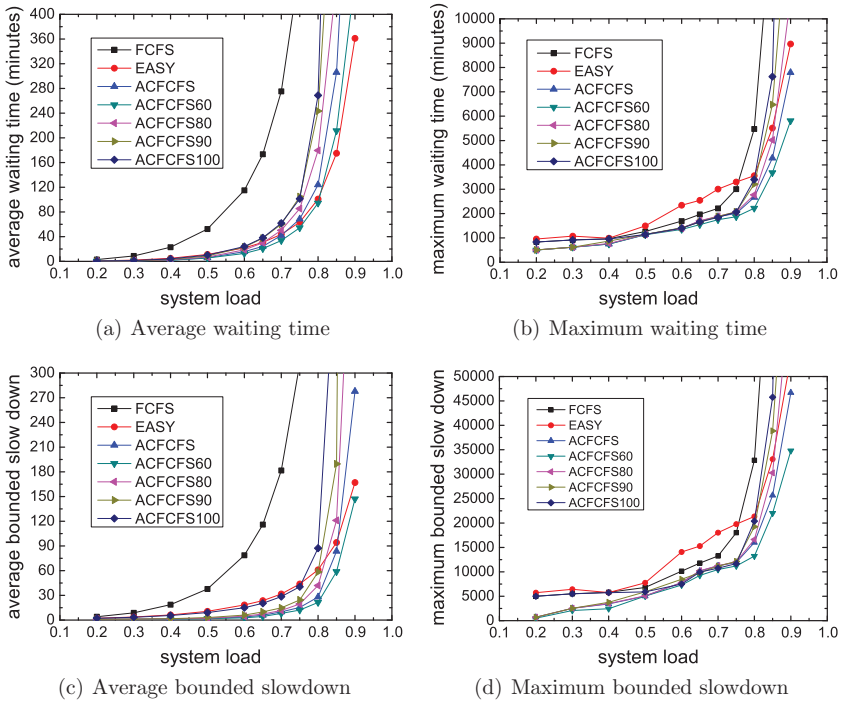
Fig. 9. Performance for *FWload* under varying average CPU usage. Here and hereafter, ACFCFS60 denotes the average CPU usage of processes in multi-processes jobs is 60% (20–100%); ACFCFS100 means that all the processes involved consume the whole CPU cycles of their host processors.

way, incorrect estimation of such information can be a problem. In this section, we investigate the impact of information inaccuracy on the performance of our algorithms by three experiments. We assume that the CPU usage estimation of each process has an error r (a factor of r), i.e., the estimation is within of the accurate value. We set r to 0.1 and 0.2 in two experiments. In the third experiment, we assume no CPU usage information is known and a process has to be randomly placed in the place where CPU usage is needed.

As shown in Figs. 7, 8 and Table 2, ACFCFS outperforms FCFS significantly and produces comparable performance to EASY (especially when the system load is less than 0.8) even without any CPU usage information of parallel processes. However, certain estimation of CPU usage (even with 20% error) can improve the scheduling performance significantly.

4.3.2. *Impact of average CPU usage of parallel processes*

ACFCFS makes use of remaining computing capacity of each processor. In this section, we further investigate the impact of average CPU usage of parallel processes on the performance of ACFCFS. We



(a) Average waiting time (b) Maximum waiting time

(c) Average bounded slowdown (d) Maximum bounded slowdown

Fig. 10. Performance for *JWload* under varying average CPU usage.

change the average CPU usage of multi-processes jobs and examine the performance change of ACFCFS.

Figs. 9, 10 and Table 2 show the results under different average CPU usage. The results demonstrate that the benefit gained from VM collocation is inversely proportional to the average CPU usage of parallel processes. Even if the average CPU usage is 100%, ACFCFS still performs significantly better than FCFS even with context switching overhead. This is due to lots of tentative runs successfully by chance.

5. Conclusions and future work

Running parallel applications in the cloud becomes more and more popular now. It is a challenging for cloud providers to achieve responsiveness of parallel jobs and high processor utilization simultaneously. In this paper, we introduced a prioritized two-tier VMs architecture to organize VMs for running parallel jobs. The foreground tier of VMs has higher CPU priority than that of the background tier of VMs. The performance of jobs running in the foreground VMs is close to that of jobs running in dedicated processors (less than 4% performance loss), meanwhile, the idle CPU cycles can be well used by the jobs running in background VMs. We gave a scheduling algorithm named ACFCFS to exploit the increased computing capacity provided by the two-tier VMs architecture. The proposed ACFCFS algorithm extends the popularly used FCFS algorithm, and it preserves all the advantages of FCFS, such as no starvation, no requirement for job's runtime estimation, easy to implement and no job migration. Our evaluation showed that ACFCFS significantly outperforms FCFS, and achieves comparable performance to the runtime-estimation-based EASY algorithm. ACFCFS is robust in terms that it allows inaccurate CPU usage estimation of parallel processes and low available idle CPU cycles.

In our future work, we will exploit mechanisms that can effectively partition the computing capacity of a processor into *k*-tiers, which may further improve the processor utilization and responsiveness for parallel workload in the cloud. Another issue is that in a large datacenter, processes of a job may need to be allocated to nodes that are close to each other to minimize the communication cost. At last, node with multi-cores is ubiquitous nowadays, parallel job scheduling under this architecture, deserves our further research.

Acknowledgments

This work was supported by NSFC award nos. 61403402, 91024030 and 61074108. A lot of thanks should be given to referees and editors, their valuable comments greatly improved the quality of the manuscript.

References

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A., 2003. Xen and the art of virtualization. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03), October 2003. ACM, New York, NY, USA, pp. 164–177.

Bui, V., Zhu, W., Botta, A., Pescape, A., 2010. A Markovian approach to multipath data transfer in overlay networks. IEEE Trans. Parallel Distrib. Syst. 21 (10), 1398–1411

Etsion, Y., Tsafrir, D., 2005. A Short Survey of Commercial Cluster Batch Schedulers. Technical Report 2005-13, Hebrew University of Jerusalem.

Feitelson, D., 1996. Packing schemes for gang scheduling. In: Feitelson, D., Rudolph, L. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 1162. Springer, Berlin, Heidelberg, pp. 89–110.

Feitelson, D., 2011. Parallel workload models. http://www.cs.huji.ac.il/labs/parallel/workload/index.html.

Feitelson, D., Jettee, M., 1997. Improved utilization and responsiveness with gang scheduling. In: Feitelson, D., Rudolph, L. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 1291. Springer, Berlin Heidelberg, pp. 238–261.

Feitelson, D., Rudolph, L., Schwiegelshohn, U., Sevcik, K., Wong, P., 1997. Theory and practice in parallel job scheduling. In: Feitelson, D., Rudolph, L. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 1291. Springer, Berlin Heidelberg, pp. 1–34.

Jann, J., Pattnaik, P., Franke, H., Wang, F., Skovira, J., Riordan, J., 1997. Modeling of workload in MPPS. In: Feitelson, D., Rudolph, L. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 1291. Springer, Berlin Heidelberg, pp. 95–116.

Lawson, B., Smirni, E., 2002. Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 2537. Springer, Berlin Heidelberg, pp. 72–87.

Lawson, B., Smirni, E., Puiu, D., 2002. Self-adapting backfilling scheduling for parallel systems. In: Proceedings of the 2002 International Conference on Parallel Processing, August 2002. IEEE Computer Society, Vancouver, B.C., Canada, pp. 583–592.

Lifka, D., 1995. The ANL/IBM SP scheduling system. In: Feitelson, D., Rudolph, L. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 949. Springer, Berlin Heidelberg, pp. 295–303.

Lindsay, A.M., Galloway-Carson, M., Johnson, C.R., Bunde, D.P., Leung, V.J., 2013. Backfilling with guarantees made as jobs arrive. Concurr. Comput. Pract. Exp. 25 (4), 513–523.
Liu, X., Wang, C., Zhou, B.B., Chen, J., Yang, T., Zomaya, A., 2013. Priority-based consolidation of parallel workloads in the cloud. IEEE Trans. Parallel Distrib. Syst. 24 (9), 1874–1883.
Moschakis, I., Karatza, H., 2012. Evaluation of gang scheduling performance and cost in a cloud computing system. J. Supercomput. 59 (2), 975–992.
Mu'alem, A., Feitelson, D., 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel Distrib. Syst. 12 (6), 529–543.
Nicod, J., Philippe, L., Rehn-Sonigo, V., Toch, L., 2011. Using virtualization and job folding for batch scheduling. In: Proceedings of the 2011 10th International Symposium on Parallel and Distributed Computing (ISPDC 2011), July 2011. IEEE Computer Society, Cluj Napoca, Cluj, Romania, pp. 33–40.
Perkovic, D., Keleher, P., 2000. Randomization, speculation, and adaptation in batch schedulers. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, November 2000. IEEE Computer Society, Washington, DC, USA, pp. 1–7.
Schwiegelshohn, U., Yahyapour, R., 1998. Analysis of first-come-first-serve parallel job scheduling. In: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98), January 1998. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 629–638.
Snell, Q., Clement, M., Jackson, D., 2002. Preemption based backfill. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 2537. Springer, Berlin, Heidelberg, pp. 24–37.
Sodan, A., 2009. Adaptive scheduling for QOS virtual machines under different resource allocation c performance effects and predictability. In: Frachtenberg, E., Schwiegelshohn, U. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 5798. Springer, Berlin, Heidelberg, pp. 259–279.
Sonmez, O., Grundeken, B., Mohamed, H., Iosup, A., Epema, D., 2009. Scheduling strategies for cycle scavenging in multicluster grid systems. In: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09), May 2009. IEEE Computer Society, Shanghai, China, pp. 12–19.
Speitkamp, B., Bichler, M., 2010. A mathematical programming approach for server consolidation problems in virtualized data centers. IEEE Trans. Serv. Comput. 3 (4), 266–278.
Thebe, O., Bunde, D., Leung, V., 2009. Scheduling restartable jobs with short test runs. In: Frachtenberg, E., Schwiegelshohn, U. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 5798. Springer, Berlin, Heidelberg, pp. 116–137.
Tsafrir, D., 2011. A model/utility to generate user runtime estimates and append them to a standard workload file. http://www.cs.huji.ac.il/labs/parallel/workload/m_tsafrir05.
Tsafrir, D., Etsion, Y., Feitelson, D., 2005. Modeling user runtime estimates. In: Feitelson, D., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (Eds.), Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 3834. Springer, Berlin, Heidelberg, pp. 1–35.
Wiseman, Y., Feitelson, D., 2003. Paired gang scheduling. IEEE Trans. Parallel Distrib. Syst. 14 (6), 581–592.
Zhu, X., Qin, X., Qiu, M., 2011. Qos-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters. IEEE Trans. Comput. 60 (6), 800–812.

Xiaocheng Liu: The authors are all from the college of Information systems and management at National University of Defense Technology (P. R. China), in which Xiaocheng Liu is a lecturer on resource allocation in the cloud, Yabing Zha is a professor on parallel and distributed systems, Quanjun Yin is an associate processor on parallel algorithms, Yong Peng is a lecturer on distributed simulation, Long Qing is a Ph.D. student on agent-based modelling and simulation.