Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2012, Article ID 695757, 18 pages
doi:10.1155/2012/695757

Research Article

Scheduling Parallel Jobs Using Migration and Consolidation in the Cloud

Xiaocheng Liu, Bin Chen, Xiaogang Qiu, Ying Cai, and Kedi Huang

System Simulation Lab, Mechatronics and Atuomation School, National University of Defense Technology, Hunan Province, Changsha, 410073, China

Correspondence should be addressed to Xiaocheng Liu, nudt200203012007xcl@gmail.com

Received 27 February 2012; Revised 26 June 2012; Accepted 5 July 2012

Academic Editor: Rubén Ruiz García

An increasing number of high performance computing parallel applications leverages the power of the cloud for parallel processing. How to schedule the parallel applications to improve the quality of service is the key to the successful host of parallel applications in the cloud. The large scale of the cloud makes the parallel job scheduling more complicated as even simple parallel job scheduling problem is NP-complete. In this paper, we propose a parallel job scheduling algorithm named MEASY. MEASY adopts migration and consolidation to enhance the most popular EASY scheduling algorithm. Our extensive experiments on well-known workloads show that our algorithm takes very good care of the quality of service. For two common parallel job scheduling objectives, our algorithm produces an up to 41.1% and an average of 23.1% improvement on the average response time; an up to 82.9% and an average of 69.3% improvement on the average slowdown. Our algorithm is robust even in terms that it allows inaccurate CPU usage estimation and high migration cost. Our approach involves trivial modification on EASY and requires no additional technique; it is practical and effective in the cloud environment.

1. Introduction

Cloud computing provides an easy-to-use and cost-effective solution for running high performance computing (HPC) parallel applications nowadays. There are efforts [1–4] to run HPC parallel applications in commercial cloud computing platforms. Some public IaaS providers such as Amazon EC2 have launched their HPC cluster instance [5] to cater for HPC parallel applications. Low utilization is a major issue in datacenters mainly due to the fact that datacenter operators prepare computing resources based on the peak load [6]; thus, a lower than 50% utilization is very common in many datacenters [7]. For a datacenter that

hosts HPC parallel applications in addition to normal workload, the utilization can be lower because of the following two factors.

(1) Processor fragmentation: if currently free processors cannot meet the requirement of the candidate job, these free processors may remain idle.

(2) Idle CPU cycles: even there is a process running in a processor, the utilization of the processor is hardly 100% due to the communication and synchronization among the processes.

Low utilization increases the cost of running HPC parallel applications in the cloud and reduces the benefit of the datacenter. How to make use of the underused computing resource to improve the quality of service is the basic question for a cloud provider.

A parallel job schedule S determines the starting time t_i and the processor set M_i for each parallel job. t_i must be bigger than the submission time, and M_i must be disjoint [8]. The common objectives of the scheduling are average response time (response time = completion time − submission time) and average slowdown (slowdown = response time/execution time). Response time measures how long a job takes to finish, and slowdown measures how much slower than a dedicated machine the system appears to the users [9]. Even simple parallel job scheduling problems based on these objectives are NP-complete [10, 11], and many parallel job scheduling methods have been proposed for cluster [12].

Variable partitioning is the most popular parallel job scheduling scheme in practice. FCFS (First-Come-First-Serve) [13] algorithm is the basic variable partitioning scheme. It is simple and straightforward, but this manner suffers from severe processor fragmentation. When the number of free processors is less than the requirement of the next job, these free processors remain idle until additional ones become available [14]. FCFS with backfilling [14, 15], which was developed as the Extensible Argonne Scheduling sYstem (EASY) for IBM SP1, allows short and/or small jobs to use idle processors while the job that arrives earlier than them does not have enough number of processors to run. EASY makes reservation to the head job in the queue, and it is now the most popular scheduling method in the cluster installation [12]. In addition, SJF (Shortest-Job-First) is often employed to assist parallel job scheduling in the literature [16]. Other parallel job scheduling schemes such as dynamic partition [17] and gang scheduling [18] have not been widely accepted because of their limitations in practice [14].

These parallel scheduling schemes try to improve utilization caused by processor fragmentation but none of them takes the idle CPU cycles into account.

In datacenters, parallel application migration based on checkpoint is as always well supported by datacenters [19–22] or parallel applications themselves [23]. Parallel workload consolidation for better utilization is well supported by virtualization technology in the cloud as well [24, 25]. Thus, a natural idea is to employ these approaches to enhance existing parallel job scheduling algorithms. In this paper, we propose a migration and consolidation based parallel job scheduling algorithm named MEASY. Our algorithm uses parallel application migration to alleviate the pain of processor fragmentation on the one hand and introduces workload consolidation to further improve the utilization of the processors on the other hand.

When a parallel job is submitted by a user, the following information should be provided along with it:

(1) the estimated execution time of the job;

(2) the processor requirement of the job;

(3) the estimated CPU usage of the process(es) of the job, which can be obtained by historical data or test runs [26].

Our evaluation results show that our algorithm significantly outperforms the commonly used EASY algorithm on well-known parallel workloads.

The remainder of this paper is organized as follows: Section 2 proposes the parallel workload consolidation method and our migration and consolidation based algorithm. Section 3 describes the evaluation on our algorithm, and Section 4 concludes the paper.

2. Scheduling Algorithms

In this section, we describe our migration and consolidation based parallel job scheduling algorithms. We first introduce the parallel workload consolidation method, then a basic algorithm followed by two refined algorithms that are discussed; the refined ones try to improve the basic algorithm.

2.1. Parallel Workload Consolidation Method (See [27])

The hardware virtualization technology used in cloud computing gives an easy-to-use way to consolidate parallel workload in a datacenter. In order to improve the CPU utilization of the physical processor, we first partition each processor into two-tier virtual machines (VMs) by pinning two virtual CPUs (VCPUs) on the processor and then allocating these two VCPUs to the two VMs. But the execution time of jobs running in the VMs is stretched if there exists no CPU priority control on the VMs. Thus, we secondly assign the VMs on one tier with highest CPU priority and assign the VMs on the other tier with lowest CPU priority. We call the tier with high CPU priority *foreground (fg)* tier and the one with low CPU priority *background (bg)* tier. In this setting, the background VM only uses CPU cycles when its corresponding foreground VM is idle. Under this prioritized two-tier architecture, experiments in our small cluster conclude the following.

(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) Jobs running in the background tier can make good use of the idle CPU cycles left by the foreground tier. For a single-process background job, the utilization of the idle CPU cycles is between 80% and 92%, which roughly obeys uniform distribution; for a multiprocesses background job, the value is between 19.8% and 76.6%, which is likely to obey normal distribution with $\mu = 0.428$ and $\sigma = 0.144$.

(3) 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 the VM does not have much chance to run and the context switch overhead incurred.

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

```
input: J_fg: jobs running in fg VMs;
       J_bg: jobs running in bg VMs;
       J_q: jobs waiting in the queue.
(1) begin
(2)   /*Step 1: Schedule runnable jobs according to FCFS */
(3)   sort J_q ∪ J_bg according to job arrival time;
(4)   for each job j in J_q ∪ J_bg do
(5)     N_j ← process number of j; N_idle ← idle fg VM number;
(6)     if N_j > N_idle then
(7)       break;
(8)     else
(9)       If Deploy (j, 'K') then
(10)          remove j from J_q or J_bg, insert it into J_fg;
(11)  if J_q ∪ J_bg is empty then
(12)    return;
(13)  /*Step 2: Make reservation for the first job in J_q ∪ J_bg, then backfill*/
(14)  let T_S = 0 (shadow time), N_E = 0 (extra fg VM number);
(15)  HdJob ← first job in J_q ∪ J_bg; N_R ← process number of HdJob;
(16)  N_future ← current idle fg VM number;
(17)  Sort J_fg in ascending order of their termination time;
(18)  for each job j in the sorted J_fg do
(19)    N_j ← processe number in j; N_future = N_future + N_j;
(20)    if N_future ≥ N_R then
(21)      T_S = the termination time of j; N_E = N_future − N_R; break;
(22)  /* Backfill runnable jobs*/
(23)  for each job j in J_q ∪ J_bg do
(24)    N_j ← processe number of j; N_idle ← idle fg VM number;
(25)    if N_j > N_idle then
(26)      continue;
(27)    t_r ← the runtime of j; t_c ← current time;
(28)    if (t_r + t_c ≤ T_S) or N_j ≤ N_E then
(29)      if Deploy (j, 'K') then
(30)        remove j from J_q or J_bg, insert it into J_fg;
(31)      if t_r + t_c > T_S then
(32)        N_E = N_E − N_j;
(33)  /*Step 3: Try to deploy jobs to the background tier*/
(34)  sort J_q in ascending order of their runtime;
(35)  for each job j in the sorted J_q do
(36)    N_j ← processe number of j; N^b_idle ← idle bg VM number;
(37)    if N_j ≤ N^b_idle then
(38)      Dispatch('BG',j);
```

Algorithm 1: FCFS with KEASY backfilling and consolidation.

2.2. The Basic Algorithm

2.2.1. Algorithm Description

The basic algorithm was proposed by us in [27], which is named FCFS with KEASY (job Kill based EASY) in this paper, as shown in Algorithm 1. In the KEASY algorithm, shadow time is the latest time that the reservation (of foreground VMs here and hereafter) for a job starts at. The scheduling progress of FCFS with KEASY (shortened as KEASY in the rest of this paper) is as follows.

Step 1. Use FCFS to schedule all possible jobs to foreground VMs.

Step 2. Use EASY to make reservation to the currently head job of the unforeground-running job list (a merged list contains both jobs waiting in the queue and jobs running in the background tier, see $J_q \cup J_{\text{bg}}$ in Algorithm 1), and then backfill all possible jobs to foreground VMs.

Step 3. Use SJF to deploy all possible jobs into background VMs.

The following four points need to be emphasized.

(1) KEASY is called to schedule jobs to run in foreground VMs when job arrives or job departs the foreground tier; SJF is called to schedule jobs to run in background VMs when job departs the background tier.

(2) When selecting jobs to run in foreground VMs, jobs both waiting in the queue and running in the background VMs are candidates.

(3) Only the background VM whose corresponding foreground VM's CPU utilization is less than a threshold (96%) can accommodate process.

(4) Jobs' execution time estimates are used whenever this information is needed for the scheduling decision making.

2.2.2. Examples

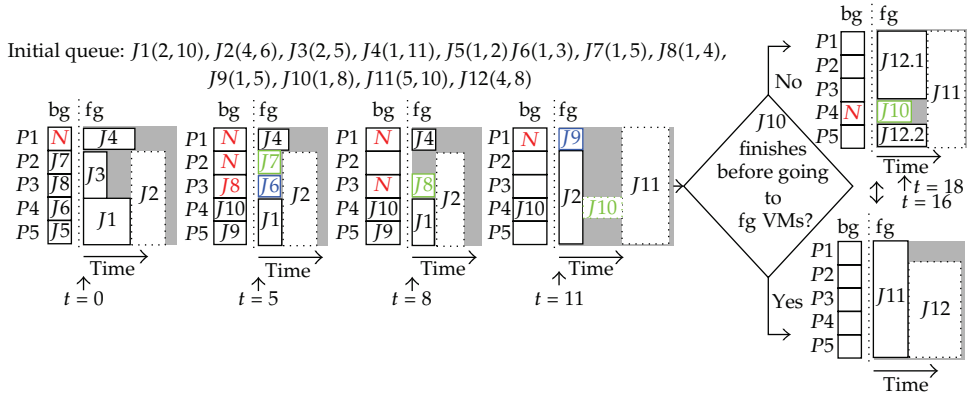
Figure 1(a) gives an example of the KEASY algorithm. Let there be 5 processors (P1–P5) and 12 jobs (J1–J12) at the initial time. Each job is denoted by (n, t), in which n is the processor requirement and t is the execution time. Each processor has two tier VMs denoted as fg and bg in Figure 1. For the convenience of illustration here, we assume that the process in a single-process job incurs a CPU usage of 100% and the processes within a multiprocesses job involve a CPU usage less than the utilization threshold.

At time 0, J1 is placed onto P4-5 according to FCFS; J3 and J4 are deployed onto the foreground VMs of P1–3 according to EASY backfilling; J5, J6, J7, and J8 are scheduled onto the background VMs of P2–5 by SJF. We use a simple process to collocate a background VM with a foreground VM, as shown in the Dispatch function in Algorithm 2. The 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.

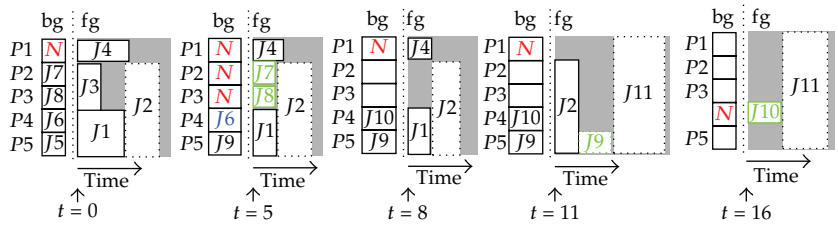
At time 5 (we assume J5–8 all advance 2 time units during time 0–5 here), J3 and J5 depart the system. J6 is backfilled onto the foreground VM of P3 by killing its run in P4 and then restarting its run from the very beginning in the foreground VM of P3; J7 is backfilled onto the foreground VM of its original processor by swapping the CPU priorities of the foreground VM and background VM of P2. Then, J9 and J10 are arranged onto the background VMs of P5 and P4, respectively. Note that although J8 still stays at the background VM of P3, it hardly advances because there is no idle CPU cycle left by J6.

At time 8, J6 and J7 depart the system. Although J8 cannot be backfilled as its original execution time is 4, it still finishes at time 10 by running in the background VM (it is actual the foreground VM because its foreground VM is idle). J1 finishes along with J8 at time 10 as well. At time 11 (we here assume J9 advances 2 time units during time 5–11), J4 finishes. J9 is restarted from the very beginning on the foreground VM of P1 because it is the head job now.

If J10 finishes earlier than time 16, the reservation for J10 should be deleted and the reservation for J11 should be made. In this situation, J11 can run on the foreground VMs

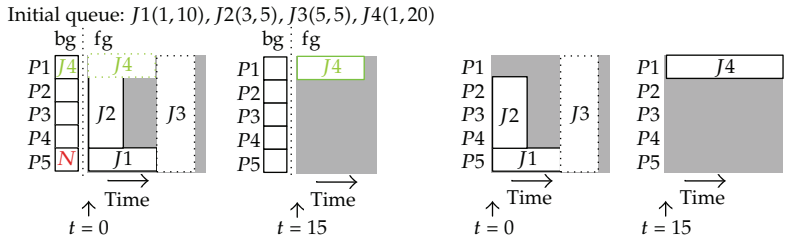


(a) Example of KEASY



(b) Example of REASY

Figure 1: Examples of our algorithms, in which red font means a background VM is not idle; green font indicates a job can run on its original nodes straightforward; in KEASY, blue font tells that a job should be killed first and then restart from the very beginning on other nodes.



(a) KEASY (b) EASY

Figure 2: Benefit for head job.

of P1–5 immediately at time 16. If J10 cannot finish before going to the foreground tier, the reservation for J11 is made at time 16 and, in this case, J12 leapfrogs J11.

In the example described above, the idle CPU cycles unused by the multiprocesses jobs give KEASY the opportunity to improve the scheduling performance. We further observe that even when all processes of all jobs consume the whole computing capacity (100% CPU usage) of the processors, KEASY may still produce better performance than EASY. Figures 2 and 3 show two scenarios that KEASY outperforms EASY if jobs' runtime is accurate. Figure 4 shows a case in which KEASY outperforms EASY due to the overestimation of jobs' runtime. From the three examples, we can see that KEASY is capable of dispatching a job to run in background VMs while it is not qualified for backfilling according to EASY. There is a chance

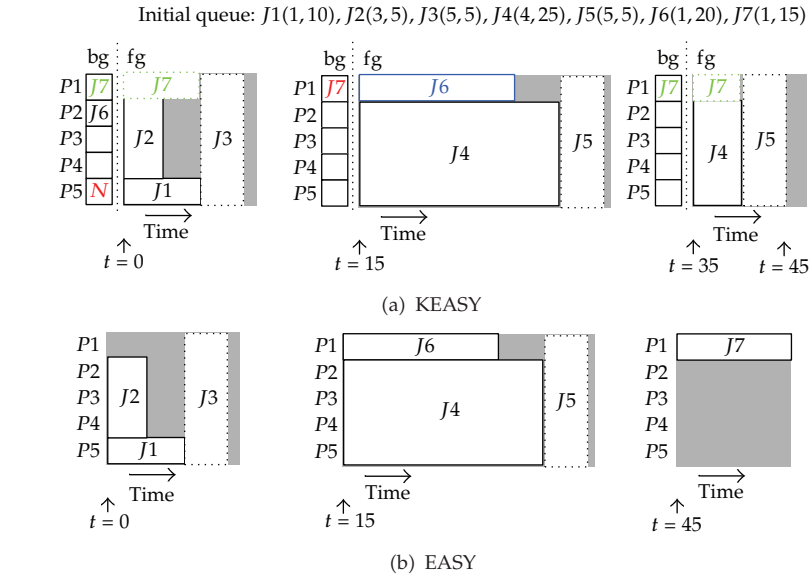


Figure 3: Benefit for backfilled job.

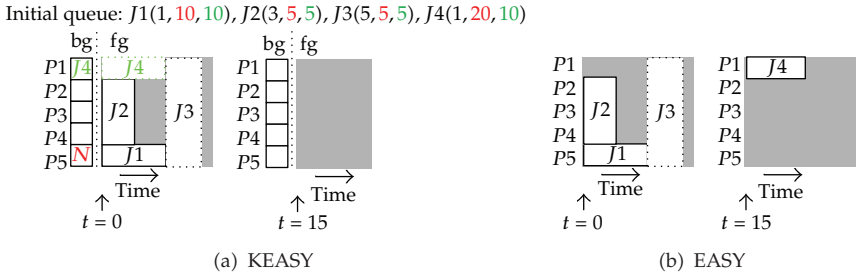


Figure 4: Benefit for overestimated execution time: a job is described by tuple (n, t_e, t_a), in which n is the number of processes, t_e is the estimated execution time, and t_a is the actual execution time.

that the corresponding foreground VMs of these background VMs are idle during the job's lifetime, which leads to performance improvement.

2.3. *Two Improved Algorithms*

The basic algorithm described above adopts job kill during the scheduling; this somewhat leads to a waste of computing resource. In this section, we try to remove job kill and present two refined algorithms.

2.3.1. REASY—Using Reservation instead of Job Kill

The REASY algorithm is shown in Algorithm 3. In REASY, job kill is not allowed in the scheduling; once a job is deployed onto background VMs of a set of processors, its run

```
input: j:  the job to be deployed;
           BackfillType: the algorithm who calls this function.
output:    IsSucc: indicate whether j be deployed successfully or not;
(1) begin
(2)   IsSucc = false;
(3)   if j is from the background tier then
(4)     if j can still run in its original processors then
(5)       swap the CPU priorities of its bg VMs and fg VMs;
(6)       IsSucc = true;
(7)     else
(8)       /* Job kill is not allowed in REASY but allowed by KEASY */
(9)       IsSucc = false;
(10)      if BackfillType =='K' then
(11)        kill j and Dispatch(FG, j).
(12)        IsSucc = true;
(13)    else
(14)      Dispatch(FG, j).
(15)      IsSucc = true;
(16)    return IsSucc;
(17) function Dispatch (flag, j)
(18) sort processes of j in descending order of their processor utilization;
(19) if flag == FG then
(20)    p ← sorted idle processors in ascending order of their utilization;
(21)    for each process j_i in the sorted list do
(22)      place j_i to p_i and run j_i in the foreground VM;
(23) else
(24)    p ← sorted idle processors in ascending order of their utilization;
(25)    for each process j_i do
(26)      place j_i to p_i and run j_i in the background VM;
```

Algorithm 2: Deploy(j, BackfillType)—job deploy function.

is pinned onto this set of processors. Only all the foreground VMs of this set of processors are available can this job run in the foreground tier. Thus, REASY differs than KEASY in making reservation for the head job and invoking *Deploy* function. For the reservation making, if a reservation is being made for a job running in the background tier, the shadow time is the last termination time of the jobs running in its foreground VMs; the extra foreground VMs are the ones now idle and no process of the job is running in their background VMs. Figure 5 illustrates an instance of this situation; when making reservation for J3, the shadow time is 10 as J1 finishes at time 10 and the extra foreground VM is the foreground VM on P1. For the *Deploy* function invoking, REASY passes "R" rather than "K" to the *Deploy* function (Algorithm 2) so that it only schedules a job now running background tier to foreground tier when all the foreground VMs of its host processors are idle.

An example of REASY is given in Figure 1(b). In the example, we assume that J6 advances 3 time units during time 0–8 and J9 advances 5 time units during time 5–16. One can find that, in time 5, J6 still stays at its original background VM other than restarts on the foreground VM of P1. At time 11, J9 also still runs on the background VM of P5 but a reservation for J9 is made because it is now the head job.

Input: J_{fg}: jobs running in fg VMs;
J_{bg}: jobs running in bg VMs;
J_q: jobs waiting in the queue.

(1) **begin**
(2) /* Make reservation for the first job in $J_q \cup J_{\text{bg}}$, then backfill */
(3) let $T_S = 0$ (shadow time), $N_E = 0$ (extra **fg VM** number);
(4) $HdJob \leftarrow$ first job in $J_q \cup J_{\text{bg}}$; $N_R \leftarrow$ process number of $HdJob$;
(5) $N_{\text{future}} \leftarrow$ current idle **fg VM** number;
(6) **if** $HdJob \in J_q$ **then**
(7) Sort J_{fg} in ascending order of their termination time;
(8) **for** *each job j in the sorted* J_{fg} **do**
(9) $N_j \leftarrow$ processe number in j; $N_{\text{future}} = N_{\text{future}} + N_j$;
(10) **if** $N_{\text{future}} \geq N_R$ **then**
(11) T_S = the termination time of j; $N_E = N_{\text{future}} - N_R$;
(12) **break**;
(13) **else**
(14) $J_h^{fg} \leftarrow$ the jobs running in the foreground VMs of $HdJob$;
(15) $SumSize \leftarrow$ the size sum of jobs in J_h^{fg};
(16) T_S = last termination time of jobs in J_h^{fg};
(17) $N_E = N_{\text{future}} + SumSize - N_R$;
(18) /* Backfill runnable jobs */
(19) **for** *each job j in* $J_q \cup J_{\text{bg}}$ **do**
(20) $N_j \leftarrow$ processe number j; $N_{\text{idle}} \leftarrow$ idle **fg VM** number;
(21) **if** $N_j > N_{\text{idle}}$ **then**
(22) **continue**;
(23) $t_r \leftarrow$ the runtime of j; $t_c \leftarrow$ current time;
(24) **if** $(t_r + t_c \leq T_S)$ or $N_j \leq N_E$ **then**
(25) **if** ***Deploy*** $(j, 'R')$ **then**
(26) remove j from J_q or J_{bg}, insert it into J_{fg};
(27) **if** $t_r + t_c > T_S$ **then**
(28) $N_E = N_E - N_j$;

Algorithm 3: REASY backfilling.

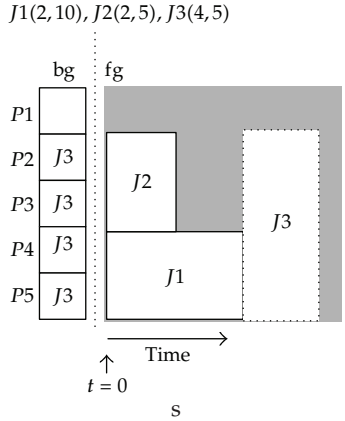


Figure 5: Example of shadow time and extra foreground VMs.

Table 1: Size distribution of workload models used in the experiment.

Model	=1	=2	(2, 4]	(4, 8]	(8, 16]	(16, 32]	(32, 64]	(64, 128]	>128
FWload	17.8%	18.1%	14.9%	19.0%	10.6%	6.2%	5.2%	4.2%	4.0%
JWload	40.8%	7.4%	12.2%	10.9%	13.6%	7.9%	5.2%	1.4%	0.7%

2.3.2. MEASY—Using Migration instead of Job Kill

The only difference between KEASY and MEASY exists in the *Deploy* (Algorithm 2). MEASY does not kill a selected job but uses migration instead.

3. Evaluation (See [27])

The evaluation is performed by trace-driven simulation. During the simulation, once a job arrives, the simulator is informed of the processor need, the execution time estimate, and the execution time (but the execution time is not the input of the scheduling algorithms). Upon jobs departure, the simulator is also notified. The number of processors is 320 in the simulated system; a default cost of 20 seconds is configured in MEASY.

3.1. Workloads

We use the following commonly used workload models in our simulation.

(1) Feitelson workload, denoted by *FWload*: a general model based on data from six different traces [14, 28]. It contains 200,000 jobs in our simulation, the average parallelism of it is 23.4, and the average runtime is 2606.6 seconds.

(2) Jann workload, denoted by *JWload*: a workload model for MPP and it fits the actual workload of Cornell Theory Center Supercomputer [29]. It contains 100,000 jobs in our simulation, the average parallelism of it is 10.5, and the average runtime is 11267.0 seconds.

The job size distributions of the two workloads are shown in Table 1. The execution time estimates of jobs are generated by the model proposed by Tsafrir et al. in [30, 31], denoted by *TModel*.

As the generated workload does not contain CPU usage information for each process, we assign the CPU usage to a process according to the following rules.

(1) If a job has only one process, the process is assigned a CPU usage of 100%.

(2) If a job has more than one processes, the average CPU usage of each process is a random number between 40% to 100%.

All the models mentioned in this section are available in [32].

3.2. The Progress of a Process in the Two-Tier Architecture

Running a foreground VM and a background VM simultaneously on a physical processor incurs overhead due to context switch. We model the progress of a process running in the foreground VM as follows:

$$t = \begin{cases} T & \text{if its bg VMs are all idle} \\ T \cdot \text{loss} & \text{otherwise,} \end{cases} \tag{3.1}$$

in which 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. According to our experimental results described in Section 2.1, loss is randomly generated by a uniform distribution between 0.5% and 4%.

In a time slice, the progress of a background process is calculated as follows:

$$t = \begin{cases} T & \text{if its fg VMs are all idle} \\ T \cdot \text{eff} & \text{else if } \text{CPU}_{\text{idle}} \geq \text{CPU}_{\text{req}} \\ T \cdot \text{eff} \cdot \dfrac{\text{CPU}_{\text{idle}}}{\text{CPU}_{\text{req}}} & \text{otherwise,} \end{cases} \tag{3.2}$$

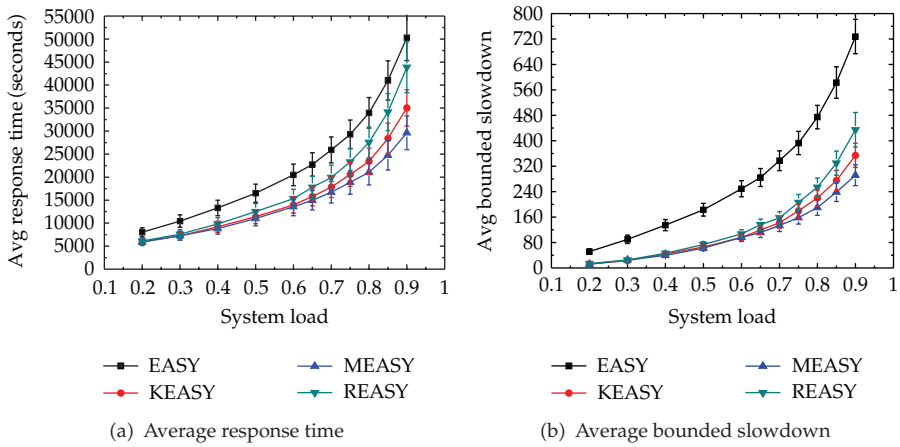
in which eff is a variable between 0 and 1. It represents how much time in a time slice effectively contributes to the progress of the process. A background process is frequently preempted, and eff characterizes the overhead associated. According to Section 2.1, for a single-process job, eff is a random number between 0.80 and 0.92, which is also modeled by a uniform distribution; for a multiprocesses job, eff is randomly generated by a normal distribution with $\mu = 0.428$ and $\sigma = 0.144$. CPU_{req} is the CPU utilization of the background process on a dedicated processor. CPU_{idle} is the portion of unused CPU cycles in the processor, that is, the portion that is not fully utilized by the foreground VM.

Furthermore, the progress of a job depends on the progress of the slowest process in the job.

3.3. Performance Metrics

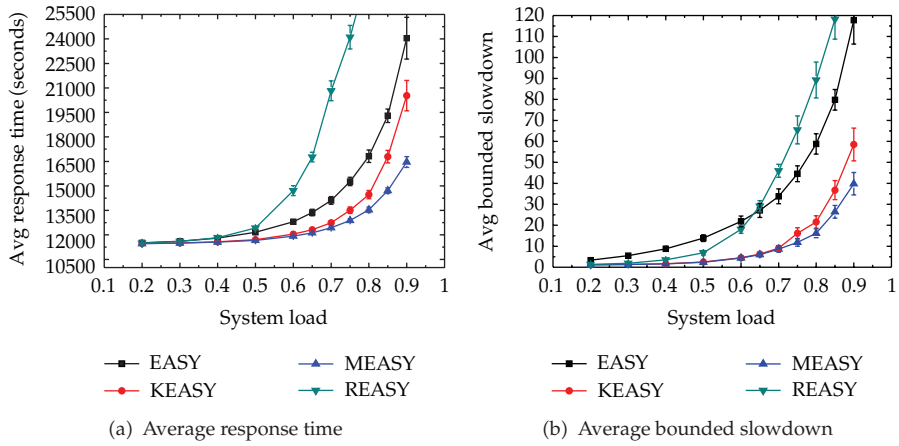
The performance metrics used are the average response time and the average bounded slowdown. The response time of a job is defined as $\text{rt} = t_f - t_a$, and the bounded slowdown of a job is defined as $b\_\text{sld} = (t_f - t_a)/\max(\Gamma, t_e)$, where t_f is the finish time of the job; t_a 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 [14, 33]. According to [14], we set Γ to 10 seconds. The batch means method [34] is used in our simulation analysis.

The system load of the simulated system is modified by multiplying the interarrival times by a factor. For instance, if the model produces a default load of 0.45, a higher load of 0.9 can be created by multiplying all interarrival times by a factor of 0.5 (= 0.45/0.9) [14].



(a) Average response time (b) Average bounded slowdown

Figure 6: Performance for *FWload*. The bars represent the standard deviation.



(a) Average response time (b) Average bounded slowdown

Figure 7: Performance for *JWload*. The bars represent the standard deviation.

3.4. Results

Figures 6 and 7 show the scheduling results of our algorithms for *FWload* and *JWload*; the average number of migration is shown in Figure 8. We have the following observations from the figures.

(1) REASY produces the worst performance among our algorithms. For *JWload*, it is even worse than EASY. This is due to that *JWload* contains about 40.8% of single-process jobs and the percentage in *FWload* is less than 18.0% as shown in Table 1, so REASY delays more jobs which can run earlier by EASY backfilling in *JWload* than in *FWload*.

(2) The best performance is achieved by MEASY. More specifically, it brings an up to 41.1% and an average of 23.1% improvement on the average response time and an up to 82.9% and an average of 69.3% improvement on the average slowdown in contrast to EASY.

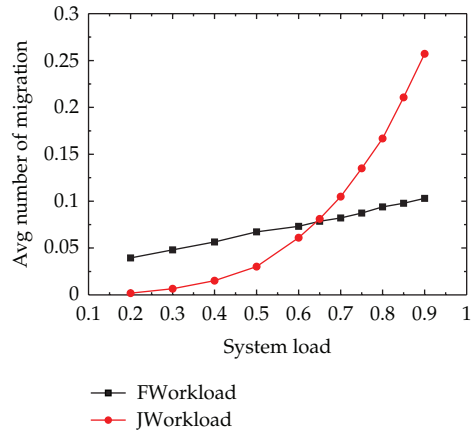


Figure 8: Average number of migration in *FWload* and *JWload*.

(3) The number of job migration in *JWload* is greater than in *FWload* when the system load is high, but *JWload* needs less migration than *FWload* when the system load is low. This is also due to the characteristic of *JWload*. In *JWload*, more jobs can be deployed onto foreground tier straightforward when the system load is low; in the case of high system load, jobs running in background tier are easy to meet the backfilling criteria thus improve the number of migration.

In the results described above, MEASY shows better performance than other algorithms in all aspects. We will use MEASY for comparison in the following discussions.

3.5. Discussions

3.5.1. CPU Usage Estimation Error Tolerance in MEASY

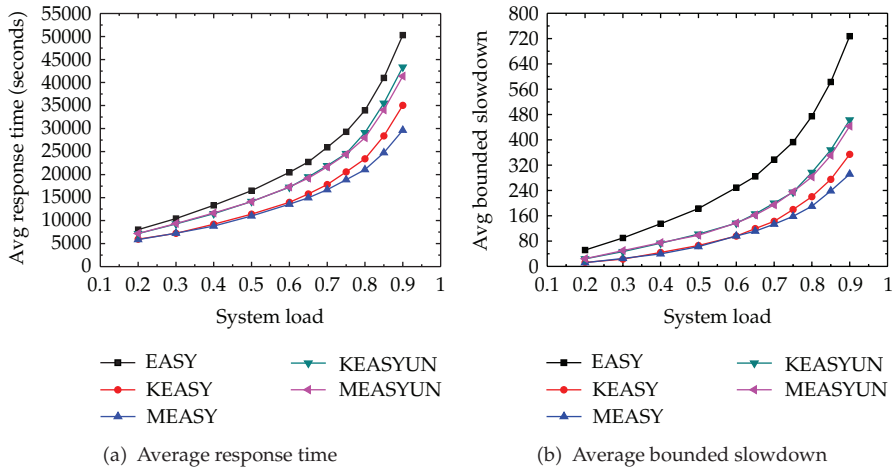
Our scheduling algorithms rely on the CPU usage information of parallel processes of jobs to make scheduling decisions. The information can be obtained from profiling a job in test runs or based on users' estimation. Either way, information inaccuracy can be a problem. In this section, we assume that the CPU usage estimation is not available at all.

As shown in Figures 9 and 10, both MEASY and KEASY outperform EASY even without any CPU usage information of parallel processes, and MEASY is always better than KEASY.

3.5.2. Average CPU Usage of Processes Tolerance in MEASY

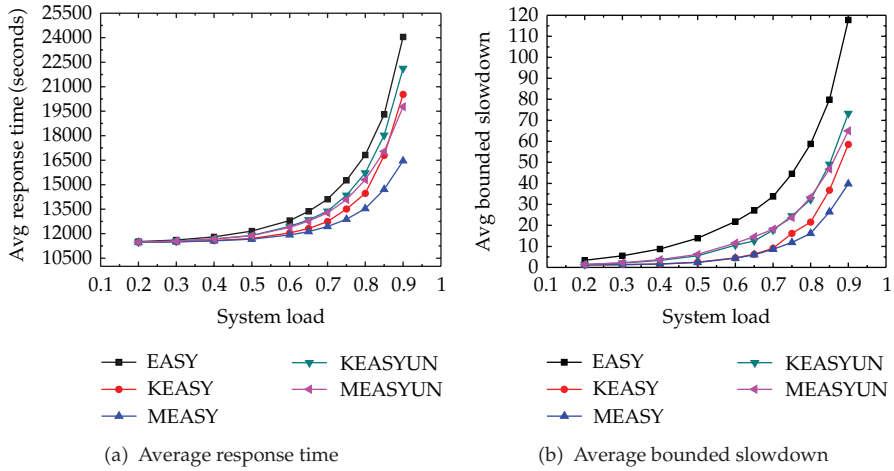
Our algorithms make use of remaining computing capacity of each processor. In this section, we further investigate the impact of average CPU usage of parallel processes on the performance of our algorithms. We change the average CPU usage of multiprocesses jobs and examine the performance change of our algorithms.

Figures 11 and 12 show the results for *FWload* and *JWload* under the situation of 100% CPU usage. The results reflect that even the average CPU usage is 100%; our algorithms



(a) Average response time (b) Average bounded slowdown

Figure 9: Performance for *FWload* without the CPU usage information.



(a) Average response time (b) Average bounded slowdown

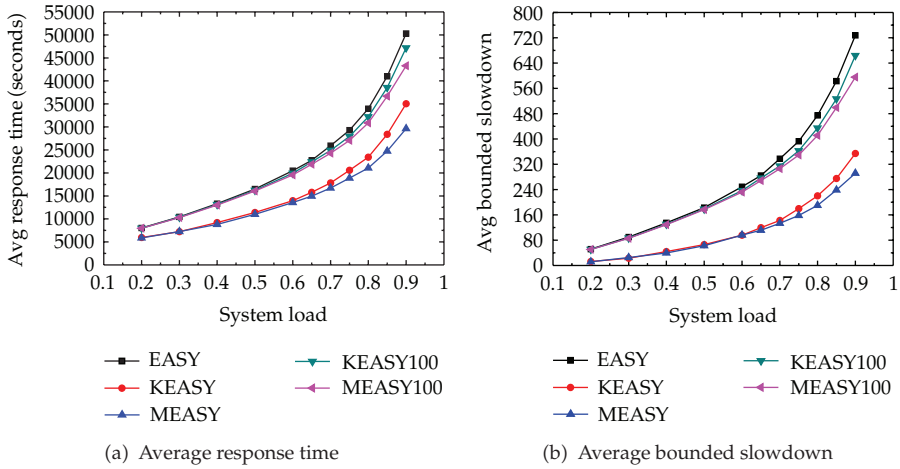
Figure 10: Performance for *JWload* without the CPU usage information.

perform better than EASY even with context switching overhead. MEASY is always better than KEASY as well. This is due to the situations shown in Figures 2, 3, and 4.

3.5.3. Migration Cost Tolerance in MEASY

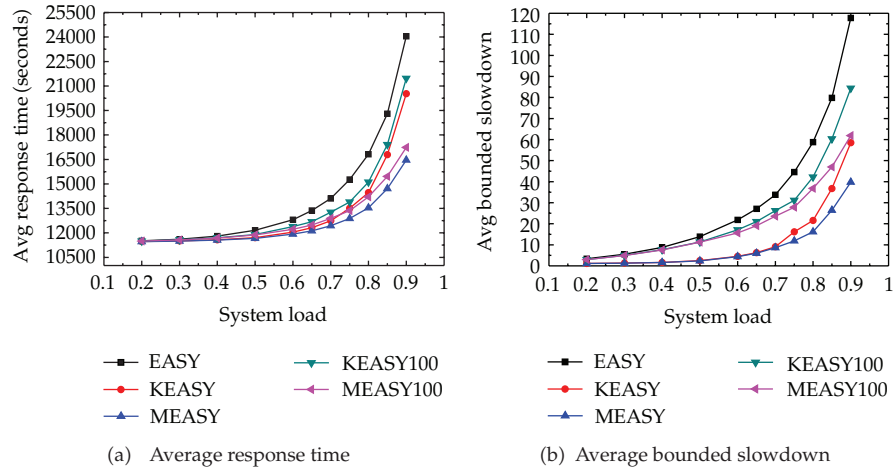
In this section, we examine the impact of migration cost on the performance of MEASY.

Figures 13 and 14 show the results for *FWload* and *JWload* under different job migration costs. From the figures, better performance than KEASY can be obtained by MEASY in the two workloads even with high job migration cost. For *FWload*, if the migration cost is less than 900 seconds, better performance than KEASY can always be obtained by MEASY; if the migration cost is bigger than 1500 seconds, MEASY can only produce comparable (a little worse than KEASY) performance to KEASY. For *JWload*, these values are 480 and



(a) Average response time (b) Average bounded slowdown

Figure 11: Performance for *FWload* with the average CPU usage of 100%.



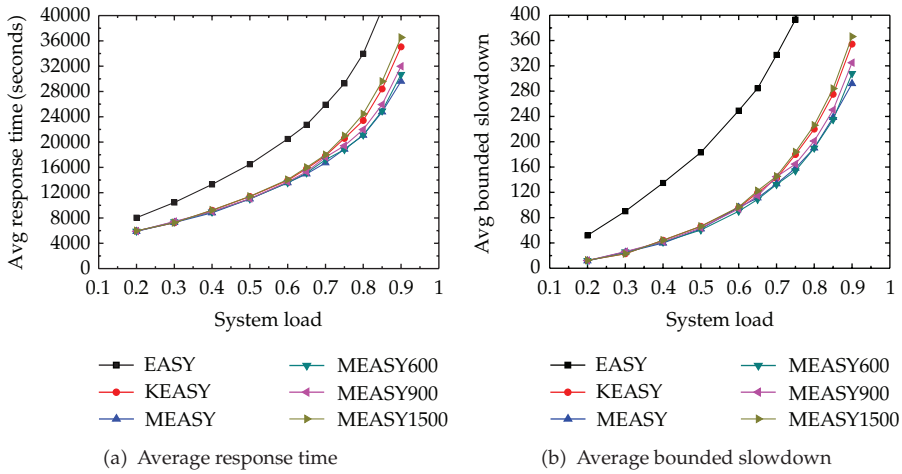
(a) Average response time (b) Average bounded slowdown

Figure 12: Performance for *JWload* with the average CPU usage of 100%.

900, respectively. This is due to the number of migration in *JWload* is greater than in *FWload* as mentioned above. Note that if the migration cost is greater than an upper bound, KEASY which needs no migration may be the best choice in practice.

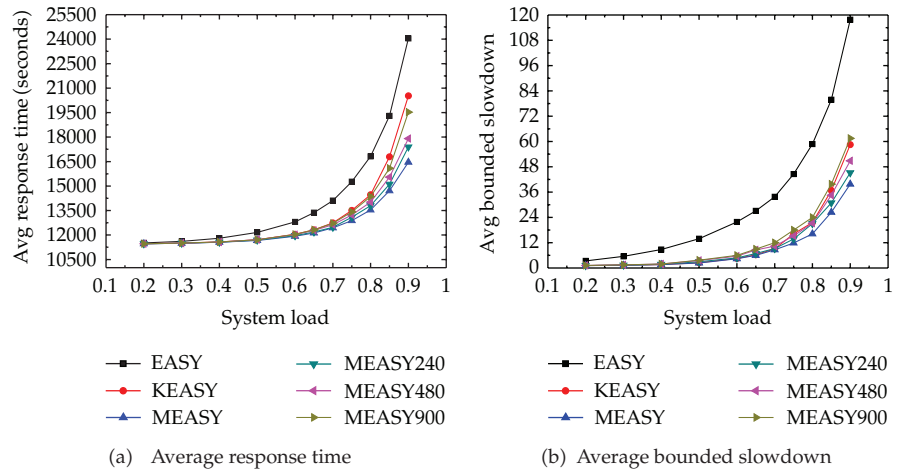
4. Conclusions

Parallel job scheduling is increasingly important for a datacenter in the cloud computing era. It is in charge of the quality of service of the datacenter. In this paper, we employed a two-tier processor partition architecture and put forward three parallel job scheduling algorithms (KEASY, REASY, and MEASY), culminating with MEASY. KEASY is the basic algorithm, and REASY and MEASY extend KEASY. Both KEASY and MEASY produce significant better performance than EASY, but REASY fails in some cases. Moreover, MEASY is always better



(a) Average response time (b) Average bounded slowdown

Figure 13: Performance for *FWload* with varying migration costs.



(a) Average response time (b) Average bounded slowdown

Figure 14: Performance for *JWload* with varying migration costs.

than KEASY and it is robust in terms of allowing inaccurate CPU usage estimation of parallel processes and high job migration cost.

For the future work, we will further study REASY and try to improve its performance, such as only allowing jobs with some constraints to run in background VMs; we will evaluate the performance using other policies other than SJF when scheduling jobs to background VMs as well; we will also exploit mechanisms that can effectively partition the computing capacity of a processor into *k-tiers*, which may further improve the processor utilization and job responsiveness for parallel workload in the cloud, particularly for CPU with multicores.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (no. 91024030 and no. 61074108). A lot of thanks should be given to referees and editors; their valuable comments greatly improved the quality of the paper.

References

[1] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, 2011.

[2] A. W. Malik, A. Park, and R. M. Fujimoto, "Optimistic synchronization of parallel simulations in cloud computing environments," in *Proceedings of IEEE International Conference on Cloud Computing (CLOUD '09)*, pp. 49–56, September 2009.

[3] R. Fujimoto, A. Malik, and A. Park, "Parallel and distributed simulation in the cloud," *Simulation Magazine, Society for Modeling and Simulation*, no. 3, 2010.

[4] G. D'Angelo, "Parallel and distributed simulation from many cores to the public cloud," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS '11)*, pp. 14–23, IEEE, Istanbul, Turkey, 2011.

[5] Amazon, "High performance computing (HPC) on AWS," 2011, http://aws.amazon.com/hpc-applications/.

[6] A. Do, J. Chen, C. Wang, Y. Lee, A. Zomaya, and B. Zhou, "Profiling applications for virtual machine placement in clouds," in *Proceedings of IEEE International Conference on Cloud Computing (CLOUD '11)*, pp. 660–667, Washington, DC, USA, July 2011.

[7] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[8] U. Schwiegelshohn and R. Yahyapour, "Fairness in parallel job scheduling," *Journal of Scheduling*, vol. 3, no. 5, pp. 297–320, 2000.

[9] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 236–247, 2003.

[10] J. Bruno, E. G. Coffman, Jr., and R. Sethi, "Scheduling independent tasks to reduce mean finishing time," *Communications of the Association for Computing Machinery*, vol. 17, pp. 382–387, 1974.

[11] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 4, pp. 473–487, 1989.

[12] Y. Etsion and D. Tsafrir, "A short survey of commercial cluster batch sched-ulers," Tech. Rep. 2005-13, The Hebrew University of Jerusalem, 2005.

[13] U. Schwiegelshohn and R. Yahyapour, "Analysis of first-come-first-serve parallel job scheduling," in *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 629–638, Society for Industrial and Applied Mathematics, New York, NY, USA, 1998.

[14] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[15] D. Lifka, "The anl/ibm sp scheduling system," in *Job Scheduling Strategies for Parallel Processing*, pp. 295–303, Springer, 1995.

[16] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007.

[17] C. McCann, R. Vaswani, and J. Zahorjan, "Dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, no. 2, pp. 146–178, 1993.

[18] D. Feitelson and M. Jettee, "Improved utilization and responsiveness with gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, pp. 238–261, Springer, 1997.

[19] N. Stone, J. Kochmar, R. Reddy, J. Scott, J. Sommerfield, and C. Vizino, "A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system," Tech. Rep. CMU-PSC-TR-2001-0002, Pittsburgh Supercomputer Center, 2001.

[20] Platform Computing, "Platform lsf," 2011, http://www.platform.com/products/LSFfamily/.
[21] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira, *Workload Management with Loadleveler*, IBM, 1st edition, 2001.
[22] V. Systems, *Portable Batch System, Administrator Guide*, OpenPBS Release 2.3, 2000.
[23] E. Mascarenhas, F. Knop, R. Pasquini, and V. Rego, "Checkpoint and recovery methods in the PARASOL simulation system," in *Proceedings of the 29th Winter Simulation Conference*, pp. 452–459, IEEE Computer Society, December 1997.
[24] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing," in *Proceedings of the Conference on Power Aware Computing and Systems*, p. 10, USENIX Association, 2008.
[25] Y. C. Lee and A. Y. Zomaya, "Energy efficient utilization of resources in cloud computing systems," *Journal of Supercomputing*, pp. 1–13, 2010.
[26] Y. Lin, "Parallelism analyzers for parallel discrete event simulation," *ACM Transactions on Modeling and Computer Simulation* , vol. 2, pp. 239–264, 1992.
[27] X. C. Liu, C. Wang, X. G. Qiu, B. B. Zhou, B. Chen, and A. Y. Zomaya, "Backfilling under two-tier virtual machines," in *Proceedings of the International Conference on Cluster Computing (Cluster '12)*, IEEE, 2012.
[28] D. Feitelson, "Packing schemes for gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, pp. 89–110, Springer, 1996.
[29] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan, "Modeling of workload in mpps," in *Job Scheduling Strategies for Parallel Processing*, pp. 95–116, Springer, 1997.
[30] D. Tsafrir, Y. Etsion, and D. Feitelson, "Modeling user runtime estimates," in *Job Scheduling Strategies for Parallel Processing*, pp. 1–35, Springer, 2005.
[31] D. Tsafrir, "A model/utility to generate user runtime estimates and append them to a standard workload file," 2006, http://www.cs.huji.ac.il/labs/parallel/workload/m_tsafrir05/.
[32] "Parallel workload models," 2005, http://www.cs.huji.ac.il/labs/parallel/workload/models.html.
[33] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing*, pp. 1–34, Springer, 1994.
[34] R. Jain, *The Art of Computer Systems Performance Analysis*, Wiley & Sons, 1991.