Priority-Based Consolidation of Parallel Workloads in the Cloud

Xiaocheng Liu, Chen Wang, Bing Bing Zhou, Junliang Chen, Ting Yang, and Albert Y. Zomaya, *Fellow*, *IEEE*

Abstract—The cloud computing paradigm is attracting an increased number of complex applications to run in remote data centers. Many complex applications require parallel processing capabilities. Parallel applications of certain nature often show a decreasing utilization of CPU resources as parallelism grows, mainly because of the communication and synchronization among parallel processes. It is challenging but important for a data center to achieve a certain level of utilization of its nodes while maintaining the level of responsiveness of parallel jobs. Existing parallel scheduling mechanisms normally take responsiveness as the top priority and need nontrivial effort to make them work for data centers in the cloud era. In this paper, we propose a priority-based method to consolidate parallel workloads in the cloud. We leverage virtualization technologies to partition the computing capacity of each node into two tiers, the foreground virtual machine (VM) tier (with high CPU priority) and the background VM tier (with low CPU priority). We provide scheduling algorithms for parallel jobs to make efficient use of the two tier VMs to improve the responsiveness of these jobs. Our extensive experiments show that our parallel scheduling algorithm significantly outperforms commonly used algorithms such as extensible argonne scheduling system in a data center setting. The method is practical and effective for consolidating parallel workload in data centers.

Index Terms—Cloud computing, parallel computing, parallel job scheduling, resource consolidation, parallel discrete event simulation

1 Introduction

The cloud computing paradigm promises a cost-effective solution for running business applications through the use of virtualization technologies, highly scalable distributed computing, and data management techniques as well as a pay-as-you-go pricing model. In recent years, it also offers high-performance computing capacity for applications to solve complex problems [1]. Improving resource utilization is essential for achieving cost effectiveness. Low utilization has long been an issue in data centers. Servers in a typical data center are operated at 10 to 50 percent of their maximum utilization level [2]. 10 to 20 percent utilization is common in data centers [3]. For a data center, or a subset of servers in a data center that mainly handles applications with high-performance computing needs and runs parallel jobs most of the time, the problem can be significant.

There are two factors that may reduce the utilization of nodes that run parallel jobs:

1. A parallel job often requires a certain number of nodes to run. A set of nodes is likely to be fragmented by parallel jobs with different node number requirement. If the number of available nodes cannot satisfy the requirement of an incoming job, these nodes may remain idle [4], [5], [6].
2. Typical parallel programming models, such as BSP [7] often involve computing, communication, and synchronization phase. A process in a parallel job may frequently wait for the data from other processes. During waiting, the utilization of the node is low.

The most basic but popular batch scheduling algorithm for parallel jobs is first come first serve (FCFS) [8]. Each job specifies the number of nodes required and the scheduler processes jobs according to the order of their arrival. When there is a sufficient number of nodes to process the job at the head of the queue, the scheduler dispatches the job to run on these nodes; otherwise, it waits till jobs currently running finish and release enough nodes for the job. FCFS may cause node fragmentation and methods such as backfilling [9] and Gang scheduling [10] were proposed to improve it. However, they do not target on the utilization degradation caused by parallelization itself.

In this paper, we focus on improving resource utilization for data centers that run parallel jobs, particularly we intend to make use of the remaining computing capacity of data center nodes that run parallel processes with low resource utilization to improve the performance of parallel job scheduling. The parallel jobs we deal with have the following characteristics:

1. The job execution time is unknown.
2. Saving and restoring the state of a job is relatively cheap with checkpoint support.

- X. Liu is with the Information and Management School, National University of Defense Technology, Changsha 410073, China. E-mail: nudt200203012007xcl@gmail.com.
- C. Wang is with the CSIRO ICT Centre, PO Box 76, Epping, NSW 1710, Australia. E-mail: chen.wang@csiro.au.
- B.B. Zhou, J. Chen, and A.Y. Zomaya are with the Centre for Distributed and High Performance Computing, School of Information Technologies, University of Sydney, NSW 2006, Australia. E-mail: {bing.zhou, jche7466, albert.zomaya}@sydney.edu.au.
- T. Yang is with the School of Electrical Engineering and Automation, Tianjin University, Tianjin 300072, China. E-mail: yangting@tju.edu.cn.

Manuscript received 25 Apr. 2012; revised 8 Aug. 2012; accepted 20 Aug. 2012; published online 6 Sept. 2012.
Recommended for acceptance by M. Guo.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-04-0415.
Digital Object Identifier no. 10.1109/TPDS.2012.262.

3. The CPU usage of the processes of a job can be estimated either during design phase or through the historical data [11], [12].

Parallel discrete event simulation [13] belongs to this category of jobs, and there are efforts [14], [15], [16] to run this type of jobs in the cloud. In this paper, we propose a priority-based consolidation method for scheduling this type of parallel jobs with the following goals: 1) improve the utilization of servers allocated to these jobs; 2) preserve the FCFS order of jobs when available resources satisfy the needs of these jobs. Our method gives a systematic way to consolidate parallel workload. The basic idea is to put a background virtual machine (VM) in each node so that the background VM can use computing resources when the foreground VM cannot fully utilize them. We make the following contributions:

1. We conduct extensive experiments for workload consolidation. We found that using virtualization technologies with appropriate assignment of priorities to VMs, we can effectively allow jobs collocated in a physical node to efficiently use the computing capacity without significant impact to the performance of the high-priority job.
2. Built on the above observation, we give a priority-based workload consolidation method with the support of underlying VM collocation mechanism. We partition the computing capacity of each physical node into two tiers, namely foreground VM (with high CPU priority) and background VM (with low CPU priority) by pinning two VMs to the node. They can simultaneously process different jobs. The background job can therefore use the underutilized computing capacity whenever the foreground job cannot fully use it. Our method supports backfilling in such a two-tier setting.

Our evaluation results show that our consolidation-based algorithm (Aggressive Migration and Consolidation supported BackFilling (AMCBF)) significantly outperforms FCFS and Extensible Argonne Scheduling sYstem (EASY) (accurate job execution time is available for EASY in our experiment) on well-known traces. In addition, our method outperforms EASY even when it only knows the information of the jobs' node number requirement. Finally, our algorithm can achieve two commonly conflicting goals in parallel job scheduling: improving the system utilization and the job responsiveness.

The remainder of this paper is organized as follows: Section 2 discusses some related work. Section 3 presents our priority-based workload consolidation method. Detailed descriptions of our job scheduling algorithms are given in Section 4. Section 5 evaluates the performance of our algorithms. Section 6 concludes the paper and discusses future work.

2 Related Work

There have been many efforts on scheduling mechanisms for parallel jobs in clusters [17]. FCFS is the basic but popularly used batch scheduling algorithm. Backfilling [9], which was developed as the EASY for IBM SP1, is a technique that allows short/small jobs to use idle nodes while the job at the head of the queue does not have enough number of nodes to run. Backfilling can improve node utilization, but it requires each job to specify its maximum execution time so that only jobs that will not delay the start of the job at the head of the queue are backfilled. Furthermore, a preempted job is often given a reservation for a future time to run. Different methods of assigning reservations differentiate several variances of backfilling techniques [9], [18], [19]. Backfilling techniques address the low-utilization problem caused by different node number requirements of parallel jobs. However, backfilling does not deal with low resource utilization due to parallel jobs themselves.

Gang scheduling [10] allows resource sharing among multiple parallel jobs. The computing capacity of a node 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 sleep state when another process needs to communicate with it. The allocation of time slices of different nodes to parallel processes is coordinated, which requires OS support. Some gang scheduling algorithms, such as paired gang scheduling [20] investigate how to place processes with complement resource needs together to minimize their interference, e.g., when a process performs I/O activities and leaves CPU idle, the paired gang scheduling algorithm can find a process to use the idle CPU resources. A similar strategy is used in cloud resource consolidation through correlation analysis of resource use among VMs [21]. Processes of parallel jobs share the computing capacity of a node equally in common gang scheduling algorithms. This approach can improve the utilization to a certain degree, but is likely to stretch the execution time of individual jobs. There is attempt to integrate backfilling and gang scheduling [22], but it only results in a comparable performance to that of the simple backfilling algorithm [23].

Both backfilling and gang scheduling intend to improve utilization caused by node fragmentation. They do not target on the utilization degradation caused by parallelization itself.

3 Workload Consolidation Method

For a parallel application with dependency among its parallel processes, achieving high utilization on the nodes on which these processes run is often difficult. For a cloud service provider that runs this kind of applications, how to address this issue is important for its competitiveness in the market. We do two workload consolidation experiments in attempting to improve node utilization and examine the impact to the execution time of parallel jobs.

In the first experiment, we collocate two VMs in each physical node and give these VMs the same priority, i.e., each VM is assigned a *weight* of 256. In the second experiment, the collocated two VMs have different priorities, in which one is assigned a *weight* of 10,000 and the other is assigned a *weight* of 1. We call the high-priority one *foreground VM* and the low-priority one *background VM*. In this setting, the background VM only runs when the foreground VM is idle.

Throughout the experiments,[1] we made the following observations:

1. Priority-based VM collocation incurs trivial performance impact to jobs running in the high-priority VMs. The average performance loss of jobs running in the foreground tier is between 0.0 and 3.7 percent compared to those running in the nodes exclusively (one-tier VM). We simply model the loss as a uniform distribution.
2. When a foreground VM runs a job with a CPU utilization higher than 96 percent, collocating a VM to run in background does not benefit either the foreground or the background job due to that context switching incurs overhead and the background VM has very small chance to get physical resource 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 share of physical resources to run. For a single-process background job, the utilization of the idle CPU cycles is between 80 and 100 percent and roughly follows uniform distribution; for a multi-processes background job, the value is between 19.8 and 76.6 percent and can be modeled by a normal distribution with $\mu = 0.428$ and $\sigma = 0.144$.

Based on these observations, we will discuss our scheduling algorithms in the following section.

4 Scheduling Algorithms

In this section, we describe our scheduling algorithms for priority-based workload consolidation. We first discuss the basic scheduling algorithms and then give our consolidation strategies based on these algorithms.

4.1 Basic Algorithms

Our basic scheduling algorithm, Conservative Migration supported BackFilling (*CMBF*) is backfill based. The algorithm assumes that the state of a job can be saved and restored; therefore, the scheduler is able to suspend a job and resume it on other nodes in a later time.

CMBF schedules jobs to run according to their arrival time when there is enough number of nodes. When the number of idle nodes is not sufficient for a job, another job with a later arrival time but smaller node number requirement may be scheduled to run via backfilling. To avoid starving a preempted job, CMBF uses the following policy: *A preempted job is scheduled to run whenever it sees the total number of nodes that are either idle or occupied by jobs with a later arrival time is equal or greater than the number of nodes it needs.* The job may preempt jobs arriving later but being scheduled on some nodes. The scheduler instructs these jobs to save states, suspends their execution, and moves them back to the job queue.

1. The detail of our experiments can be found in Section 1 in the supplementary materials, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.262.

Algorithm 1 describes the detail of the process. The algorithm is activated when a new job arrives or a job finishes execution.

Algorithm 1: CMBF

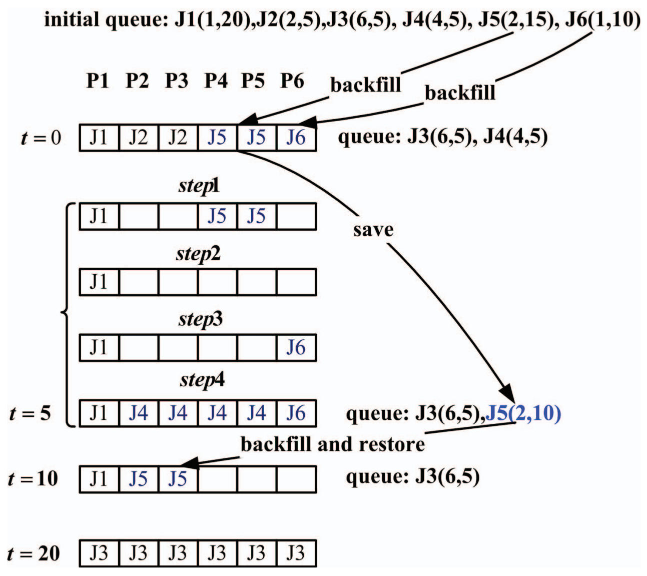
```
input :  $Q$: the queue for incoming jobs;
         $\mathcal{M}$: a map between jobs and nodes ;
output:  $\mathcal{M}'$: the updated allocation map;
1  begin
2      $j \leftarrow$ get the first job from $Q$;
3      while $j \neq null$ do
4          $N_j \leftarrow$ the number of nodes required by $j$;
5          $N_{idle} \leftarrow$ the number of idle nodes;
6          if $N_j \leq N_{idle}$ then
7              remove $j$ from $Q$ and dispatche it to any
               $N_j$ idle nodes;
8              update $\mathcal{M}$ accordingly;
9              if $j$ is not at the head of $Q$ then
10                 insert $j$ into $Q_{backfill}$ ;
11         else
12             $N_{backfill} \leftarrow$ the number of nodes running
               jobs arriving later than $j$;
13             if $N_j \leq (N_{backfill} + N_{idle})$ then
14                 suspend jobs in $Q_{backfill}$ that arrive
                   later than $j$ and move them back to $Q$
                   according to descending order of their
                   arrival time until the number of idle
                   nodes is greater than $N_j$;
15                 remove $j$ from $Q$ and dispatch it to
                   $N_j$ idle nodes;
16                 update $\mathcal{M}$;
17         $j \leftarrow$ get the next job from $Q$;
```

4.1.1 CMBF Example

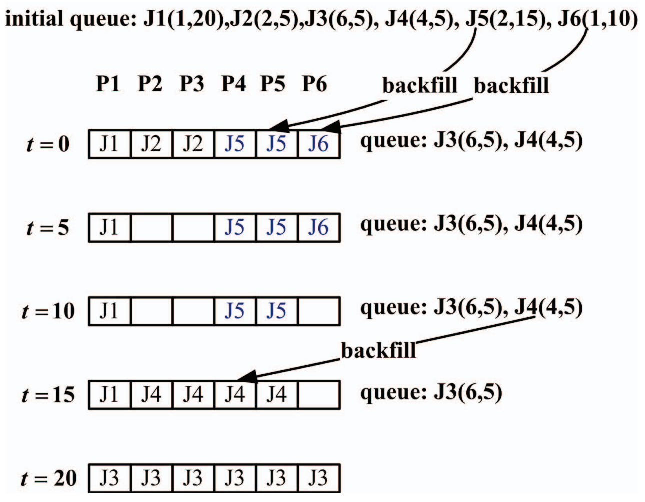
We illustrate how CMBF works using an example in Fig. 1a. $J_i(S, L)$ in Fig. 1 represents the job arrival order with i, the number of nodes requested by the job with S, and the execution time of the job with L. For simplicity of description, we here assume that the job suspension and restoring incur trivial cost in this example.

At time 0, there are six jobs in the queue. We consider that there are six nodes, labeled from P1 to P6 allocated to jobs in this queue. The node allocation layout is shown in the row labeled as $t = 0$. $J3$ and $J4$ are not allocated due to that their node requirements exceed the number of available nodes. As a result, $J5$ and $J6$ are backfilled into node P4, P5, and P6.

When $J2$ finishes its execution at time 5, CMBF tries to dispatch job $J3$ to run. $J3$ requests six nodes, but there are only two idle nodes and three additional nodes used by backfilling jobs $J3$ can preempt. $J3$ cannot be allocated according to CMBF (line 13 in Algorithm 1). However, $J4$ can be allocated as its request of four nodes can be satisfied through the combination of two idle nodes and two nodes used by a job ($J5$) it can preempt. Steps 1, 2, and 3 show the process that CMBF evicts backfilling jobs to make room for $J4$. First, collect the nodes used by $J6$, then $J5$, but only evicting $J5$ is enough for $J4$, thus, $J6$ remains in its original position without being removed. Then, the final allocation layout at time 5 is shown in step 4.



(a) Job scheduling in CMBF



(b) Job scheduling in AMBF

Fig. 1. Example of CMBF and AMBF.

When $J4$ and $J6$ finish execution at time 10, $J5$ is backfilled and resumes its execution as there is not enough number of nodes for $J3$ to run. $J3$ is only dispatched at time 20 when $J1$ and $J5$ finish execution.

4.1.2 Aggressive Migration Supported BackFilling (AMBF): A Simplified CMBF

In the worst case, CMBF requires tracking backfilling jobs for each job in the queue when making preemption decisions. When the number of jobs in the queue is large, the cost can be high. We here give a simplified algorithm called AMBF to address this problem.

Different to CMBF, AMBF only tracks backfilling jobs for the job at the head of the queue and allows the head-of-queue job to preempt other jobs. The rest of jobs in the queue are not allowed to preempt jobs, in another word, they can only be dispatched to idle nodes. The algorithm pseudocode of AMBF is similar to that of CMBF except that only the head-of-queue job executes the *else* (line 11 in Algorithm 1) code block.

We use a similar example in Fig. 1b to illustrate AMBF. After $J2$ departs at time 5 and $J6$ departs at time 10, $J3$ is at the head of the queue but the number of nodes it requests cannot be satisfied. There is no backfilling job for it to preempt either. As AMBF does not allow none-head-of-queue jobs to preempt, $J4$ cannot preempt $J5$ and is only dispatched to run when there is enough idle nodes at time 15.

As a job is less likely to preempt other jobs, AMBF also incurs job suspension and resuming less frequently than CMBF.

4.2 Scheduling with Workload Consolidation

The basic algorithms described above only consider mapping one parallel process to one node. As we described in Sections 1 and 3, node utilization can be low for these nodes due to that high efficiency in parallel computing is often difficult to achieve. In this section, we extend the basic algorithms to be node utilization aware in attempting to improve the overall node utilization in the cloud.

Based on our observation in Section 3, we divide the computing capacity of a physical node into two tiers, namely foreground and background. We assume that a physical node can run at most two VMs with one in the foreground and one in the background. The VM running in foreground is assigned a high CPU priority while the VM running in background is assigned a low CPU priority. In the following, we give a scheduling algorithm to handle two types of VM resources.

Conservative Migration and Consolidation supported BackFilling (CMCBF), as shown in three parts as Algorithms 2, 3, and 4, is based on the policy used in CMBF. It ensures that a job is dispatched to run in foreground VMs whenever the number of foreground VMs that are either idle or occupied by jobs arriving later than it satisfies its node requirement. Meanwhile, it allows jobs to run in background VMs simultaneously with those foreground VMs to improve node utilization. Compared to CMBF, CMCBF also deals with how to ensure that the background workload does not affect the foreground job. Note, CMCBF only dispatches a job to run in background VMs when the corresponding foreground VMs have a utilization lower than a given threshold (96 percent in our paper according to Section 3). The foreground VM utilization can be obtained from the profile of foreground jobs, or from the runtime monitoring data.

We use an example in Fig. 2 to illustrate the algorithm. We consider five nodes (P1-P5) for the job queue that has job $J1$ to $J10$ at the time of consideration. Each node has two-tier computing capacity denoted as *fg* and *bg* in Fig. 2. For simplicity of description in this example, we assume the cost of saving and restoring incur trivial cost, the process in a single-process job incurs a node utilization of 100 percent and the processes within a multiprocesses job involve a node utilization less than 96 percent, the VMs both in background and foreground have enough capacity to support the run of jobs.

Algorithm 2: CMCBF – On the depature of a foreground job

```
input : $Q$: the incoming job queue ;
        $J_{bg}$: a list of jobs running in the background;
        $\mathcal{M}$: the job allocation map;
output: $\mathcal{M}'$: the updated job allocation map;
1  begin
2      $j \leftarrow$ get the first job in $Q \cup J_{bg}$;
3      while $j \neq null$ do
4          $N_j \leftarrow$ the number of nodes needed by $j$;
5          $N_{fidle} \leftarrow$ the number of nodes with idle foreground VM;
6          if $N_j > N_{fidle}$ then
7              $N_{backfill} \leftarrow$ the number of nodes running jobs arriving later than $j$;
8              if $N_j \leq (N_{backfill} + N_{fidle})$ then
9                  switch $j$ to the background tier (if its background VMs are all idle) or save
                   and suspend backfilling foreground jobs of $j$ (otherwise), according to the
                   descending order of their arrival time until $N_{fidle} \geq N_j$;
10         if $N_j \leq N_{fidle}$ then
11             if $j \in J_{bg}$ then
12                 remove $j$ from $J_{bg}$;
13                 save the state of $j$ and suspend its execution on background;
14             else
15                 remove $j$ from $Q$;
16             dispatch(FG, $j$);
17         $j \leftarrow$ get the next job from $Q \cup J_{bg}$;
18 function dispatch(flag, j)
19 begin
20     sort the parallel processes of $j$ in descending order of their node utilization based on the profile of this type of jobs;
21     if $flag == FG$ then
22         $p \leftarrow$ sorted idle nodes in ascending order of their utilization (caused by the background load);
23         for each process $j_i$ in the sorted list do
24             place $j_i$ to $p_i$ and run $j_i$ in the foreground VM;
25             evict the background job if the utilization of $j_i$ is above the threshold;
26     else
27         $p \leftarrow$ sorted idle nodes in ascending order of their utilization (caused by the foreground load);
28         for each process $j_i$ do
29             place $j_i$ to $p_i$ and run $j_i$ in the background VM if $p_i$ is below the utilization threshold;
30         add $j$ to $J_{bg}$;
```

At time 0, job $J1$, $J2$, and $J3$ are allocated to five nodes and run in foreground VMs according to Algorithm 4. As $J1$ is a single-process job, therefore P1 cannot accommodate another VM running in background. However, $J4$ and $J5$ can run in background VMs at node P2-P5. How to collocate a background VM with which a foreground VM is determined through a simple process. The process matches the background VM that is likely to incur high node utilization to the foreground VM that is likely to incur low node utilization. The process is shown in the *dispatch* function in Algorithm 2. This matchmaking process balances the load on physical nodes and minimizes the interference between background and foreground jobs.

Algorithm 3: CMCBF – On the depature of a background job

```
input : $Q$: the incoming job queue ;
        $J_{bg}$: a list of jobs running in the background;
        $\mathcal{M}$: the job allocation map;
output: $\mathcal{M}'$: the updated job allocation map;
1  begin
2      remove the depature job from $J_{bg}$;
3      for each job $j \in Q$ do
4          $N_j \leftarrow$ the number of nodes needed by $j$;
5          $N_{bidle} \leftarrow$ the number of nodes with idle background VM;
6          if $N_j \leq N_{bidle}$ then
7              remove $j$ from $Q$;
8              dispatch(BG,$j$);
```

Algorithm 4: CMCBF – On the arrival of a new job

```
input : $\mathcal{M}$: the job allocation map;
        $j$: the new job;
output: $\mathcal{M}'$: the updated job allocation map;
1  begin
2      add $j$ to $Q$;
3      $N_j \leftarrow$ the number of nodes needed by $j$;
4      $N_{fidle} \leftarrow$ the number of nodes with idle foreground VM;
5      $N_{bidle} \leftarrow$ the number of nodes with idle background VM;
6      if $N_j \leq N_{fidle}$ then
7          remove $j$ from $Q$;
8          dispatch(FG, $j$);
9          return;
10     if $N_j \leq N_{bidle}$ then
11         remove $j$ from $Q$;
12         dispatch(BG, $j$);
```

At time 5, $J2$ finishes and departs the system. As there is not enough foreground VMs for $J4$ to run, $J5$ is backfilled from background to run in foreground according to lines 10-16 in Algorithm 2. As $J5$ is also a single-process job, therefore P2 cannot accommodate a background VM either. $J4$ is evicted and put back into the queue. $J6$ requests only one node and is backfilled to run on P3. $J6$ incurs a utilization above the threshold; therefore, P3 cannot accommodate a background VM. As there are only two background VMs available, $J4$ cannot run on them and $J7$ is placed to run on P4 and P5 as background job.

At time 10, $J1$ and $J3$ finish execution. $J4$ is at the head of the queue and dispatched to run in foreground. $J10$ is dispatched to run in background at P1.

At time 15, $J4$ finishes and $J10$ is moved to run in foreground due to that there are not enough foreground VMs for $J8$ and $J9$.

At time 20, $J6$ finishes. The total number of idle foreground VMs and VMs running backfilling jobs ($J10$ in this case) satisfies the needs of $J9$. As a result, $J10$ is evicted and $J9$ is allocated to run on P1, P3-P5. Subsequently, $J10$ is put to run as a background job in $P1$ ($J10$ is actually switched from

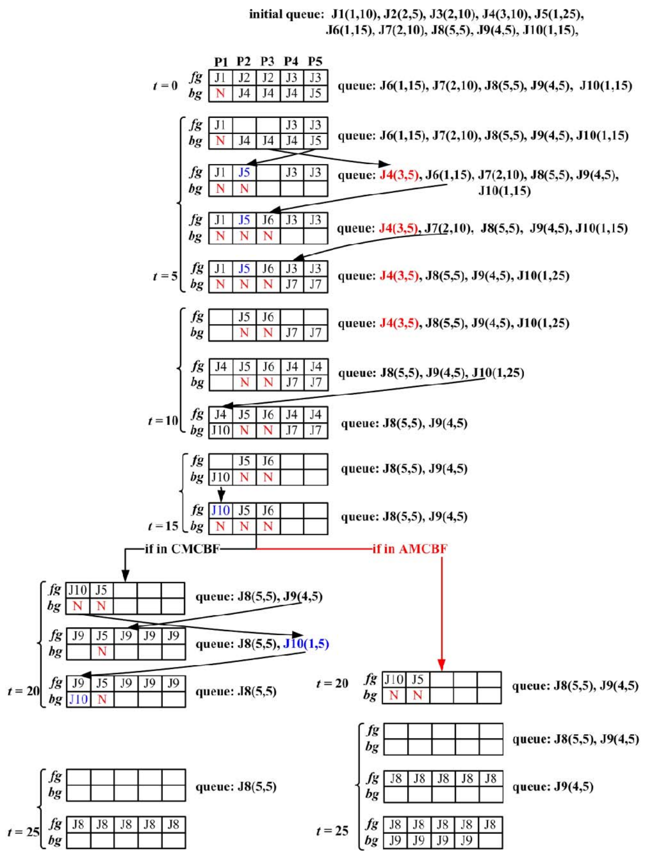


Fig. 2. Example of CMCBF and AMCBF.

the foreground to the background VM by swapping the CPU priorities of the foreground VM and background VM of P1).

At time 25, $J8$ is allocated when other jobs finish and foreground VMs are available.

The handling of background job departure and job arrival are straightforward as Algorithms 3 and 4 show. When a job arrives, if enough idle VMs exist in foreground, this newly arrival job will be placed into foreground, else if enough idle VMs exist in background, the destination of the newly arrival job is background, otherwise, push it at the end of the job queue.

CMCBF treats jobs running in background equally. In other words, a job running in the background will not be preempted by other jobs if the foreground situation does not change. This reduces suspension and state restoring cost of background jobs. When a background job departs, the scheduler just scans the queue according to job arrival time and place a matching job to run in the available background VMs.

CMCBF is similar to CMBF when it comes the way it handles foreground jobs; however, CMCBF differs to CMBF in the following two ways:

1. CMCBF considers the jobs both running in background and waiting in the queue when making scheduling decisions, not only jobs in the queue.
2. When moving a job from background to foreground in CMCBF, some parallel processes of the job do not have to be suspended and restored, instead they can be switched to run in foreground through the change of VM priority.

CMCBF faces similar problem as CMBF when tracking backfilling jobs for each job. To reduce the cost, we also simplify the process in a way similar to AMBF by only keeping the backfilling job list for the job at the head of the queue. We call the modified algorithm AMCBF.

Fig. 2 also includes how the example case is handled in AMCBF. When $J6$ departs at time 15, $J9$ can not preempt $J10$ as $J8$ is at the head of the queue and only the head-of-queue job can preempt other jobs in AMCBF.

Similar to CMBF, CMCBF has the following property: A job is dispatched immediately whenever it has sufficient resource to run. The resource includes idle nodes and those nodes occupied by jobs arriving later than it. CMCBF improves the node utilization by allowing jobs to run in background. AMBF and AMCBF slightly relax the dispatching order of CMBF and CMCBF by preventing none head-of-the-queue jobs from preempting.

5 Evaluation

5.1 Settings

We evaluate our algorithms using trace-driven simulation.

Running a foreground VM and a background VM simultaneously on a physical node incurs overhead due to context switch. According to our experimental results in our cluster, we model the overhead to the process running in the foreground VM as a random number varying from 0 to 3.7 percent of the length of a time slice. In a time slice, the progress of a background process is calculated as below:

$$t = \begin{cases} T & \text{if its } fg \text{ VMs are all idle} \\ T \cdot eff & \text{if } CPU_{idle} \geq CPU_{req} \\ T \cdot eff \cdot \dfrac{CPU_{idle}}{CPU_{req}} & \text{if } CPU_{idle} < CPU_{req}, \end{cases}$$

in which T is the length of a time slice, denoting the progress of the background process running on a dedicated node in a time slice. eff is a variable between 0 and 1 that measures how much time in the slice effectively contributes to the progress of the process. A background process is frequently preempted and eff characterizes the overhead associated. For a single-process job, eff is a random number between 0.8 and 1; for a multiprocesses job, eff (between 0.2 and 0.8) 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 node. CPU_{idle} is the portion of unused CPU cycles in the node, i.e., the portion that is not fully utilized by the foreground VM. We set the lower bound of CPU_{idle} as 4 percent. When the portion of idle CPU cycles in a node is equal to or below the threshold, no process will be dispatched to the node to run as a background process. The threshold setting is obtained from the observation in our experiments.

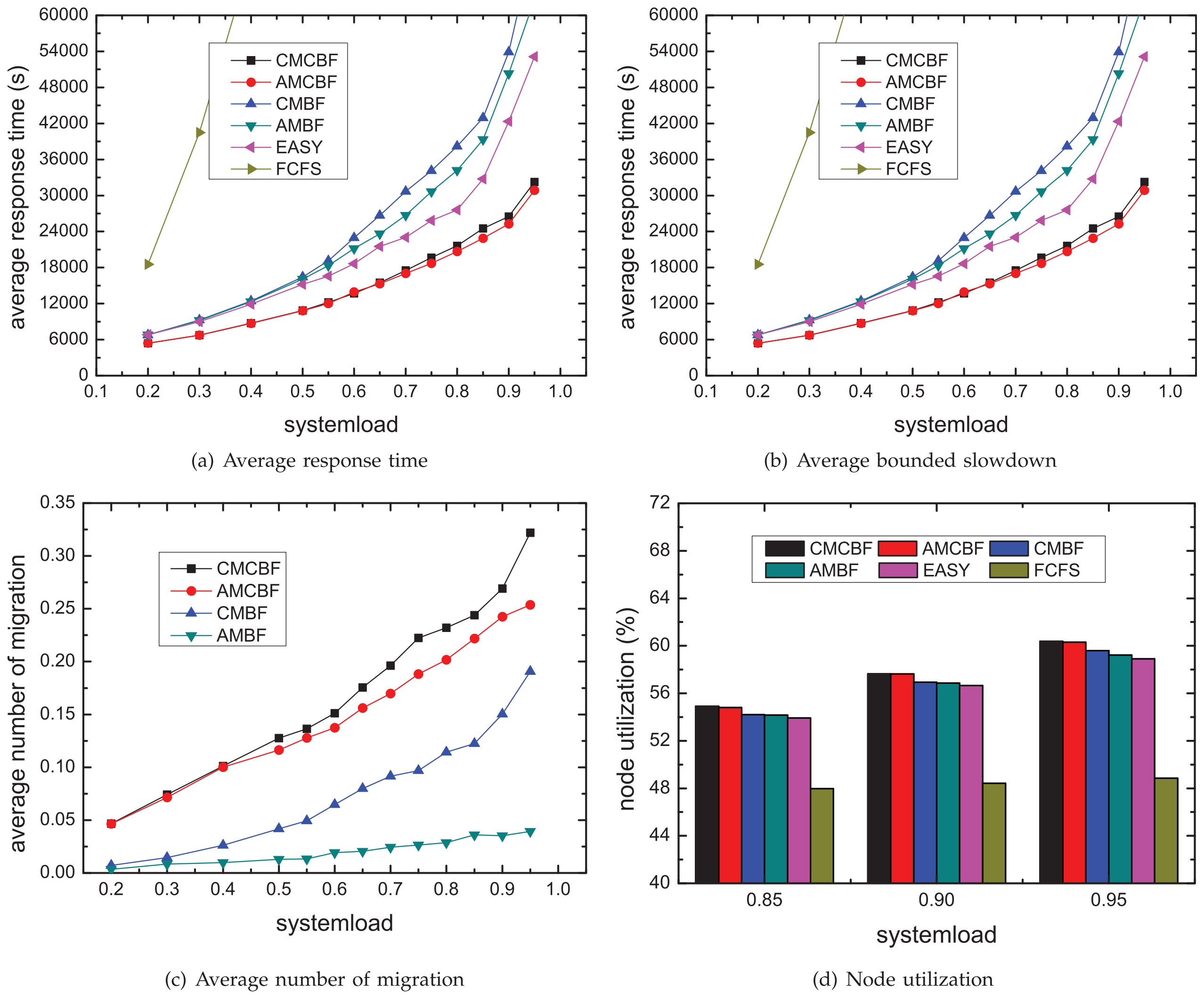
Furthermore, the progress of a job depends on the progress of the slowest process in the job.

In our simulation, we set the job migration cost to 20 seconds. The total number of nodes is 320 and the number of parallel processes in a job is set between 1 and 256 when using workload models.

5.2 Workload and Performance Metrics

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

1. Feitelson workload, denoted by *FWorkload*: A general model based on data from six different traces [24], [18]. It contains 20,000 jobs in our simulation.



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

(c) Average number of migration (d) Node utilization

Fig. 3. Performance comparison using FWorkload.

2. Jann workload, denoted by *JWorkload*: A workload model for MPP and it fits the actual workload of Cornell Theory Center Supercomputer [25]. It contains 10,000 jobs in our simulation.

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 rule: 1) if a job has only one process, the process is assigned a CPU usage of 100 percent; 2) if a job has more than one processes, the CPU usage of each process is a random number between 40 to 100 percent.

The workload is then characterized as the following:

- t_i^r: the execution time of job i in a dedicated node.
- t_i^a: the arrival time of job i.
- t_i^f: the finish time of job i.
- b_i: the number of nodes requested by job i.
- CPU_i^j: the average CPU usage of process j of job i.

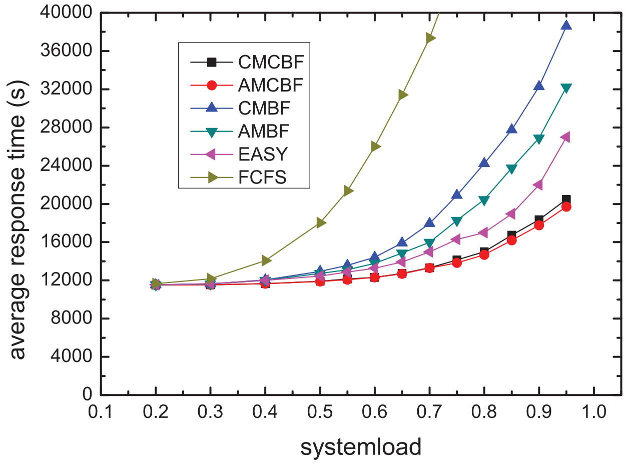
We use the following performance metrics to evaluate our algorithms:

- $rpt = \frac{\sum_{i=1}^{n}(t_i^f - t_i^a)}{n}$: the average response time.
- $b\_sld = \frac{\sum_{i=1}^{n}\frac{t_i^f - t_i^a}{max(\Gamma, t_i^r)}}{n}$: the average bounded slowdown. Compared with slowdown, the bounded slowdown metric is less affected by very short jobs as it contains a minimal execution time element Γ [18], [26]. According to [18], we set Γ to 10 seconds.
- mig_num: the average number of migrations per job.
- $utilization$: the average node utilization, which is the fraction of busy CPU cycles in the total simulation time.

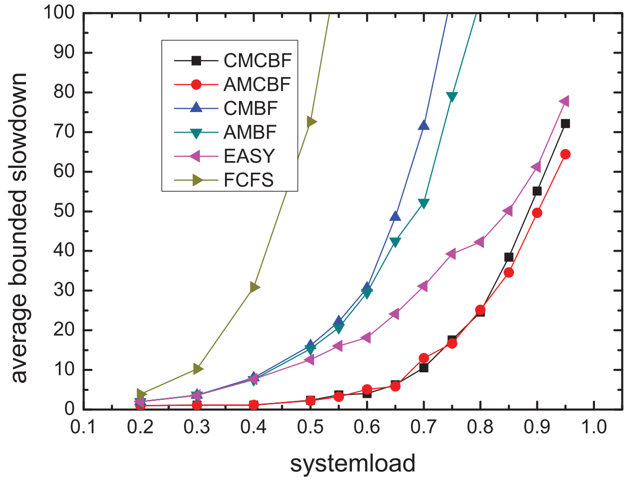
5.3 The Results

Figs. 3 and 4 show the performance of our algorithms for FWorkload and JWorkload. In comparison, we give the EASY scheduler accurate estimation of job execution time, but keep the execution time unknown from our algorithms. We have the following observations from Figs. 3 and 4:

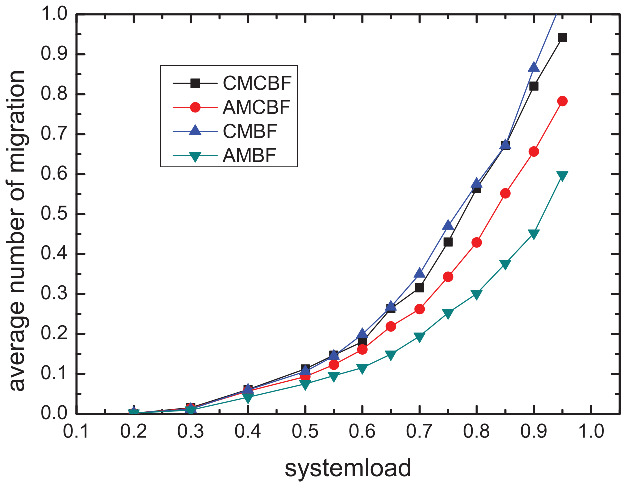
1. Our basic algorithms, CMBF and AMBF, outperform FCFS in big margin but produce worse response time and bounded slowdown than EASY. It is not a surprise as EASY performs well when the estimation of job execution time is accurate. However, our algorithms that consolidate parallel workload, CMCBF and AMCBF, significantly outperform FCFS, CMBF, AMBF, and EASY on response time and bounded slowdown.



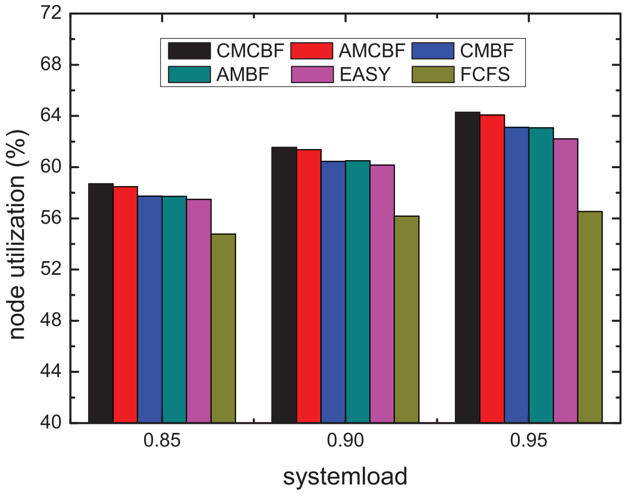
(a) Average response time



(b) Average bounded slowdown



(c) Average number of migration



(d) Node utilization

Fig. 4. Performance comparison using JWorkload.

2. The two consolidation algorithms lead to better node utilization compared to other algorithms. Compared with EASY, a utilization improvement up to 2.4 and 3.1 percent can be gained in FWorkload and JWorkload, respectively. And, moreover, the improvements on the saturation CPU utilization are 11 and 16 percent, respectively.
3. By relaxing the job execution order and only disallowing preemption to the head-of-the-queue job, AMBF achieves better performance than CMBF, the one that preserves the order in terms of job response time and bounded slowdown. This can be explained by that short jobs get more chances to run with the relaxation. AMCBF also shows slightly better performance than CMCBF. The improvement is not significant due to that the consolidation mechanism in CMCBF already gives short jobs enough chance to run in the background.
4. The average number of job migrations in AMCBF is slightly less than that in CMCBF for both FWorkload and JWorkload. But, AMBF results in significantly less number of job migrations than CMBF for the two workloads. Generally, the two algorithms with relaxed job execution order often require less number of job migrations than those preserving the order. It is due to that AMBF and AMCBF treat all none-head-of-the-queue jobs equally, thus the chance that a job preempts another is reduced to a certain degree. We note that there are more job migrations in CMBF than in AMCBF and CMCBF, as shown in the JWorkload results. This is due to that the computing capacity left by the foreground VMs allows many short jobs to be accommodated in the background VMs without incurring job migration. We also note that the average number of job migrations in JWorkload is larger than that in FWorkload. It is because of the characteristic of JWorkload. JWorkload contains about 40 percent of single-process jobs and there are about 86 percent of jobs with less than 20 processes in JWorkload [25]. The large number of small jobs triggers many backfilling operations, thus increases the chance of job migrations.

In the results described above, AMCBF shows better performance than other algorithms especially in terms of response time and bounded slowdown. We will use AMCBF for comparison in the following discussions.

5.4 Discussions

AMCBF uses the CPU usage information of parallel processes to make scheduling decisions. We study the impact of the accuracy of CPU usage information on the performance of AMCBF. In addition, job migration cost and the amount of remaining computing capacity on each node also have impact on the performance of our scheduling algorithms. We further investigate the impact of these two factors on the scheduling performance. Our experimental results[2] show the following:

1. AMCBF's performance improves as the accuracy of CPU usage estimation increases. And, moreover, AMCBF performs better than EASY in most cases even without any CPU usage information of parallel processes.
2. The performance of AMCBF degrades as the migration cost increases, but AMCBF bears high migration cost. For FWorkload, even with a migration cost at 600 seconds (24 percent of the average job execution time), AMCBF can still outperform EASY in most cases; for JWorkload, the threshold is 360 seconds (3.2 percent of the average job execution time).
3. The benefit gained from consolidation is inversely proportional to the average CPU usage of parallel processes. For FWorkload, if the average CPU usage is less than 95 percent, AMCBF outperforms EASY in most cases. But for JWorkload, the upper threshold reduces to about 80 percent.

6 Conclusion

As an increasing number of complex applications leverages the computing power of the cloud for parallel computing, it becomes important to efficiently manage computing resources for these applications. Due to the difficulty in realizing parallelism, many parallel applications show a pattern of decreasing resource utilization along with the increase of parallelism. Scheduling parallel jobs for both efficient resource use and job responsiveness is challenging.

Workload consolidation supported by virtualization technologies is commonly used for improving utilization in data centers. In this paper, we gave a priority-based workload consolidation method to schedule parallel jobs in data centers to make use of under utilized node computing capacity to improve responsiveness. Our method partitions a node's computing capacity into the foreground VM (with high CPU priority) tier and the background VM (with low CPU priority) tier. The performance of jobs running in the foreground VMs is close to that of jobs running in dedicated nodes (less than 3.7 percent performance loss in our experiments); meanwhile, the idle CPU cycles can be well used by the jobs running in background VMs. The algorithms we gave integrated backfilling and migration techniques to make effective use of the two types of VMs. Our extensive simulation showed that our consolidation-based algorithm (AMCBF), even without knowing the job execution time, significantly outperforms the commonly used EASY algorithm. In addition, AMCBF is robust in terms that it allows inaccurate CPU usage estimation of parallel processes.

In our future work, we will exploit mechanisms that can effectively partition the computing capacity of a data center node into *k-tiers*, which may further improve the node utilization and responsiveness for parallel workload in the cloud. Another issue is that in a large data center, processes of a job may need to be allocated to nodes that are close to each other to minimize the communication cost. Our current method does not take this into account. As the future work, we will exploit buddy allocation mechanisms [27], [28] to tackle this issue.

2. For details, please refer to Section 2 in the supplementary materials, available online.

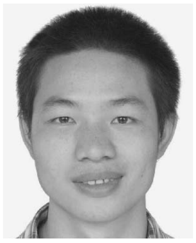
Acknowledgments

The work was supported by the National Natural Science Foundation of China (Nos. 91024030, 61074108, 61172014, and 60702037). Professors Albert Y. Zomaya and Bing Bing Zhou would like to acknowledge the support of the Australian Research Council Discovery Grant (DP1097111).

References

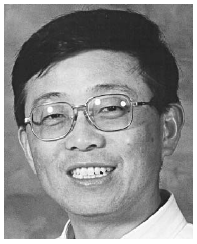
[1] "High Performance Computing (HPC) on AWS,"Amazon Inc., http://aws.amazon.com/hpc-applications/, 2011.
[2] L. Barroso and U. Holzle, "The Case for Energy-Proportional Computing," *Computer,* vol. 40, no. 12, pp. 33-37, Dec. 2007.
[3] J. Hamilton, "Cloud Computing Economies of Scale," *Proc. AWS Genomics Cloud Computing Workshop,* http://www.mvdirona.com/jrh/TalksAndPapers/JamesHamilton_GenomicsCloud20100608.pdf, 2010.
[4] D. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems.* IBM TJ Watson Research Center, 1994.
[5] D. Feitelson and B. Nitzberg, "Job Characteristics of a Production Parallel Scientific Workload on the Nasa Ames ipsc/860," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 337-360, 1995.
[6] J. Jones and B. Nitzberg, "Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 1-16, 1999.
[7] L.G. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM,* vol. 33, no. 8, pp. 103-111, 1990.
[8] U. Schwiegelshohn and R. Yahyapour, "Analysis of First-Come-First-Serve Parallel Job Scheduling," *Proc. Ninth Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 629-638, 1998.
[9] D. Lifka, "The Anl/Ibm SP Scheduling System," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 295-303, 1995.
[10] D. Feitelson and M. Jettee, "Improved Utilization and Responsiveness with Gang Scheduling," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 238-261, 1997.
[11] Y. Lin, "Parallelism Analyzers for Parallel Discrete Event Simulation," *ACM Trans. Modeling and Computer Simulation,* vol. 2, no. 3, pp. 239-264, 1992.
[12] Z. Juhasz, S. Turner, K. Kuntner, and M. Gerzson, "A Performance Analyser and Prediction Tool for Parallel Discrete Event Simulation," *J. Simulation,* vol. 4, no. 1, pp. 7-22, 2003.
[13] R. Fujimoto, "Parallel and Distributed Simulation," *Proc. 31st Conf. Winter Simulation: Simulation—A Bridge to the Future,* vol. 1, pp. 122-131, 1999.
[14] A. Malik, A. Park, and R. Fujimoto, "Optimistic Synchronization of Parallel Simulations in Cloud Computing Environments," *Proc. IEEE Int'l Conf. Cloud Computing (CLOUD '09),* pp. 49-56, 2009.
[15] R. Fujimoto, A. Malik, and A. Park, "Parallel and Distributed Simulation in the Cloud," *Int'l Simulation Magazine, Soc. for Modeling and Simulation,* vol. 1, no. 3, 2010.
[16] G. D'Angelo, "Parallel and Distributed Simulation from Many Cores to the Public Cloud," *Proc. Int'l Conf. High Performance Computing and Simulation (HPCS),* pp. 14-23, 2011.
[17] Y. Etsion and D. Tsafrir, "A Short Survey of Commercial Cluster Batch Schedulers," Technical Report 2005-13, The Hebrew Univ. of Jerusalem, May 2005.

[18] A. Mu'alem and D. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM sp2 with Backfilling," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 6, pp. 529-543, June 2001.
[19] D. Jackson, Q. Snell, and M. Clement, "Core Algorithms of the Maui Scheduler," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 87-102, 2001.
[20] Y. Wiseman and D. Feitelson, "Paired Gang Scheduling," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 6, pp. 581-592, June 2003.
[21] A. Do, J. Chen, C. Wang, Y. Lee, A. Zomaya, and B. Zhou, "Profiling Applications for Virtual Machine Placement in Clouds," *Proc. IEEE Int'l Conf. Cloud Computing (CLOUD),* pp. 660-667, 2011.
[22] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 3, pp. 236-247, Mar. 2003.
[23] D. Tsafrir, Y. Etsion, and D. Feitelson, "Backfilling Using System-Generated Predictions Rather than User Runtime Estimates," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 6, pp. 789-803, June 2007.
[24] D. Feitelson, "Packing Schemes for Gang Scheduling," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 89-110, 1996.
[25] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan, "Modeling of Workload in Mpps," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 95-116, 1997.
[26] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and Practice in Parallel Job Scheduling," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 1-34, 1997.
[27] E. Von Puttkamer, "A Simple Hardware Buddy System Memory Allocator," *IEEE Trans. Computers,* vol. C-100, no. 10, pp. 953-957, Oct. 1975.
[28] K. Knowlton, "A Fast Storage Allocator," *Comm. ACM,* vol. 8, no. 10, pp. 623-624, 1965.



Xiaocheng Liu received the PhD degree from the National University of Defense Technology and he is now a lecturer at the same university. His recent research interests include resource allocation in the cloud, parallel and distributed simulation, component-based modeling. He had a year's (from November 2010 to November 2011) training in cloud computing at the Centre for Distributed and High Performance Computing, School of Information Technologies, the University of Sydney.

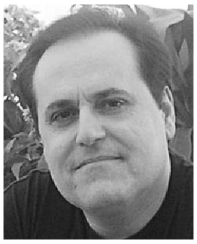
Chen Wang received the PhD degree from Nanjing University. He is a senior research scientist at Commonwealth Scientific and Industrial Research Organisation ICT Centre, Australia. His research interests are primarily in distributed, parallel and trustworthy systems. His current work focuses on resource management in cloud computing, accountable distributed systems and demand response algorithms in the smart grid.



Bing Bing Zhou received the BS degree from the Nanjing Institute of Technology, China, and the PhD degree in computer science from Australian National University. He is currently an associate professor at the University of Sydney. His research interests include parallel/distributed computing, grid and cloud computing, peer-to-peer systems, parallel algorithms, and bioinformatics. He has a number of publications in leading international journals and conference proceedings. His research has been funded by the Australian Research Council through several Discovery Project grants.

Junliang Chen is currently working toward the PhD degree at the University of Sydney. His research interests include resource management and job scheduling in distributed systems, economic models in cloud computing and software as a service.

Ting Yang is currently an associate professor at the School of Electrical Engineering and Automation, Tianjin University, China. His scientific interests include robust communication networks (interlayer/interdomain protocols and quality of service in data center network), reliable cloud computing infrastructure and parallel and distributed algorithms. His research has been supported by the National Science Foundation of China and the Ministry of Education of China. He has published more than 60 papers in major journals and refereed conference proceedings. He is also the coauthor of book *Modern Sensor Technology*. He is a senior member of the Chinese Institute of Electronic, the fellow of Graph Theory Research and Application committee, and the member of International Society for Industry and Applied Mathematics.



Albert Y. Zomaya is currently the chair professor of High Performance Computing and Networking and Australian Research Council Professorial fellow at the School of Information Technologies, The University of Sydney. He is also the director of the Centre for Distributed and High Performance Computing which was established in late 2009. He is the author/coauthor of seven books, more than 400 publications in technical journals and conferences, and the editor of nine books and 11 conference volumes. His research interests are in the areas of parallel and distributed computing and complex systems. He is currently the editor-in-chief of the *IEEE Transactions on Computers* and serves as an associate editor for 19 journals including some of the leading journals in the field. He was the chair the IEEE Technical Committee on Parallel Processing (1999-2003) and currently serves on its executive committee. He also serves on the advisory board of the IEEE Technical Committee on Scalable Computing, the advisory board of the Machine Intelligence Research Labs. He served as general and program chair for more than 60 events and served on the committees of more than 500 ACM and IEEE conferences. He delivered more than 100 keynote addresses, invited seminars and media briefings. He is a fellow of the IEEE, AAAS, the Institution of Engineering and Technology, United Kingdom, a distinguished engineer of the ACM and a chartered engineer. He received the 1997 Edgeworth David Medal from the Royal Society of New South Wales for outstanding contributions to Australian Science. He also received the IEEE Computer Society's Meritorious Service Award and Golden Core Recognition in 2000 and 2006, respectively. Also, he received the IEEE TCPP Outstanding Service Award and the IEEE TCSC Medal for Excellence in Scalable Computing, both in 2011.